A High Accuracy Volume Renderer for Unstructured Data

Peter L. Williams, *Member*, *IEEE Computer Society*, Nelson L. Max, and Clifford M. Stein, *Member*, *IEEE*

Abstract—This paper describes a volume rendering system for unstructured data, especially finite element data, that creates images with very high accuracy. The system will currently handle meshes whose cells are either linear or quadratic tetrahedra. Compromises or approximations are not introduced for the sake of efficiency. Whenever possible, exact mathematical solutions for the radiance integrals involved and for interpolation are used. The system will also handle meshes with mixed cell types: tetrahedra, bricks, prisms, wedges, and pyramids, but not with high accuracy. Accurate semitransparent shaded isosurfaces may be embedded in the volume rendering. For very small cells, subpixel accumulation by splatting is used to avoid sampling error. A revision to an existing accurate visibility ordering algorithm is described, which includes a correction and a method for dramatically increasing its efficiency. Finally, hardware assisted projection and compositing are extended from tetrahedra to arbitrary convex polyhedra.

Index Terms—Volume rendering, unstructured meshes, high accuracy, finite element method, isosurfaces, splatting, cell projection, visibility ordering, depth sorting.

1 Introduction

Typically, unstructured meshes have a complex geometric configuration and the mathematics of the absorption-emission integral are quite complex. Therefore, most existing volume rendering systems for unstructured data [5], [6], [7], [8], [10], [12], [13], [15], [16], [23], [29], [30], [31], [33], [34], [35], [37], [41] introduce various simplifying assumptions and approximations into the algorithm in order to cope with these complexities in an efficient manner.

Another aspect of unstructured meshes is that, typically, they are adaptively refined, so that, in areas where the field is changing rapidly, the cells are smaller than in other areas of the mesh. It is not uncommon for such cells to be several orders of magnitude smaller than the largest cells. The behavior of the field on these smallest cells is often of great interest to the simulation scientist. However, all volume rendering systems that we are aware of are liable to miss these smaller cells due to sampling error.

This paper describes a high accuracy (HIAC) volume rendering system for unstructured data, especially finite element data, that, for a given mathematical optical model [17], creates images with very high accuracy. Compromises, or approximations, are not introduced for the sake of efficiency. Whenever possible, exact mathematical solutions for the differential equations involved and for interpolation are used. Subpixel accumulation by splatting is used to avoid sampling error. Accurate semitransparent shaded isosurfaces may be embedded in the volume rendering. In addition, a modified version of the accurate visibility ordering algorithm for unstructured meshes, reported by Stein et al. [31], is used. Several important revisions to the original sorting algorithm, including a correction and a method for dramatically improving its efficiency, are described herein.

Our goal was to design a volume rendering system to create benchmark images for use as a standard of comparison. The benchmarks can be used to compare results from other volume rendering systems for unstructured data that use approximations and simplifying assumptions, and can serve as a validation suite for verifying the correctness of new algorithms and implementations.

The HIAC volume rendering system is based on the absorption plus emission optical model [17], [27], [38] and utilizes the cell projection method to accumulate the image. A ray integration is performed individually for every pixel onto which a cell projects. The system will correctly render images in both parallel and perspective projection, provided the transfer functions for color and opacity are piecewise linear. It is intended primarily for data sets from the finite element method, but will render any unstructured data set whose cells are tetrahedra, bricks, prisms, or pyramids, or any combination thereof; see Fig. 1. The meshes may be nonconvex or even disconnected; the faces of adjacent cells may meet on only part of their common adjacent face, i.e., sliding interfaces are permitted. However, the cells are expected to be convex and nonintersecting, and the visibility ordering graph should not contain cycles.

The system will accurately render data sets where the scalar field varies linearly along the edges of the cells, called *linear cells* or *linear elements*. For linear tetrahedra, the system uses the exact solution to the radiance integral described in [38]. This paper shows how the exact solution can be implemented utilizing the Dawson integral [24], rather than the table-based method described in [38].

• *P.L. Willaims is with the IBM T.J. Watson Research Center, H0-C10, 30 Saw Mill River Road, Hawthorne, NY 10532.*
E-mail: p.williams@computer.org.
• *N.L. Max and C.M. Stein are with the Lawrence Livermore National Laboratory, Mail Stop L-307, 7000 East Avenue, Livermore, CA 94550.*
E-mail: {max2, stein2}@llnl.gov.

For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number 106328.

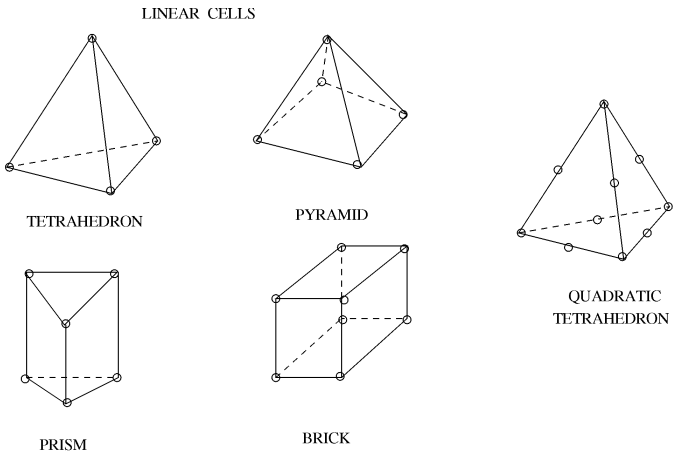


Fig. 1. Examples of different types of cells used in the finite element method. The HIAC system will render data defined on meshes with any of these cell types.

In addition, the HIAC system will accurately volume render *quadratic tetrahedra* (tetrahedra where the scalar field varies quadratically along the edges of the cells, and, in fact, along any ray through the cell). In Appendix C, we describe how the system could be extended to accurately render other higher-order elements, such as quadratic bricks, as well as linear bricks and prisms.

This paper is an amalgamation and extension of the results in [16], [31], [38]. The main new additions are the use of the Dawson integral in the computation of the exact solution of the radiance integral for linear tetrahedra, the methodology for accurate radiance integration and isosurface generation for quadratic tetrahedra, the use of splatting for subpixel accumulation, the revision to Stein's visibility ordering algorithm, and the extension of hardware-assisted projection and compositing to arbitrary convex polyhedra.

The next section discusses related previous work. Section 3 discusses the geometry of the cells used in the finite element method, the related interpolation equations, and relevant terminology. Section 4 gives a broad overview of the system and, then, in Sections 5 and 6, we present the details of the rendering system and of the visibility ordering algorithm. Section 7 discusses hardware assisted polyhedron projection. Section 8 presents timing results and example images.

2 Previous Work

A method for approximating the volume rendering integral with bounded error is described by Novins and Arvo in [21]. By bounding the magnitude of the derivatives of the integrand, they are able to obtain remainder terms that provide bounds on the approximation error. They apply this to the trapezoid rule, Simpson's rule, and a power series method. The first two methods are more suited to low to medium accuracy approximations. The power series method, on the other hand, is preferable for very high precision results.

The techniques developed by Novins and Arvo are very valuable for bounding the error in the evaluation of the integral. However, there are other sources of error in the volume rendering process, e.g., sampling error, which may miss small but highly important cells in the accumulation process, that may be even more significant than integration error. The HIAC system addresses some of these other sources of error. For example, it uses subpixel splatting and a high accuracy visibility ordering algorithm.

For high accuracy integration, the HIAC system uses a closed-form solution to the integral when possible; otherwise, high accuracy Gaussian numerical integration is used. This approach appears to be more efficient than the power series method, since the power series error bounds are loose. (Novins did not provide timing data for comparative purposes.) The error bounds are not easy to calculate for Gaussian quadrature, but it is known to be very accurate, and it is the quadrature method generally used in the finite element method. When a guaranteed error bound is required for integration on higher-order elements, the Novins and Arvo power series approximation may be valuable. It is an open question whether Gaussian quadrature or the power series method described by Novins is preferable for high accuracy integration.

Silva and Mitchell [30] describe a very efficient and interesting sweep plane volume rendering method that accurately traverses all types of tetrahedral meshes with nonintersecting cells, even those with cyclically overlapping cells. They claim it can be extended in a straightforward way to more complex convex cells. The real value of the sweep plane algorithm is that it provides a very efficient and accurate depth ordering of the cells of an irregular mesh along any given ray to the eye; it does not try to give a global visibility ordering of the cells. The mathematics of the volume rendering integral is not addressed in their paper, nor is sampling error. The integration methods described in our paper could be utilized in the sweep plane algorithm.

Gallagher and Nagtegaal [4] describe methods for rendering 3D contour surfaces of finite element data, as well as methods for smooth shading these surfaces. They render the contour surfaces, which may be curved within a cell, as a polygonal approximation to a parametric bicubic surface fit to each contour in a cell, whereas we render these same surfaces on a pixel-by-pixel basis to reproduce the exact implicit curved surface and use Phong shading calculated at each pixel. Cline et al. [1] also reproduce this curved surface by recursive subdivision of the volume cells containing the contour surface.

3 Cell Geometry and Interpolation Functions

The cells used for 3D modeling in the finite element method (FEM) have many different shapes, but only a few are in widespread use [14]. We will focus our attention on the more commonly used 3D cells (also called *elements*): the tetrahedron, brick, and prism. See Fig. 1.

In addition to the vertices (also referred to as *nodes*) used to define the endpoints of a cell's edges, which we will call the *conventional vertices* or *nodes*, a cell may have additional vertices, which we will call *interior nodes*, see, for example, the quadratic tetrahedron in Fig. 1. The interior nodes, along with the conventional nodes, may be used:

1) to define a nonlinear field inside the cell by the use of what we will refer to as an *interpolation function*; and/or

2) to define curvilinear facets by the use of a parametric mapping function.

In this paper, we will not deal with elements whose geometry is defined by a parametric mapping, since those elements may have facets that are highly curved, and the parametric mapping must be inverted before the scalar function can be evaluated. We will limit our consideration to the first category of cells. For those cells, the scalar field value is specified at all vertices, conventional and interior; but the geometry of the cell is determined from its conventional vertices.

The number of terms in a cell's interpolation function is equal to the number of nodes that the cell has. So, a tetrahedron with four nodes will have an interpolation function with four terms. In most applications, the interpolation function is a polynomial whose terms are elements of the three-dimensional power series. Those terms through third degree are:

$$\begin{gathered} 1 \\ x, y, z \\ x^2, y^2, z^2, xy, xz, yz \\ x^3, y^3, z^3, x^2y, x^2z, xy^2, xz^2, y^2z, yz^2, xyz. \end{gathered} \quad (1)$$

The interpolation function for a four-node tetrahedron is:

$$f(x, y, z) = c_1 + c_2x + c_3y + c_4z.$$

The scalar field varies linearly along any ray through a four-node tetrahedron, hence, it is called a linear tetrahedron.

A brick with eight nodes has the eight-term interpolation function:

$$f(x, y, z) = c_1 + c_2x + c_3y + c_4z + c_5xy + c_6xz + c_7yz + c_8xyz. \quad (2)$$

The particular terms of the 3D power series that are chosen for a given interpolation function are dictated by the need of the FEM for certain desirable properties, such as symmetry, nonsingularities, etc. Here, the scalar field varies linearly along the edges of the brick and, so, it is sometimes called a linear brick. However, the field inside the brick varies trilinearly, so it is also called a trilinear brick. Others refer to it as an eight-node brick, or a hexahedron. Often it is the case, in the FEM, that nontriangular facets are slightly nonplanar.

Although the four-node tetrahedron and the eight-node brick are both referred to as linear elements, the higher-order terms in the interpolation equation for the linear brick give it extra degrees of freedom that allow it to solve some problems much more accurately than could be done with tetrahedra alone. From the perspective of visualization, it should be noted that a contour surface inside an eight-node brick is curved and not planar, as it is inside a four-node tetrahedron.

The tetrahedron, brick, and prism are the basic cells. They are often referred to as linear cells, since the field varies linearly along the edges of the cells. By adding interior nodes to the basic cells, we get cells with higher-order interpolation functions. We refer to this class of cells as *higher-order cells*. There are three important higher-order cells.

The first is the 10-node tetrahedron, also referred to as a quadratic tetrahedron, whose interpolation function is:

$$f(x, y, z) = c_1 + c_2x + c_3y + c_4z + c_5x^2 + c_6y^2 + c_7z^2 + c_8xy + c_9xz + c_{10}yz. \quad (3)$$

This function is complete through the quadratic terms of the 3D power series in (1), therefore, the field varies quadratically along any ray through the volume. In the FEM, the six interior nodes may be specified in different configurations; however, the most common configuration is for the interior nodes to be located on the edges of the cell, usually at the midpoints. Elements where all of the nodes lie on the boundary of the element are called *serendipity* elements. Serendipity elements are the most common 3D elements.

The remaining higher-order cell types, the cubic tetrahedron and the quadratic brick, as well as the prism, are discussed in Appendix C.

4 Overview of the HIAC System

The HIAC volume rendering system for unstructured meshes uses the cell projection method and is based on the absorption plus emission volume density optical model [17]. Either the Williams and Max [38] or the Wilhelms and Van Gelder [33] treatment of glow energy may be specified for use.

The system reads in an image specification file [39], generates the specified volume rendered image, and, then, writes to disk either an image file in SGI *RGBA* format or separate floating point *R*, *G*, *B*, and *A* files. Transfer functions for color and opacity are specified in a piecewise linear method, as in [39]. The radiance integration along a ray may be specified so as to use exact integration [38], which is appropriate when the cells are linear tetrahedra, or five-point Gaussian integration, which is appropriate for quadratic tetrahedra. A faster, but somewhat less accurate, method, which we call the *approximate method*, assumes the opacity varies linearly along the ray segment and assumes the color is constant, equal to the average of the color at the front and the back of the ray segment. This is not exactly correct, since the opacity along the ray segment hides the far color more than the near one, but is much quicker to evaluate.

The data ranges on which the transfer functions are actually linear are separated by data values which we call *breakpoints*. For the exact integration and the approximate method, a cell is sliced into slabs at each transfer function breakpoint that occurs within a cell; in addition, cells are sliced at each user-specified isosurface value. For quadratic tetrahedra, the cell is sliced conceptually at all breakpoints and contour surfaces as a part of the integration procedure. This ensures that the color and extinction coefficient are smooth polynomials within a slab. Within a slab from a linear tetrahedron, we can linearly interpolate either the color and extinction coefficient, or the scalar field. It would not be correct to interpolate the color or extinction coefficient if the cell contained a breakpoint in a transfer function.

Images may be generated in either perspective or orthographic projection, with any specified view transform, and to any resolution. Near and far clipping planes parallel to the screen may be specified, in what we call *z-clipping*, in order to select a volume slab of interest. Any number of

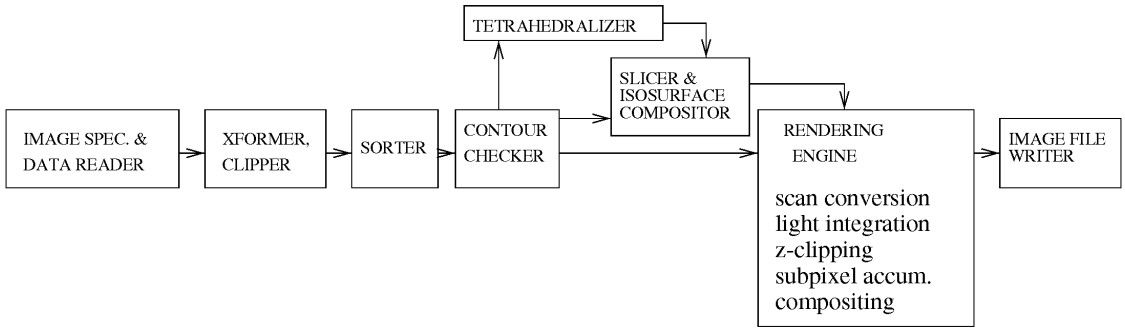


Fig. 2. Schematic diagram of the HIAC volume rendering system. If cells have neither transfer function breakpoints nor contour surfaces in their interior, then the cells go directly from the sorter to the rendering engine. Otherwise, the cells are sliced into slabs, with nontetrahral cells first being partitioned into tetrahedra. Quadratic tetrahedra go directly from the sorter to the rendering engine. as described in Section 5.2.

illuminated Phong-shaded semitransparent colored isosurfaces may be specified for inclusion in the volume rendering. For linear tetrahedra, the contour surfaces are polygonal, and the surface normal for each polygon is used to shade the surfaces. We do not smooth-shade the contour surfaces, because this might obscure important information in the visualization that is useful for determining whether the mesh has been properly refined. For higher- order cells, the isosurfaces are created and shaded on a pixel-by-pixel basis, and not as a set of polygons; therefore, these contours are smoothly curved surfaces within each cell.

Subpixel splatting may be specified, so that contributions from small cells that fall between pixel centers are included in the image. Any background color can be specified, as well as any one of a selection of background patterns. Fast previewing of images is facilitated by hooks to Williams' splatting system [37] based on Shirley and Tuchman's rendering algorithm [29] and Williams' MPVO visibility ordering algorithm [36], and an extension of the techniques of [29] to arbitrary convex polyhedra.

The HIAC system will sort and render all the types of linear cells described in Section 3, as well as quadratic tetrahedra and zoo meshes (meshes which include a combination of the various cell types). Data structures for dealing with mixed cell types and higher-order elements are described in Appendix A.

The HIAC system will also sort and render curvilinear data, provided the faces of the cells are only slightly nonplanar (distorted), as is usually the case in curvilinear grids. In the finite element method, cells can be distorted by a parametric mapping function, in which case the facets of the cells can be highly curved. How to volume render these nonconvex cells with curved facets is an important and interesting open question. We discuss it briefly in Section 9.

For linear cells, if the color or density is linear throughout the cell, i.e., no breakpoints nor isosurfaces occur within the cell, then the cell is rendered as a whole. For nontetrahedral linear cells, including hexahedral cells from a curvilinear grid, if a breakpoint or contour value occurs within a cell, that cell is first tetrahedralized and, then, processed as described above, slicing the resulting tetrahedra as necessary. The system will also deal with quadratic tetrahedra, which it slices conceptually during the integration process whenever a contour lies within the cell.

After the image specification file is parsed, the data set is read in, the view transform and the perspective transform (if applicable) are applied, and the data set is clipped to the view volume. Next, the cells are sorted in visibility order from back to front, sliced (if necessary) into slabs bounded by contour levels and transfer function breakpoints, and, then, the slabs (or cells) are scan converted and the results of the ray integration through the slab (or cell) for each pixel are composited into the image buffers. The image buffers hold the red, green, blue, alpha, and z values in floating-point format. The z buffer is used as a witness to verify the correctness of the visibility ordering. The alpha buffer is used to permit postprocessing accumulation of more than one semitransparent image. A schematic diagram of the system is shown in Fig. 2. The rendering engine does the scan conversion, radiance integration, subpixel splatting, z-clipping, and compositing.

5 The Rendering Engine

Max [17] describes several theoretical optical models for light interacting with a volume density, each with differing degrees of realism. A volume rendering system can be created based on any one of these models. If the system is constructed faithfully according to its model, without the use of approximations, then that system will create accurate images. (If the differential equation for radiance can only be solved by numerical methods, then the system will create images to some predetermined degree of precision.)

A volume rendering system can either integrate the radiance over rays cast out from each pixel through the entire volume density, or project each cell in the volume density onto the screen in visibility order and integrate the radiance over each projected cell for each pixel covered by it. When the cell projection approach is used, a visibility ordering of the cells is required in order to composite the semitransparent volume cells into the image in back-to-front order.

The HIAC volume rendering system, which uses the cell projection approach, is an evolved version of the system reported by Max et al. in [16]. That system used the isotropic *density emitter* optical model of Sabella [27] for the volume effects, and allowed Phong shading of selected contour surfaces—at most, one contour surface could pass through a tetrahedron. We have now improved the slicing algorithm to allow any number of contour surfaces.

In the HIAC system, the *absorption plus emission* optical model [17], [38] is used. In this model, every point in the cloud absorbs light and also emits light (glows). The differential equation for the radiance along a ray towards the eye through the volume is:

$$\frac{dI(t)}{dt} = g(t) - \tau(t)I(t), \tag{4}$$

where t is a length parameter along the ray, and $I(t)$ is the radiance at t. The optical density or extinction coefficient of the volume at t, $\tau(t)$, is considered to be a physical property of each point in the cloud, and defines the rate that light is absorbed or occluded at that point.

The remaining term $g(t)$ is the glow energy emitted at each point of the cloud. There are two ways to treat the glow energy. Wilhelms and Van Gelder [33] treat the glow energy as a physical property of the cloud, whereas Williams and Max [38] consider the glow energy to be defined as $g(t) = \kappa(t)\tau(t)$, where the chromaticity $\kappa(t)$ is considered to be a physical property of each point in the cloud. The HIAC system will generate images using either treatment of the glow energy, as chosen by the user.

We assume the use of piecewise linear transfer functions for specifying the dependence of chromaticity (or glow energy) and optical density on the scalar field being visualized.

By the use of an integrating factor and by applying boundary conditions at t_1 and t_2, we get the following integral equation for the radiance using the Williams and Max treatment of glow energy, see [17], [38]:

$$I(t_2) = I(t_1)e^{-\int_{t_1}^{t_2}\tau(t)dt} + \int_{t_1}^{t_2} e^{-\int_{t}^{t_2}\tau(u)du}\kappa(t)\tau(t)dt\,. \tag{5}$$

This equation is instantiated once for each of the three component wavelengths of light. The second term represents the glow energy along the ray segment, attenuated by the opacity in front of it, and the first term represents the incoming illumination $I(t_1)$ at the far end of the ray, also attenuated by the intervening opacity.

For Wilhelms and Van Gelder's [33] *neon and smog* treatment of glow energy, we get the following integral equation, whose terms can be understood in the same way:

$$I(t_2) = I(t_1)e^{-\int_{t_1}^{t_2}\tau(t)dt} + \int_{t_1}^{t_2} e^{-\int_{t}^{t_2}\tau(u)du}g(t)dt\,. \tag{6}$$

The integral which is the second term on the right side of (5) and (6) cannot be solved in closed form for general $\tau(t)$. However, if the scalar field and the transfer functions vary piecewise linearly along a ray segment within a cell, then the equations can be integrated exactly over each piecewise linear region. This solution is described by Williams and Max in [38] and discussed further in the next section, together with its implementation.

Let the second term of the right-hand side (of either equation) be described as:

$$\int_{t_1}^{t_2} e^{a(t)}c(t)dt\,. \tag{7}$$

A general closed-form solution for this integral is not known when $a(t)$ is cubic or higher order, regardless of the form of $c(t)$. Therefore, even though the $c(t)$ term is lower order in the neon and smog treatment, the integral still cannot be solved exactly when the scalar field is quadratic or higher order (even with linear transfer functions). The neon and smog treatment does, however, permit the glow energy to be mapped independently to a different scalar field than the optical density, which is not possible with the other treatment. Nevertheless, we have created successful visualizations where κ and τ each depended on separate scalar fields using the Williams and Max treatment of glow energy. The main advantage of the Williams and Max treatment is that it makes the specification of the transfer functions somewhat more intuitive. For example, increasing the extinction coefficient makes the surface color more dominant, rather than making the image darker and ultimately black, as is the case with the neon and smog model. More details on this and on the relative merits of the two different treatments of glow energy are given in an Appendix to [39]. The HIAC system allows the optical density to be mapped to a different scalar field than the color, and the contour surfaces can be keyed to a third scalar field.

The HIAC system uses the cell projection approach, therefore, a visibility ordering of the cells is required. The sorting algorithm originally used in [16] was restricted to rectilinear volumes or Delaunay triangulations in 3D. The HIAC system uses a different sorting algorithm, a modified version of the one reported by Stein et al. in [31]. The revised algorithm, which works on an arbitrary collection of acyclic nonintersecting convex polyhedra, is described in Section 6.

The next section describes how the HIAC rendering engine processes linear cells. Section 5.2 describes the treatment of quadratic tetrahedra. Finally, Section 5.3 describes the subpixel splatting procedure.

5.1 Linear Cells

Linear cells, as discussed in Section 3, are convex polyhedra where the scalar field is specified at the conventional vertices and varies linearly along the edges of the cells.

After visibility ordering the cells, each cell is checked to determine if the range of the scalar field within it includes any transfer function breakpoint values. If any are found, and the cell is tetrahedral, the cell is sliced at each breakpoint, resulting in *slabs* in which the color and opacity are linear. Each slice is defined by a contour surface for the field value corresponding to a transfer function breakpoint. Since the scalar field varies linearly within a tetrahedron, the slices are planar and parallel. An example slab is shown in Fig. 3. If an isosurface is to be separately rendered, the tetrahedron must also be sliced at these contour values so the slabs and surface polygons can be composited individually in the correct order. Currently, if the cell is nontetrahedral, we first subdivide the cell into tetrahedra and then slice the tetrahedra into slabs.

The visibility ordering of the tetrahedra within a hexahedron, or the slabs within a tetrahedron, is simple, and is done separately from the global visibility ordering of all the cells. (Methods like the marching cubes algorithm of Lorensen and Cline [11] can slice hexahedral cells directly, but the slices, which are curved surfaces, must be divided into triangles. If there are multiple contour levels inside a single

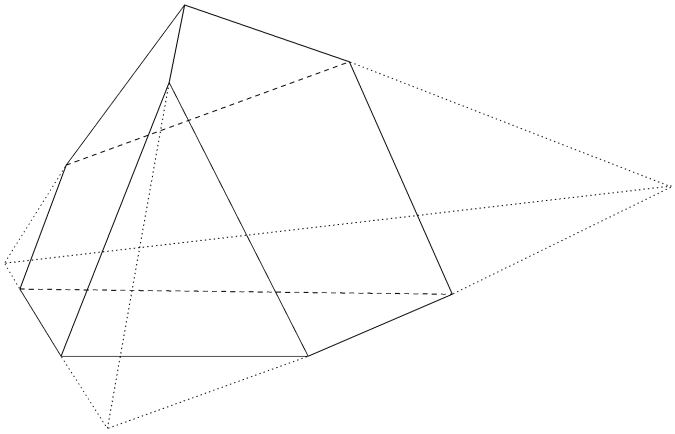


Fig. 3. Example of slab of linear tetrahedron. Within the slab, the color and opacity are linear. The slicing planes defining the slab are contour surfaces for field values corresponding to transfer function breakpoints or for user-specified isosurface values.

cell, it is not clear that they will be nonintersecting, or that the volume pieces they slice off will be convex, as required by our projection algorithm.) We do not subdivide nontetrahedral cells into tetrahedra when the range of scalar function within the cell does not include a slicing value. Both the sorter and the scan converter will handle the different types of linear cells described in Section 1. The rendering engine will correctly process convex cells or slabs with any number of faces and vertices.

The slicing algorithm is robust, permitting up to three vertices of a tetrahedron to take on the same slicing value. When all four vertices have the same contour value, the set of points taking on the contour value is no longer a surface, but contains the whole tetrahedron. In that case, the contribution of such a tetrahedron to the isosurface is neglected, leaving a hole in the contour surface, but the tetrahedron will still contribute to the volume rendering. When a contour surface intersects three vertices of a tetrahedron, the surface corresponds to one of the cell's faces, which may be shared with another cell. In that case, it is important to render the contour polygon only once. The boundary polygons for the slabs are found from a case-by-case analysis of the ways a slice plane can intersect a tetrahedron, and the ways in which the slab between two consecutive slice planes can intersect the tetrahedron's face triangles.

For orthogonal projection, the back-to-front sorting order of the slabs within a cell can be determined from the z-component of the gradient of the scalar field S on the cell: $S(x, y, z) = c_1 + c_2x + c_3y + c_4z$. (Since the field is known at the four vertices of the cell, the four constants can be determined for the cell.) If the z-component of the gradient is greater than zero, then the back-to-front order starts with the slab having the largest scalar value.

For perspective projection, consider the slicing planes defining the slabs to be infinite parallel planes in world coordinates. If the viewpoint lies between two of these infinite planes which define a slab, we call that slab the *eye slab*. The slabs and contour surfaces are then composited in two groups: from one side up to, but not including, the eye slab and, then, from the other side, up to and including the eye slab. If there is no eye slab, only one group is required, as in the parallel projection case.

Next, the cells or slabs are sent in visibility order to the rendering engine for projection and accumulation. The first step in this process is to scan convert the cell or slab. The front-facing polygons bounding the cell are scan converted into a front z-buffer, with values z_f, and the back-facing polygons into a back z-buffer, with values z_b. The κ and τ values are bilinearly interpolated along edges and across scan lines, as in Gouraud shading, and saved in the front or back buffers, as κ_f and τ_f, or κ_b and τ_b, respectively. Then, for each pixel in the projection of the cell, the length l of the ray segment is computed as $z_b - z_f$, and the values of κ and τ are assumed to vary linearly between their values in the front and back buffers.

In parallel projection, this results in piecewise trilinear interpolation, where the subdivision into trilinear pieces depends on the projection of the polyhedron. Thus, as in piecewise bilinear Gouraud shading, the interpolation scheme is not rotationally invariant. However, for linear tetrahedra, or for slabs cut from them on which κ and τ are linear, this trilinear interpolation reduces to linear interpolation, which is rotationally invariant.

Next, the ray integration is performed for each pixel covered by the cell or slab. When the cells are linear tetrahedra and the transfer functions are piecewise linear, the integral in (5) can be integrated exactly as shown in [38] by completing the square of the exponent and repeated application of integration by parts, yielding:

$$I(t_2) = \frac{\alpha + \beta t_2}{\delta^2} - \frac{\mu + \nu t_1}{\delta^2} e^{(t_1 - t_2)(2\gamma + \delta t_1 + \delta t_2)/2} + \eta \delta^{-2.5} e^{-(\gamma + \delta t_2)/2\delta} \left(erfi\left(\frac{\gamma + \delta t_2}{\sqrt{2\delta}} \right) - erfi\left(\frac{\gamma + \delta t_1}{\sqrt{2\delta}} \right) \right) + I(t_1) e^{-\left(\gamma t_2 + \frac{\delta t_2^2}{2} - \gamma t_1 - \frac{\delta t_1^2}{2} \right)}, \quad (8)$$

where, α, β, δ, μ, ν, γ, and η are functions of the four constants c_i in the tetrahedral interpolation function $S(x, y, z) = c_1 + c_2x + c_3y + c_4z$, the constants describing the applicable linear pieces of the transfer functions, for example, $\tau(x, y, z) = a + bS(x, y, z)$, and the three ray parameterization functions, $x = u_1 + u_2t$, etc., as described in [38]. The *erfi*() function will be discussed below. Key terms in (8) are: δ, since it appears in the denominator of several terms, and $\gamma + \delta t$, the numerator in the argument to *erfi*(). The term δ is equal to the slope of the pertinent piece of the optical density transfer function, i.e., b in the example above, times the slope of the (linear) scalar field within the cell. The term $\gamma + \delta t$ is exactly $\tau(t)$, as can be seen from the detailed derivation given in [38].

The *complex error function*, *erf*(), is defined as:

$$erf(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-u^2} du.$$

The *imaginary error function*, *erfi*(), which appears in (8), is defined as $erfi(z) = erf(iz)/i$. In (8), *erfi*'s argument is either real when $\delta > 0$, or pure imaginary when $\delta < 0$. When its argument is real,

$$erfi(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{u^2} du.$$

When its argument is pure imaginary, of the form $z = ib$, with real b,

$$erfi(ib) = i \frac{2}{\sqrt{\pi}} \int_0^b e^{-u^2} du.$$

When $\delta < 0$, the i in the latter expression cancels the i in the factor $\delta^{-2.5}$ in (8). When $\delta = 0$, the solution to the integral takes a different and relatively simple form, as shown in [38], involving only the exponential function, and not *erfi*(). (The formula for this case in [38] can be further simplified by noting that, when $\delta = q_2 = 0$, q_5 is also zero, eliminating half the terms.)

Originally, in [38], we implemented the *erfi*() functions in terms of the indefinite integrals $\int_0^x e^{-t^2} dt$ and $\int_0^x e^{t^2} dt$. We precomputed these integrals incrementally at equally spaced values of x using Simpson's rule, and stored the results in two tables. To evaluate (8), we interpolated values from these precomputed tables.

By introducing Dawson's integral, which is defined in [24] as $D(x) = e^{-x^2} \int_0^x e^{t^2} dt$, the solution to the integral equation can be simplified to eliminate the use of tables. Dawson's integral is related to the complex error function by:

$$D(x) = \frac{-i\sqrt{\pi}}{2} e^{-x^2} erf(ix) = \frac{\sqrt{\pi}}{2} e^{-x^2} erfi(x). \quad (9)$$

An efficient and accurate numerical approximation for Dawson's integral, due to Rybicki [26] and described in [24], enables the calculation of *erfi*() without the use of tables. The accuracy of Rybicki's approximation increases *exponentially* as the step size, h, used in the approximation, gets small. We use $h = 0.4$, which gives an accuracy of about 2×10^{-7}. The function $erfi(x)$ for imaginary x can be reduced, by a trivial change of variables, to the Error integral, the integral of a Gaussian normal distribution, for which subroutines also exist, as described in [24]. Appendix B discusses details of implementation and how to avoid overflow in the exponentials.

For linear tetrahedra, the HIAC system uses the exact solution from [38], utilizing subroutines for Dawson's integral and the Error integral as described above.

When Wilhelm and Van Gelder's neon and smog treatment of the glow energy is used, the techniques given above still apply, but the term $\kappa(t)\tau(t)$ in the integrand is replaced by $g(t)$, which is now linear, rather than quadratic, for linear scalar data. Unfortunately, this does not permit any significant further simplification in the calculus.

When a perspective view is specified, a bit of care is required to do mathematically correct interpolation and integration, since the distance metric along an edge or ray is distorted by the perspective transform. After performing the perspective transform, the scalar field no longer varies linearly along the edges of a cell nor on a ray through a cell. (For example, the midpoint of an edge in parallel projection is no longer the midpoint of that edge after the perspective transform.) Therefore, integration techniques that are suitable for linear functions no longer pertain. Our approach to this problem is to reverse the perspective transform and do the interpolation and integration in world coordinates rather than screen coordinates. When this is done, the length of the ray segment must be computed as a 3D (slanted) distance, rather than just a difference in z values. (For a perspective ray from the origin through a pixel at $(\bar{x}, \bar{y}, 1)$, the difference of the world coordinate z-values of the endpoints of the ray segment must be multiplied by $\sqrt{\bar{x}^2 + \bar{y}^2 + 1}$.) The details of this work are tedious and the interested reader is referred to our code, which is in the public domain as indicated in Section 9.

Near and far clipping planes parallel to the screen may be specified to achieve a volume slab of interest. This z-clipping is accomplished as follows. Cells entirely in front of the near clipping plane are skipped, as are cells entirely behind the far clipping plane. Cells intersecting the slab are processed normally, but, for each pixel, the viewing-ray/cell intersection segment is restricted to the region inside the slab. This could be done efficiently by 3D polyhedron clipping, but the above per-pixel *scissoring* alternative was easier to code. z-clipping is also implemented for quadratic cells.

5.2 Quadratic Cells

As discussed in Section 3, quadratic cells are cells where the scalar field varies quadratically along the edges of the cell. In this section, we deal with quadratic tetrahedra, which have six interior nodes, one per edge, as in Fig. 1. Other higher-order cells, as well as linear bricks and prisms, are discussed in Appendix C.

In a quadratic tetrahedron, the scalar field varies quadratically along any ray segment through the cell because (3) contains only quadratic terms. Inside these cells, contour surfaces will be curved, therefore, the slabs will be curved, and a viewing ray may intersect a single slab twice. Because of this, we do not actually partition the cells into slabs as we did for linear tetrahedra, but, rather, process each ray through a cell in segments. In each ray segment, the color and optical density vary smoothly.

The interpolation function for a quadratic tetrahedron has the form of (3), which we repeat below:

$$f(x, y, z) = c_1 + c_2x + c_3y + c_4z + c_5x^2 + c_6xy + c_7y^2 + c_8yz + c_9z^2 + c_{10}xz.$$

Substituting the coordinates of the 10 nodes, along with their field values, into this equation, we get 10 equations for the 10 unknown polynomial coefficients c_i, which we solve with the LINPACK linear algebra package.

The segment endpoints are found as follows: The ray equations parametrized by t, are

$$x(t) = \bar{x}(t_0 - t),\ y(t) = \bar{y}(t_0 - t),\ z(t) = \bar{z}(t_0 - t).$$

(We assume the viewpoint is at the origin, the pixel is located at $(\bar{x}, \bar{y}, \bar{z})$, where $\bar{z}$ is the distance from the viewpoint to the screen, and the ray is in the direction of light flow, with t increasing toward the eye.) Substituting the ray equations into the quadratic interpolating function shown in (3), gives a quadratic polynomial in one variable: $f(t) = at^2 + bt + c$. When a is nonzero, this polynomial will take on a

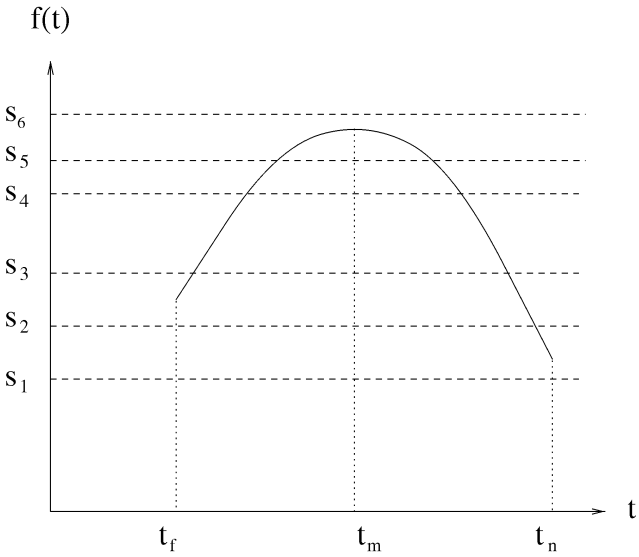


Fig. 4. The variation of the scalar field $f(t) = at^2 + bt + c$ along a ray, toward the eye, through a quadratic tetrahedron as a function of the ray parameter t. Example regions defined by contour surfaces corresponding to transfer function breakpoints are also shown. The horizontal regions between s_i and s_{i+1}, for $1 \le i \le 5$ correspond to curved slabs in the tetrahedron. The ray enters the tetrahedron at t_f.

maximum or minimum at a single t value $t_m = -b/(2a)$. If the ray segment does not contain t_m, the quadratic polynomial is monotonic in the segment. Otherwise, it contains both increasing and decreasing regions, as shown in Fig. 4. These regions, and their monotonic direction, can be determined from the sign of a, which is the sign of the second derivative of f, and the location of t_m relative to t_n and t_f, the near and far endpoints of the ray segment.

For every relevant slicing value, s_i, the corresponding breakpoints in t can be found by using the quadratic formula for the roots of $f(t) = s_i$. Between every pair of consecutive breakpoints s_i and s_{i+1}, the color and optical density will each be represented by a smoothly varying polynomial in t. The points in the quadratic tetrahedron, which have scalar values in the interval $[s_i, s_{i+1}]$, define a curved slab bounded by the contour surfaces at the breakpoints and parts of the tetrahedron's surface facets.

We will refer to this as the slab $[s_i, s_{i+1}]$. The ray/slab intersection segments and their order along the ray can be determined by comparing $f(t_n)$, $f(t_f)$, and $f(t_m)$ with the slicing values s_i. In the case illustrated in Fig. 4, $s_1 < f(t_n) < s_2 < f(t_f) < s_3 < s_4 < s_5 < f(t_m) < s_6$. Therefore, the ray enters the slab $[s_2, s_3]$ through a tetrahedron face at the left where f is increasing, passes through the slab $[s_3, s_4]$, continues through slab $[s_4, s_5]$, then enters the slab $[s_5, s_6]$, continues through $[s_5, s_4]$ (with f decreasing), and, then, slabs $[s_4, s_3]$, $[s_3, s_2]$, and $[s_1, s_2]$, and, finally, exits through a face of the tetrahedron. Though t_m is shown in Fig. 4, it is not one of the breakpoints; f reaches its maximum on the ray segment inside the slab $[s_5, s_6]$, but continues smoothly past its maximum, as do $g(t)$ or $\kappa(t)$, and $\tau(t)$, so there is no need to subdivide the integration there. Other cases can be handled similarly: The code has two loops over the increasing and the decreasing ranges of $f(t)$, but one may not be needed.

Assuming the Williams and Max treatment of glow energy, let $\kappa(t)$ and $\tau(t)$ be the chromaticity and optical density, respectively, at position t along the ray. Then, the total opacity from a ray segment $[p, q]$ is:

$$e^{-\int_p^q \tau(t)ds} \tag{10}$$

and the total radiance or color added by that segment is:

$$\int_p^q e^{-\int_t^q \tau(u)du} \kappa(t)\tau(t)dt. \tag{11}$$

For $\kappa(t)$ and $\tau(t)$ quadratic in t, (10) can be integrated exactly, but numerical integration is required for (11) because the $a(t)$ in (7) is a cubic polynomial. We have used five-point Gaussian integration [24], which gives exact answers for polynomials of up to degree nine, and very good approximations for sufficiently smooth functions that are well approximated by such polynomials, but poor approximations for functions which are not smooth. This is the reason for breaking the range of integration up into the subsegments where $\kappa(t)$ and $\tau(t)$ are smooth polynomials. The color (11) and opacity (10) on these subsegments are composited in the back-to-front order described above.

If a semitransparent contour surface is requested at the near breakpoint of a subsegment, it is composited after the subsegment. The surface normal is computed from the partial derivatives of (3), and used for Phong shading, as well as to make the surface appear more opaque when it is seen edge-on, as if it were a finite thickness of partially absorbing glass. This makes the contour surfaces appear appropriately curved within a cell, even in the absence of reflected light. However, finite element simulations rarely produce results which are C^1 across cell boundaries, so the contour surfaces may not be globally smooth.

5.3 Subpixel Splatting

In curvilinear or irregular meshes designed to concentrate small cells near shocks, boundaries, or other regions of rapid change or special interest, projections of tiny but important cells may fall between the pixel centers. This can also happen due to perspective foreshortening. Any volume rendering algorithm which samples the image only at pixel centers may, therefore, miss significant details entirely, or include them with an inappropriate weighting. This is the case in both ray tracing and cell projection methods.

The theoretically correct solution to this problem is to determine an analytic representation for the image as a function of the continuous coordinates on the image plane and, then, convolve it with a presampling filter kernel, before sampling it at the pixel centers. Because of the geometric and analytic complexity of a volume rendered image, this is a formidable task.

We use an approximation to this analytic antialiasing, suggested by Westover's *splatting* technique [32]. If a cell's projection overlaps too few pixels (for example, less than two), we assume that its color and opacity effects on the image are concentrated at its center of gravity. We, therefore, take a delta function at the projection of the cell's center of gravity, and multiply it by the volume of the cell, the perspective projection area shrinkage factor, and the color and opacity at the cell's center of gravity. We then convolve

this weighted delta function with a presampling filter kernel (described below), which is equivalent to taking a weighted translated copy of the kernel. The result is a splat to be composited onto the image.

If subpixel splatting is turned on, when the rendering engine gets a cell, we first do a rudimentary scan conversion to determine the number of pixels covered. If the pixel count is larger than a threshold, we repeat the scan conversion, doing the analytic integration for color and opacity, and composite the result into the image. Otherwise, we composite a weighted translated copy of the filter kernel.

We use a piecewise biquadratic kernel, the product of two identical 1D piecewise quadratic kernels in x and y, the B-spline kernel. This kernel is the twice iterated convolution of a pixel-sized box filter with itself. In spatial frequency, this filter has the Fourier transform $sin^3(\pi x)/(\pi x)^3$, which greatly attenuates frequencies greater than the Nyquist limit and, so, gives good antialiasing. However, it does cause some minor blurring, since frequencies less than the Nyquist limit are also attenuated, and the footprint of each cell is a 3×3 square of pixels. A wider filter, such as the one we use, is superior to a pixel-sized box filter kernel when bright objects much smaller than a pixel move during animation. With a box filter kernel (area averaging), the bright object would suddenly jump from one pixel to an adjacent one when it crossed the edge between them, but, with a wider kernel, the contributions smoothly fade up and down.

Rather than precompute and store a high resolution version of this splat, as Westover did, we just evaluate the simple quadratic polynomials each time they are needed. The polynomial variables are the fractional subpixel coordinates of the projection of the cell's center of gravity. The original algorithm of Westover used splats whose footprint decreased as the projected splats got closer together, but this method could also cause splats to be lost between pixels! Our solution is to keep the splat size to a three-pixel square, and decrease the color/opacity amplitude instead, as described above. Another approach (for regular grids only) is given by Mueller and Yagel in [18]. They use summed area tables to compute the integral of the splat footprint over the pixel area, so all splats will contribute their effects completely to the image.

We tested subpixel splatting by dividing a cube into a large number of tiny tetrahedra, each smaller than a pixel, and compositing their splats. The result was the same as the analytic integration over the projection of the five larger tetrahedra representing the original cube, except for slight blurring.

This splatting scheme is not a perfect solution to the antialiasing problem. Suppose a tiny cell is very bright, but it is totally occluded by another tiny cell directly in front of it which happens to be dark and very opaque. First, the tiny bright cell contributes a proportion to a nearby pixel, then the totally opaque cell contributes a proportion to the opacity, but, overall, the pixel will incorrectly retain some brightness. Variants of this problem with per-pixel compositing occur with any scheme that does not represent the complete geometric projection of all cells overlapping the filter kernel. We are currently working on an analytic antialiasing scheme which does take into account the complete geometry, but we expect it to be very slow.

6 The Visibility Ordering Algorithm

The HIAC volume rendering system uses the cell projection method which requires a visibility ordering of the cells. We use the accurate sorting algorithm presented by Stein et al. [31], which is an $O(n^2)$ (worst case) method for visibility ordering n arbitrary shaped, nonintersecting convex polyhedra with planar faces, whose visibility ordering does not contain cycles. The faces of adjacent cells need not be aligned, and the meshes may have disconnected portions. The algorithm is effectively a 3D generalization of the Newell et al. sort for polygons [3], [19], [20]. A z-buffer is incorporated in the rendering engine to serve as a witness to the correctness of the visibility ordering.

A correction to the original algorithm reported by Stein et al. is given in Section 6.1; then, in Section 6.2, we describe a method that, for large data sets, increases the efficiency of the algorithm by up to two orders of magnitude. The original Stein visibility ordering algorithm, which outputs the cells in back-to-front order, can be quickly described in the following steps:

First, transform all of the vertices to screen coordinates with a perspective corrected z. Next, create a roughly sorted list of the polyhedra by arranging the elements in back-to-front order based on each polyhedron's rearmost z coordinate. The algorithm *QuickSort* works well here. Last, fine-tune the sort by performing visibility tests for each relevant pair of polyhedra in the list. This fine tuning is described in more detail in the following paragraph.

The goal of the fine-tuning stage is to verify that no polyhedron P obscures any other polyhedron following it in the list. If P does not obscure any polyhedron following it in the list, then P can be safely output. However, if P does obscure some element later in the list, then a portion of the list must be rearranged.

We determine that P does not obscure an element Q by finding whether P lies behind a plane that separates the two elements. Because this can be difficult and time-consuming, the algorithm has a predetermined list of planes which it tries. This list starts with the planes that are easiest to calculate, such as the planes perpendicular to the X, Y, and Z axes, and ends with the more computationally difficult possibilities, such as the planes defined by the front- and back-facing faces of P and Q. If a separating plane cannot be found from this list, then the explicit screen-projections of the two polyhedra are examined by a subroutine *ProjectsBehind*(P, Q) to determine whether P obscures any part of Q.

The fine-tuning is described as follows: Given a roughly sorted list of elements in back-to-front order, P, the element at the head of the list, can be output (i.e., it obstructs no remaining cells) if, for all elements Q whose z-extent overlaps P's z-extent, the following subroutine *Obstructs*(P, Q) returns false:

Obstructs(P, Q)
if (P and Q have no overlapping X extents) return *False*
else if (P and Q have no overlapping Y extents) return *False*

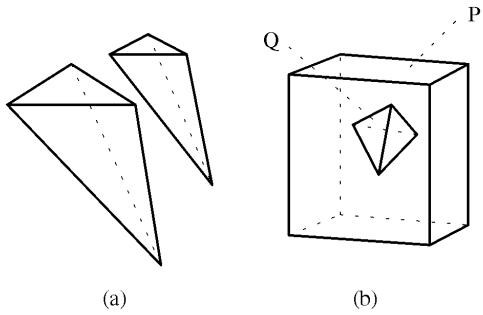


Fig. 5. Two cases when *ProjectsBehind*(*P*, *Q*) cannot find intersections in the screen projection of the edges of cells *P* and *Q*.

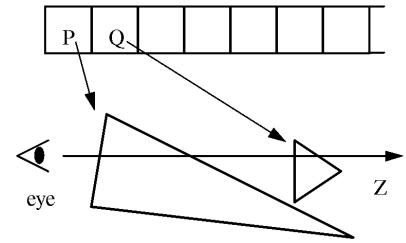


Fig. 6. Example of configuration (b).

else if (*P* is behind a back-face of *Q*) return *False*
else if (*Q* is in front of a front-face of *P*) return *False*
else if *ProjectsBehind*(*P*, *Q*) return *False*
else return *True*

If *Obstructs*(*P*, *Q*) is false for all *Q*, then *P* can be safely rendered. In this case, *P* is output, removed from the list, and the next element at the start of the list becomes the new *P*; the process repeats until the list is empty. If *Obstructs*(*P*, *Q*) is true for a pair of polyhedra *P* and *Q*, then *P* obstructs at least a portion of *Q*. In this case, *Q* is moved to the head of the list, thereby becoming the new *P*, and the process repeats for this new *P*. When a *Q* is moved to the head of the list, it is tagged as having been moved. If such a *Q* requiring moving has already been tagged, then the visibility ordering of the data set contains a cycle, and this is reported. We have not yet implemented the polyhedron subdivision necessary for breaking cycles.

ProjectsBehind(*P*, *Q*) examines the screen projections of the two polyhedra to determine whether *P* obscures any portion of *Q*. This subroutine searches for edge intersections between the screen projections of all edges of cells *P* and *Q*. If an edge intersection is found, the *z* components of the intersection point on *P*'s and *Q*'s actual edges are compared, enabling *ProjectsBehind*(*P*, *Q*) to return the appropriate value. If *P*'s *z* component of the intersection point lies behind *Q*'s *z* component of the intersection point, then *P* lies behind *Q*. Otherwise, *P* obscures at least a portion of *Q*. There are two cases, however, when *ProjectsBehind*(*P*, *Q*) cannot find edge intersections in the screen projection, as illustrated in Fig. 5. The original algorithm published by Stein et al. [31] did not deal with this situation correctly. Both the process of handling these two configurations and a correction to the original algorithm are given in the next subsection.

In the event that a facet of a cell is nonplanar, we approximate a plane equation to the vertices defining the facet, using Newell's method [25]. Then, for each vertex (x, y, z) of the facet, we calculate its deviation, $|Ax + By + Cz + D|$, where A, B, C, D are the plane equation coefficients with an outward-pointing normal, and retain the maximum deviation along with A, B, C, D for each facet. To determine if a facet of cell *P* defines a separating plane with regard to cell *Q*, we substitute the coordinates of each vertex of cell *Q* into the plane equation for the relevant facet of cell *P*. If the plane equation evaluates to a positive value for all vertices, then that facet is a separating plane. To deal with slightly nonplanar facets when evaluating the plane equation, the result plus the deviation must be greater than zero.

6.1 Correction to the Sorting Algorithm

In configuration (a) of Fig. 5, because the polyhedra are entirely disjoint, either one can be output without affecting the visibility of the other. In configuration (b) of Fig. 5, that is not true; element *Q* lies behind element *P*. The reason why *Q* lies behind *P* is as follows: Since *ProjectsBehind*(*P*, *Q*) has found no edge intersections in the screen projections of *P* and *Q*, and we are assuming configuration (b) holds, then either a front face or a back face of *P* defines a separating plane between *P* and *Q*. But, the fourth test of *Obstructs*(*P*, *Q*) has ruled out the front face of *P* as a separating plane; therefore, *Q* must be behind *P*.

This issue was not addressed by Stein et al. in [31] because they incorrectly stated that configuration (b) could not occur. We now describe how such a configuration could occur. Suppose we have two polyhedra at the beginning of our roughly sorted list (a back-to-front sorting based on the rearmost *z* component of each element), as illustrated in Fig. 6. Assuming that the screen projections of the edges of *P* and *Q* do not intersect and that their *x* and *y* extents are not disjoint, then we have an instance of either configuration (a) or (b) in Fig. 5 where the first four tests of *Obstructs*(*P*, *Q*) fail. The problem, then, is to distinguish which of the two configurations exists.

To do this, we test whether *P* and *Q* are entirely disjoint from each other by deciding whether the projection of *P* is contained in the convex hull [28] of the projection of *Q*, or whether the projection of *Q* is contained in the convex hull of the projection of *P*. If both of those tests fail, indicating configuration (a), then *P* and *Q* are entirely disjoint from each other and *ProjectsBehind*(*P*, *Q*) returns *true* indicating that *P* can be output. Otherwise, if either of the convex-hull tests is true, then *ProjectsBehind*(*P*, *Q*) returns *false*, indicating we have configuration (b) and that the order must be changed.

6.2 The Multitiled Sort

The sorting algorithm described above is $O(n^2)$ worst case, for *n* cells, because the first *z*-range overlap test may need to be performed for every pair of cells. Since, in general, a very large percentage of cells do not overlap each other in *x* and *y*, we reasoned that, by tiling the view-plane window into a rectilinear grid of *p* tiles, and, then, for each tile, sorting

TABLE 1
RESULTS OF PRELIMINARY EXPERIMENT TO TEST EFFECT OF TILING ON THE SORTING ALGORITHM

	Time in minutes to sort *n* cells using tiling and a single CPU			
Tiles	13,000	190,000	600,000	1,000,000
1	0.23	48.0	490	909
2	0.17	19.1	245	403
4	0.15	9.2	87	161
8	0.14	6.1	41	62
16	0.14	4.3	19	33
32	0.15	3.4	11	19

Shown are times in minutes to visibility order n *cells by calling the original sorting algorithm* t *times, each with one of* t *tiles, using a single R10000 CPU of an SGI Power Onyx.*

only the cells that project onto it, we could gain a speed-up of up to p times (since each tile would have approximately n/p cells). Of course, many cells may overlap tile boundaries and, typically, the algorithm does not require worst-case quadratic time. But, the sort was taking so long for large data sets that it was worth experimenting to see what improvement could be achieved. We partitioned the screen into p roughly load balanced tiles and, then, sorted the cells in each tile sequentially, using a single CPU. The end result was p sorted lists of cells, one for each tile, which could then be rendered on a tile-by-tile basis. The results of this experiment were very gratifying; the times reported in Table 1 are for an SGI Power Onyx system using a single R10000 processor.

Using the method described above, if a cell projects onto more than one tile, that cell will appear in more than one sorted list, thus requiring multiple clipping and/or scan conversion passes. Therefore, we modified the sorting algorithm to utilize tiling internally. The internal tiling process we describe next produces a single sorted list of all the cells, for the entire view plane window.

In the multitiled sort, in order to decide whether polyhedron P can be safely output, we have to verify that P lies behind all polyhedra Q that overlap the same tiles as P. The multitiled visibility sorting algorithm can be described as follows:

1) Sort the elements in back-to-front order as before using the elements' rearmost z coordinate as the sorting criterion. We call this (roughly) sorted list the *Global Sorting List* (GSL).
2) Assign each element a unique identity (an element's initial position in the GSL will suffice), and give each element a *last_comparison* variable to keep track of the last polyhedron to which it was tested and found to lie in front. In other words, if element T's *last_comparison* holds element U's identity, then T has been determined to lie in front of element U.
3) Divide the view-plane window into disjoint rectangular tiles. By default, we assume a uniform distribution of cells in the volume, and partition the window into $n^{1/6}$ by $n^{1/6}$ equal sized tiles, where n is the total number of cells. While $n^{1/3}$ by $n^{1/3}$ equal sized tiles would *ideally* result in the fewest number of cells per tile, $n^{1/3}$, large values of n would lead to a large number of tiles and, in turn, a large amount of storage. Thus, the somewhat arbitrary $\frac{1}{6}$ exponent was chosen to keep the number of windows down to roughly $n^{1/3}$. The sort can optionally be called with an array of alternative tile dimensions to allow the use of load balanced tiles. The HIAC system uses a load-balancing scheme designed for unstructured meshes which will be described in a subsequent publication.
4) For each tile, create a *Tile Sorting List* (TSL) which is a linked list of pointers to the polyhedra overlapping the given tile. In the TSL, the polyhedra are sorted in back-to-front order based on their rearmost z coordinate. For each polyhedron, create a list of pointers to tiles that the polyhedron overlaps.

The following steps are essentially a merge of the separate TSLs.

5) Begin the merge by selecting the head polyhedron in the GSL and calling it P. For each tile that P belongs to, determine whether P lies behind all of the polyhedra Q that occupy P's tiles and whose rearmost z coordinate lies behind P's frontmost z coordinate, by using the subroutine *Obstructs*(P, Q). However, before actually testing each Q, examine Q's *last_comparison* variable to see whether Q has already been tested with element P in a different tile (each element can only occur once in any TSL). If the variable contains P's identity, then P has already been found to lie behind Q and the tests can be skipped. Proceed to the next element following Q in the TSL. Otherwise, determine whether P lies behind Q by calling *Obstructs*(P, Q). Record P's identity in Q if P can be safely output before Q and, then, proceed to the next element following Q in the TSL.
6) When an element P fails a test against a particular Q, tag Q and move it to the head of the GSL (Q does not move in any of the TSLs and, therefore, does not affect the early termination condition mentioned in the following step. If Q has already been tagged as moved, then the data set contains a cycle.) This Q now becomes the new P, and the whole process is repeated.
7) Once we find an element Q whose rearmost z coordinate lies in front of P's frontmost z coordinate, we may terminate any further tests in that particular TSL because the remaining elements in the TSL lie fully in front of P. Proceed to the next TSL that P occupies and repeat the tests for all the applicable Q.
8) When an element P passes the tests for all of the applicable Q in each of the TSLs it occupies, P is output and removed from the GSL as well as from all of the TSLs to which it belongs. The next element at the head of the GSL becomes the new P and the whole process repeats.

Comparative timings for the above multitiled sort versus the original sorting algorithm reported by Stein et al. [31] (with the corrected algorithm) are given in Table 2. Load balanced tile dimensions were provided to the sorter for

TABLE 2
COMPARATIVE TIMINGS IN MINUTES TO VISIBILITY ORDER n CELLS USING THE ORIGINAL SORT AS REPORTED BY STEIN ET AL. IN [31] VERSUS THE REVISED ALGORITHM USING TILES DESCRIBED HEREIN, USING A SINGLE R10000 CPU OF AN SGI POWER ONYX

No. Cells	Original Sort	Multitiled Sort
13,000	0.23 min.	0.12 min.
190,000	48.0 min.	2.7 min.
600,000	489.5 min.	9.5 min.
1,000,000	908.6 min.	15.0 min.

these tests. For 1,000,000 cells, the tiling of the sort resulted in a 60 fold speed-up in sorting time.

7 HARDWARE BASED POLYHEDRON PROJECTION

The volume rendering system described to this point is not interactive, because it uses a precisely correct sorting, and a slow, accurate, analytic, or numerical integration along each ray segment on which the transfer functions are linear. For rapid preview, polygon-based rendering hardware can be used instead. Shirley and Tuchman [29] divided the projection of a tetrahedron into from one to four triangles, and used hardware scan conversion, transparency, and back-to-front compositing to produce an image. Stein et al. [31] point out that the linear transparency interpolation between the triangle vertices replaces what should be an exponential computation per pixel in (5), and can produce Mach bands. They suggest a more accurate method using hardware texture mapping, which we have now generalized from tetrahedra to arbitrary convex polyhedra.

The bilinear interpolation of z, τ, and κ across faces, described in Section 5.1, could be performed in a standard rendering pipeline, but standard hardware does not permit the interpolated values to be stored in separate front and back buffers and combined later. Therefore, we need smaller *homogeneous regions* where a single value for each of these parameters will suffice and can be interpolated linearly or bilinearly.

Consider the set S of polygonal regions into which the projection of a convex polyhedral cell P is divided by the projections of all its edges. Since no projected edges of P cross the interior of any polygonal element R of S, R lies within the projection of a single front facing facet of P, and of a single back facing facet of P. Since each facet is planar, the thickness $l = z_b - z_f$ of R varies linearly over R, and can be bilinearly interpolated in hardware from its values at the vertices of R. Since τ_f and τ_b are linear (or bilinear) over R, their average $\tau_{avg} = (\tau_f + \tau_b)/2$ is also, and it too can be interpolated in hardware.

Consider a ray segment s of length $l = t_2 - t_1 = z_b - z_f$, on which τ varies linearly between τ_b at t_1 and τ_f at t_2. Then, the transparency $T = e^{-\int_{t_1}^{t_2} \tau(t)dt}$ on s reduces to $T = e^{-l\tau_{avg}}$. The standard compositing hardware produces a linear combination:

$$I_{new} = (1 - \alpha)I_{old} + \alpha I_{add} \tag{12}$$

where I_{old} is the current color in the frame buffer, I_{add} is the interpolated color for the current polygon, and I_{new} is the new color to be placed in the frame buffer. The opacity $\alpha = 1 - T$ can be interpolated from its vertex values, as in [29] or [33], or taken from a texture map, as in [31]. The 2D texture map $M(u, v)$ is preloaded with the function $1 - e^{-uv}$, and the texture parameters are set at the vertices of R with $u = l = z_b - z_f$, the length of the ray segment s, and $v = \tau_{avg}$. When u and v are bilinearly interpolated by the hardware, and $M(u, v)$ is used for α, the hardware finds, per pixel, the exponential needed for correct transparency. The access to the SGI graphics hardware pipeline is through OpenGL, so the code should be fairly portable.

The chromaticity κ is bilinearly interpolated by the shading hardware, and used as I_{add} in the compositing equation (12). We have three methods for calculating κ at the vertices of R. They are, in increasing order of accuracy and computation time:

- (M1) the average chromaticity $(\kappa_f + \kappa_b)/2$,
- (M2) the table-based evaluation of (8) described in [38], and
- (M3) the subroutine-based evaluation of (8) described in Section 5.1 and Appendix B.

In methods (M2) and (M3), we divide the radiance from (8) by the opacity α, to get an effective chromaticity at the vertex. When this quantity is interpolated, and used as I_{add} in the compositing equation (12), the result gives some of the effects one would expect, such as a closer color κ_f partially obscuring a farther color κ_b along the same ray, although it is not as accurate as evaluating (8) at each pixel. We also offer a method (M0) which just uses the hardware bilinear interpolation of α from its values at the vertices of R, instead of the texture mapping. It is even faster and less accurate than method (M1).

Method (M0) is a direct generalization of the method of [29] and may be used if texture mapping hardware is not available. If the chromaticity κ is constant on a ray segment, the integral in (7) reduces to $(1 - T)\kappa = \alpha\kappa$, as explained in [17], so method (M1) is appropriate for cells of constant chromaticity. Method (M2) may be sufficiently accurate for 8-bit-per-color images, if the precomputed tables for the integrals of e^{t^2} and e^{-t^2} are large enough in range and resolution. However, since each integral is looked up individually, the separate exponential factor $e^{(-\gamma+\delta t_2)^2/2\delta}$ may cause exponent overflow or underflow. If this happens (we test for overflow in advance) or if the range of the precomputed tables is exceeded, we revert to method (M1). As explained in Appendix B, method (M3) handles all possible inputs correctly and gracefully.

Now, consider the problem of subdividing the projection of P into homogeneous regions R. Shirley and Tuchman [29] used a catalogue of four possible projection topologies for a tetrahedron, and Wilhelms and Van Gelder [33] used a line sweep algorithm for the case of a hexahedron. For a general convex polyhedral cell, we have used an incremental approach to build up a winged-edge data structure [22] for the subdivision. We add the projected edges one at a time, starting with an empty subdivision with a single unbounded face. The new projected edge is extended from its

TABLE 3
TYPICAL TIMINGS FOR THE VOLUME RENDERING ENGINE
FOR THE DIFFERENT INTEGRATION METHODS FOR BOTH 100,000 AND FOR 1,000,000 PIXEL IMAGES

TYPE OF INTEGRATION	NUMBER OF PIXELS	RENDERING ENGINE TIME	
		13,000 cells	600,000 cells
approximate method	100,000	0.17 min.	2.0 min.
	1,000,000	0.77 min.	7.6 min.
exact linear	100,000	0.24 min.	2.9 min.
	1,000,000	1.7 min.	16.3 min.
quadratic	100,000	0.27 min.	3.5 min.
	1,000,000	2.0 min.	22.8 min.

The total time for the HIAC system is the sum of the rendering engine time shown here, plus the multitiled sorting time given in Table 2 (0.12 minutes for 13,000 cells, and 9.5 minutes for 600,000 cells). All times are from an SGI Power Onyx using one R10000 CPU.

starting vertex, slicing one-by-one through the existing polygonal regions, and the winged-edge data structure is adjusted accordingly. When all edges have been added, the bounded polygons in the subdivision are the desired homogeneous regions.

Our principal current use of this hardware compositing of general polyhedra is for the slabs of Fig. 3, into which a linear tetrahedron is divided by breakpoints in the piecewise linear transfer functions. In this case, the HIAC system still uses the slow but precise back-to-front sort of Section 6. However, it also comes with a version of the much quicker approximate sort of Williams [36], [37], which is more useful for interactive applications.

8 RESULTS

Timings for the HIAC rendering engine are given in Table 3. Times are shown for the exact linear integration using the Dawson and Error integral, and for quadratic integration using Gaussian quadrature. For comparison, the times for the approximate method, as described in Section 4, are also shown. Times are given for images with 100,000 pixels and with 1,000,000 pixels. There is no significant difference in rendering time when several semitransparent illuminated isosurfaces are embedded in the image. Total volume rendering time is the sum of the time shown here plus the appropriate multitiled sorting time shown in Table 2.

Figs. 7, 8, 9, 10, 11, 12, 13, and 14 show volume rendered images of coolant velocity magnitude from a finite element simulation of coolant flow inside a component of the French Super Phoenix nuclear reactor. The data is defined on a mesh of 13,000 quadratic tetrahedra. Fig. 7 is an image created using the integration method for quadratic tetrahedra described in Section 5.2. This image is to be compared with the next four images, which were created using the same input specifications as used for Fig. 7, but different volume rendering methods. Fig. 8 was generated using the exact integration method for linear tetrahedra described in Section 5.1, by neglecting the data at the interior nodes. Fig. 9 was created using the approximate method described in Section 4, which assumes κ is a constant on each ray segment, with the value $(\kappa_f + \kappa_b)/2$. Fig. 10 was generated using the approximate method, but without slicing the cells into slabs. Fig. 11 was created using the hardware based polyhedron projection method, (M3), for sliced linear tetrahedra, described in Section 7. Differences between these images are clearly visible in the original images which are available for downloading at their full size and resolution in SGI *RGB* format at: http://www.llnl.gov/graphics/.

Figs. 12 and 13 show volume rendered images with embedded semitransparent illuminated isosurfaces; both were generated using the integration method for quadratic tetrahedra. Fig. 14 shows the same view as Fig. 12, but was created using the integration method for linear tetrahedra. Figs. 15 and 16 show volume rendered images of the density field from a finite element method simulation of air flow past an *F117a* jet aircraft flying at a 20 degree angle of attack. There is a vortex generated that breaks at the wing trailing edge. This data set is composed of 250,000 linear tetrahedra in a highly adaptively refined mesh. Fig. 15 was created using the exact integration method for linear tetrahedra. Subpixel splatting was turned on for the generation of this image; there were 9,200 projected cells covering less than two pixels, which were splatted. Fig. 16 was created using the approximate integration method, with splatting turned off.

9 FUTURE WORK AND CONCLUSION

The HIAC volume rendering system described in this paper creates highly accurate images of unstructured data sets whose cells are either linear or quadratic tetrahedra and whose facets are planar or nearly planar. The system was specifically designed to deal with data sets from the finite element method—but it is not limited to this type of data. Currently, the HIAC visibility ordering algorithm and the rendering engine will handle tetrahedra, bricks, prisms, pyramids, and wedges, or any combination thereof (zoo meshes); but will only use high accuracy integration for linear and quadratic tetrahedra. We plan to implement the procedure to perform accurate integration for linear and quadratic bricks and prisms, and cubic tetrahedra, which, along with the linear and quadratic tetrahedron, are the most widely used finite elements. The accurate integration procedure for these other elements is very similar to that for quadratic tetrahedra, and is discussed in Appendix C.

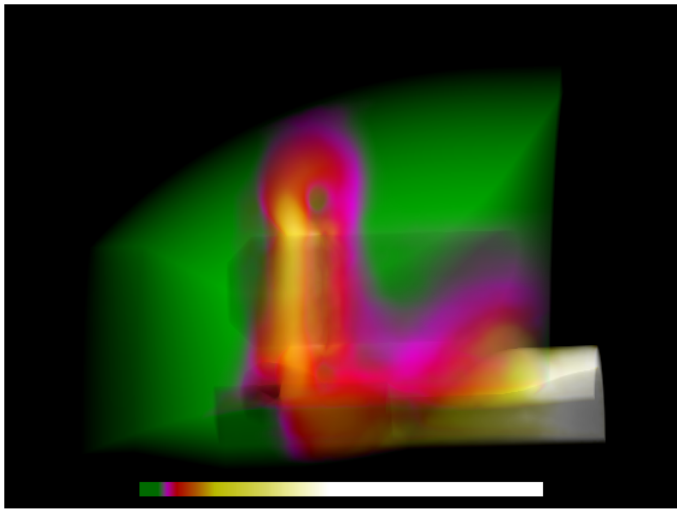


Fig. 7. Image created by integration method for quadratic tetrahedra.

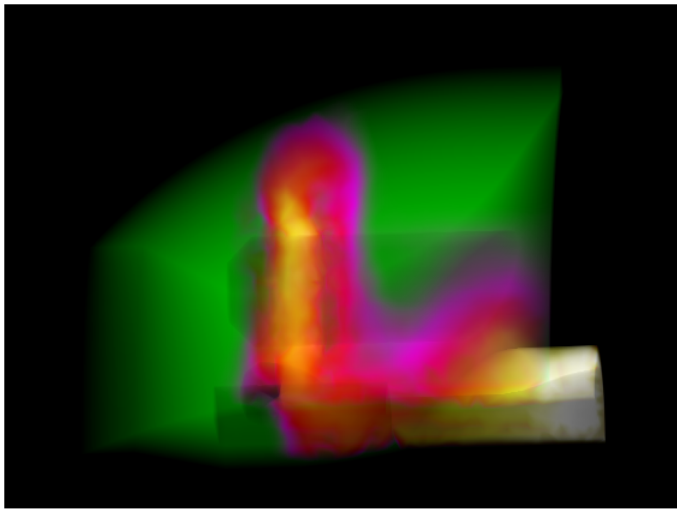


Fig. 8. Image created using exact integration method for linear tetrahedra.

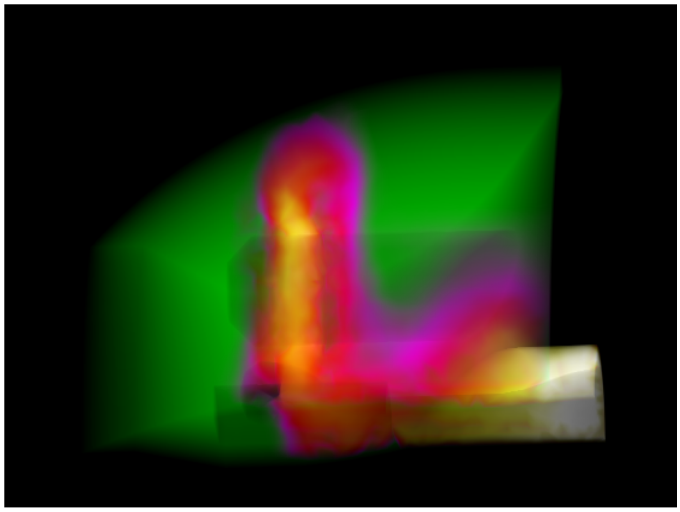


Fig. 9. Image created using the approximate method.

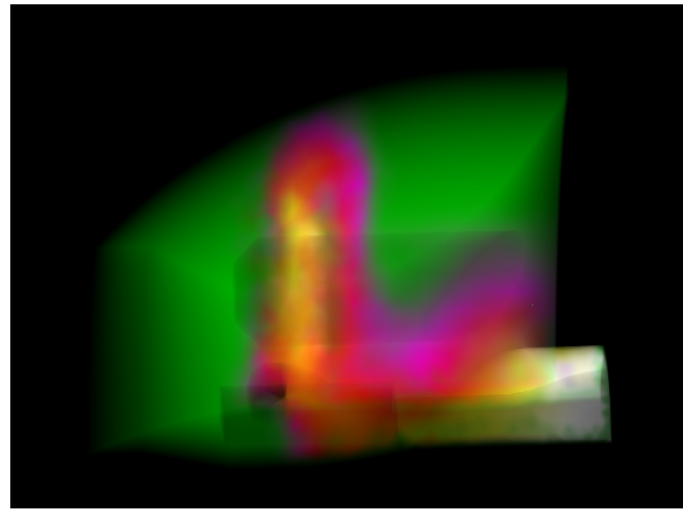


Fig. 10. Image created using the approximate method, without slicing cells.

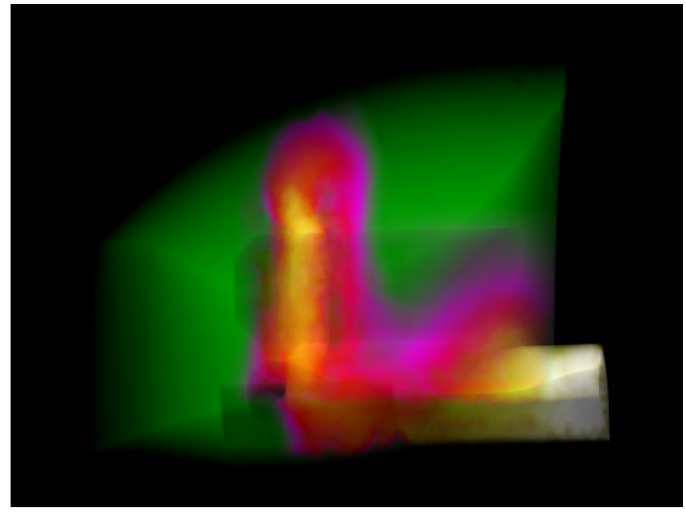


Fig. 11. Image created using the hardware based projection method (M3), with cell slicing.

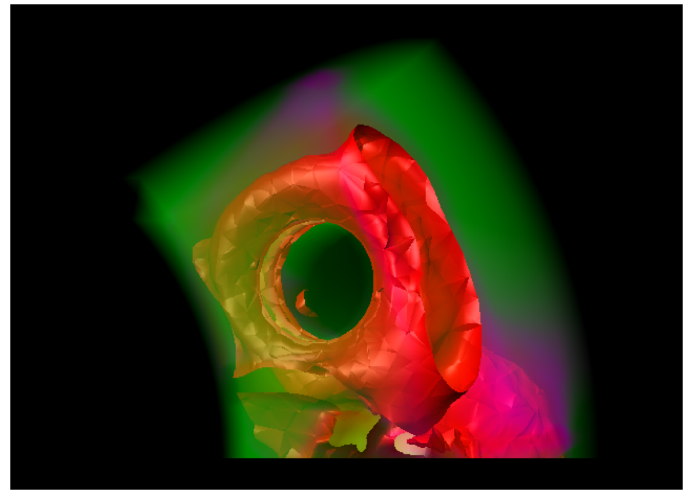


Fig. 12. Image created by integration method for quadratic tetrahedra, with embedded isosurfaces.

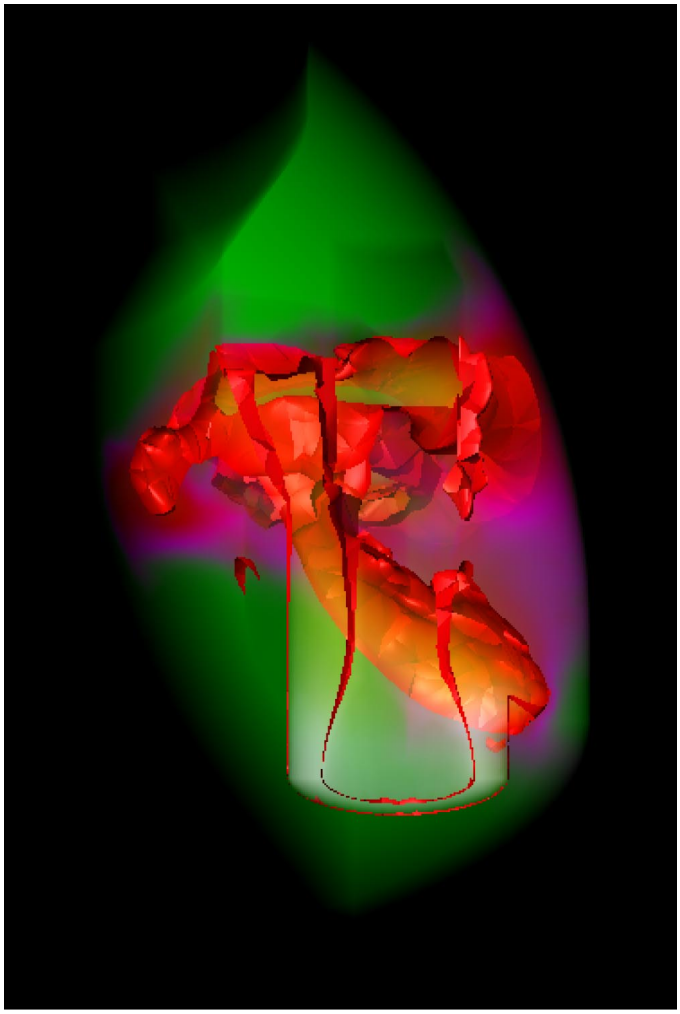


Fig. 13. Different view of Fig. 12.

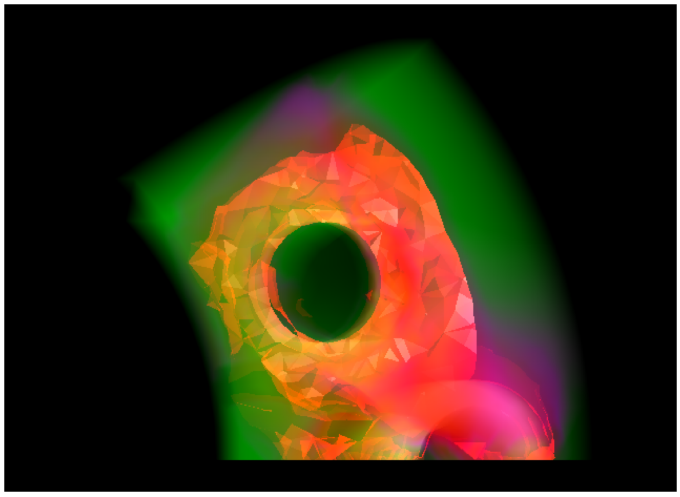


Fig. 14. Same view as Fig. 12, but using exact integration method for linear tetrahedra.

The system is not intended to be highly interactive, but, rather, to operate in batch mode to create high quality/accuracy images for publication or in-depth study, or for animations. (It would be nice if a graphical user interface were developed to facilitate the selection of the input parameters and the creation of the image specification file.) To improve the efficiency of the HIAC system, we have parallelized it for an MPP, the IBM SP-2. This parallel work, which includes load balancing techniques for unstructured meshes and methods for dealing with distributed finite element data sets has been implemented and will be described in a subsequent publication.

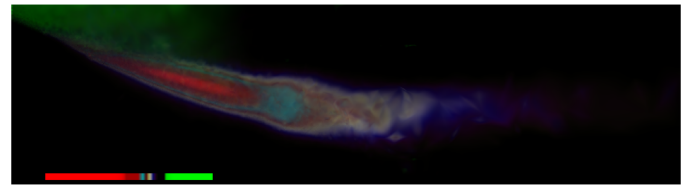


Fig. 15. Image created using exact integration method for linear tetrahedra, with subpixel splatting.

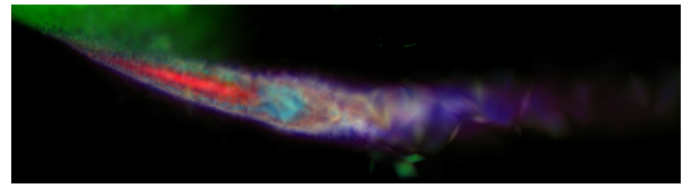


Fig. 16. Image created using approximate method, without subpixel splatting.

It may be possible to increase the efficiency of the system somewhat by the use of front-to-back compositing with early termination. This can be done on a pixel by pixel basis within cells, and will save unnecessary calls to the numerical routines for the complex error function for those pixels.

The exact visibility ordering algorithm described by Stein et al. [31], and used in the HIAC system, is $O(n^2)$ worst case. Therefore, it is not suitable for interactive use with the previewer. A faster exact depth ordering algorithm is described by de Berg et al. [2] which runs in time $O(n^{4/3+\epsilon})$ for any fixed $\epsilon > 0$. However, this algorithm, which is based on a general framework for computing and verifying linear orders extending implicitly defined binary relations, is quite theoretical and is not readily implemented. At present, Williams' [36] MPVO visibility ordering algorithm, which is a heuristic for nonconvex meshes, is used for the hardware assisted previewer described herein. However, the MPVO algorithm has a large storage requirement for its preprocessed data structures; therefore, it would be useful to investigate replacing this algorithm with a different sorting heuristic, such as one that sorts the cells by their centers of gravity. For data defined on 3D Delaunay triangulations, Karasick et al. [9] describe an efficient exact sorting algorithm based on sorting the tetrahedral cells by their powers.

An especially interesting and challenging project for the future is how to accurately volume render finite element data whose cells have been deformed by a parametric mapping function resulting in cells with highly curved facets. In this case, a ray through the volume may enter and exit the same cell more than once. Thus, a global visibility ordering may be impossible. In addition, the parametric mapping must be inverted before the scalar function can be evaluated.

As mentioned at the end of Section 5.3, our current splatting method is not always correct, and we are also working on analytic antialiasing using an exact geometric subdivision of the image plane by the projected edges of all cells.

The source code for the HIAC volume rendering system including the visibility ordering algorithm by Stein et al. [31], a previewer using Williams' interactive splatting system [37] (based on the projected tetrahedron algorithm of Shirley and Tuchman) and his MPVO visibility ordering algorithms [36], and the hardware-assisted projection and compositing system utilizing texture mapping hardware described in [31] and extended to deal with arbitrary convex polyhedra as described herein, is available at: http://www.llnl.gov/graphics/software.html. This research code is written in FORTRAN, C, and C++, and utilizes OpenGL for the hardware rendering.

APPENDIX A

This appendix describes the HIAC data structures for use with data sets having mixed cell types (zoo meshes). We allow the use of five different cells types in the input data set: tetrahedra, pyramids, prisms, wedges (cells with seven nodes, also called anvils), and bricks, each of which may be linear, quadratic, or cubic. For the higher order elements, we assume the extra nodes are located on the edges, with the exception of the cubic tetrahedron, which also has a node in the center of each face. (More than five cell types could be used as long as each cell type has a unique total number of nodes per cell.)

Each scalar field in the data set is stored in a floating point array with one element per node, in the same ordering as is used for the *xyz* array, described below. In addition to the data arrays, three other basic arrays are used: *elems*, *nodes*, and *xyz*. The elements of the *elems* and the *nodes* arrays are integer values and the elements of the *xyz* array are three-tuples of floating point values. Each cell has one entry in the *elems* array, its total number of nodes *nn*. However, rather than encoding *nn* directly, the *elems* array stores the cumulative total of *nn*. So cell i will have $nn = elems[i] - elems[i-1]$ nodes. The nodes for cell i are in the *nn* entries in the *nodes* array, starting with $nodes[elems[i]]$. The nodes stored in the *nodes* array are pointers to the coordinates of the nodes which are stored in the *xyz* array. The conventional nodes are specified first in the *nodes* array in a standard order, followed by the interior nodes, if any. We also keep a flag indicating whether the data set is linear, quadratic, or cubic. This flag disambiguates different types of cells with the same number of nodes, e.g., a quadratic brick has the same number of nodes as a cubic tetrahedron. We assume elements of different orders (linear, quadratic, cubic) will not be combined in one mesh.

APPENDIX B

This appendix gives implementation details for the evaluation of (8) of Section 5.1. In the notation of that section, let

$$f_1 = \frac{\gamma + \delta t_1}{\sqrt{2|\delta|}}$$

and

$$f_2 = \frac{\gamma + \delta t_2}{\sqrt{2|\delta|}}.$$

Since the numerators represent the extinction coefficients $\tau(t_1)$ and $\tau(t_2)$ at the two endpoints of the ray segment, f_1 and f_2 are nonnegative real numbers. (If numerical inaccuracy results in a slightly negative value, it is replaced by zero.)

Note that the *erfi* terms in (8) are multiplied by a factor of $e^{-f_2^2}$ if $\delta > 0$. When $erfi(f_2)$ is replaced by the Dawson integral from (9), the above factor $e^{-f_2^2}$ cancels the factor $e^{f_2^2}$ in $erfi(f_2) = \frac{2}{\sqrt{\pi}} e^{f_2^2} D(f_2)$. Thus, the product $e^{-f_2^2}(erfi(f_2) - erfi(f_1))$ can be evaluated as $\frac{2}{\sqrt{\pi}}(D(f_2) - e^{f_1^2 - f_2^2} D(f_1))$. If $\delta > 0$, $f_2 > f_1$, so $f_1^2 - f_2^2$ is negative, and if it becomes so negative that it leaves the valid domain of the exponential (causes underflow), then it is safe to replace $e^{f_1^2 - f_2^2}$ by zero.

If $\delta < 0$, then the corresponding product is

$$\frac{1}{i} e^{f_2^2}\left(erfi\left(\frac{f_2}{i}\right) - erfi\left(\frac{f_1}{i}\right)\right) = e^{f_2^2}\left(erf(f_1) - erf(f_2)\right).$$

(The $\frac{1}{i}$ in front comes from the $\delta^{-2.5}$ factor in (8).) We use an approximation to $erf(x)$ for $x > 0$, given in [24] under the guise of the *complementary error function* $1 - erf(x)$, of the form $erf(x) \cong 1 - ue^{(-x^2 + p(u))}$, where $u = \frac{1}{1+0.5x}$, and $p(u)$ is a ninth degree Chebyshev polynomial selected to give an accurate fit to the tail of the error function. Thus, we get $e^{f_2^2}(erf(f_2) - erf(f_1)) \cong u_2 e^{p(u_2)} - u_1 e^{f_2^2 - f_1^2} e^{p(u_1)}$, which avoids loss of accuracy when f_1 and f_2 are large, since the 1s cancel. In this case, $f_1 > f_2$, and we can again set $e^{f_2^2 - f_1^2}$ to zero if $f_2^2 - f_1^2$ is too negative.

APPENDIX C

Section 3 discussed interpolation functions for the linear tetrahedron and the quadratic tetrahedron. This appendix discusses interpolation functions for the other common cells used in the finite element method—the prism, the cubic tetrahedron, and the quadratic brick—and outlines how the volume rendering methods described in this paper may be extended to include these other cells.

The interpolation function for the six-node pentahedron or prism is:

$$f(x, y, z) = c_1 + c_2 x + c_3 y + c_4 z + c_5 xz + c_6 yz. \quad (13)$$

The prism is mainly used as a transition element to "glue" together tetrahedra and bricks in meshes that use a combination of different element types. FEM simulation code that uses such a mesh of mixed element types is often called a *zoo code*.

The interpolation function for the cubic tetrahedron, which has 20 nodes, is a cubic polynomial, and is complete through the cubic terms of the 3D power series. Therefore,

its interpolation function has all the terms shown in (1). This cell has two interior nodes per edge, usually at $\frac{1}{3}$ and $\frac{2}{3}$ of the edge, as well as a node at the center of gravity of each facet of the cell. Here, the field varies cubically along any ray through the cell.

The quadratic brick, with 20 nodes, has the following interpolation function:

$$f(x, y, z) = c_1 + c_2x + c_3y + c_4z + c_5x^2 + c_6y^2 + c_7z^2 + c_8xy + c_9xz + c_{10}yz + c_{11}x^2y + c_{12}x^2z + c_{13}xy^2 + c_{14}xz^2 + c_{15}y^2z + c_{16}yz^2 + c_{17}xyz + c_{18}x^2yz + c_{19}xy^2z + c_{20}xyz^2. \quad (14)$$

The brick's interior nodes are located at the centers of its edges.

At present, the tetrahedron and brick, in their linear and quadratic forms, dominate practical applications [14]. Therefore, in this paper, we limit our coverage of accurate volume rendering methods to these cells and the prism. Similar techniques to those given in this paper can be applied to other types of cells, provided the interpolation function is fourth degree or lower. We have implemented the high accuracy volume rendering methods for the four-node tetrahedron and 10-node tetrahedron, and we describe below how one might extend the system to deal with the linear brick and prism, and the quadratic brick. At present, the HIAC system will also handle bricks, prisms, pyramids, and wedges, although not with the highest precision. When we implement the accurate rendering scheme for linear bricks and prisms described here, these elements will be not be subdivided into tetrahedra prior to rendering.

The same basic procedure described in Section 5.2 for quadratic tetrahedra can be used for high accuracy rendering of the quadratic brick as well as the linear brick and prism. The interpolation equation for the quadratic brick is given in (14); it is a fourth order polynomial in x, y, and z, so its evaluation along a linear ray gives a fourth order polynomial $f(t)$ in the ray parameter t. The points on a ray where the polynomial takes on a contour value s_i can be found analytically by the closed form noniterative solution of the quartic equation first published by Ferarro, see [40]. Here, a case analysis, similar to, but more complex than, the one described above for quadratic tetrahedra, is required to find the regions where the polynomials $\tau(t)$ and $g(t)$ are smoothly varying, i.e., include no breakpoints. A diagram of the case analysis for finding the ray segments is given in Fig. 17. The points t_1, t_2, and t_3, where $f'(t) = 0$ separate the monotone ranges of $f(t)$, can be found as roots of the cubic polynomial $f'(t)$. Either Gaussian quadrature or the power series method of Novins and Arvo [21] can be used to do the integration.

The interpolation function for the linear brick is given in (2). It is trilinear, therefore, contours within the cells are curved and, so, the methods described above pertain. The function $f(t)$ is cubic because of the xyz term in (2). To find the roots of the cubic polynomial, $f(t) - s_i = 0$, we can use the closed form solution given in [42].

The linear prism has an interpolation function given by (13). This has the bilinear terms xz and yz, and, so, contour surfaces in the interior of the linear prism will be curved. Substituting the ray parameterization into the interpolation equation, we get a quadratic polynomial which can be analyzed on a case by case basis similarly to that used for the quadratic tetrahedron.

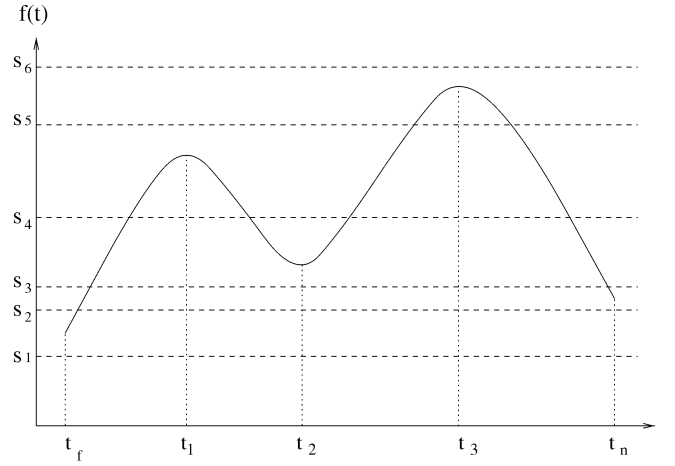


Fig. 17. Analysis of ray segment through a quadratic brick element.

The interpolation equation for the cubic tetrahedron is:

$$f(x, y, z) = c_1 + c_2x + c_3y + c_4z + c_5x^2 + c_6y^2 + c_7z^2 + c_8xy + c_9xz + c_{10}yz + c_{11}x^3 + c_{12}y^3 + c_{13}z^3 + c_{14}x^2y + c_{15}x^2z + c_{16}xy^2 + c_{17}xz^2 + c_{18}y^2z + c_{19}yz^2 + c_{20}xyz. \quad (15)$$

Since this equation is of degree three, it can be dealt with in a similar way to that described above for the other higher order elements.

ACKNOWLEDGMENTS

We are grateful to Roger Crawfis of Ohio State University for his contribution to the *scanvol* code which was modified for use in the system described herein. We also appreciate technical assistance from Barry Becker of Silicon Graphics, Inc., Kwan-Liu Ma at ICASE, and Mark Duchaineau at LLNL. Peter L. Williams is greatly indebted to Sam Uselton and Tom Lasinski at NAS, NASA Ames Research Center, for their generous summer support for three years and for equipment loans without which this work would not be possible. He is also grateful for the use of the Large-Scale Interactive Visualization Environment (LIVE) at NAS, NASA Ames, which was used to generate the images in this paper. Peter L. Williams and Nelson L. Max received summer support for two years, arranged by Becky Springmeyer, from the Accelerated Strategic Computing Initiative (ASCI). Robert Haimes of MIT graciously provided the *F117a* data set and Bruno Nitrosso at Electricité de France provided the Super Phoenix data set. This work was partially performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract number W-7405-ENG-48. Peter L. Williams performed the majority of his contribution to this research while a summer visitor at Lawrence Livermore National Laboratory, and at NAS, NASA Ames Research Center.

REFERENCES

[1] H.E. Cline, W.E. Lorensen, S. Ludke, C.R. Crawford, and B.C. Teeter, "Two Algorithms for the Three-Dimensional Reconstruction of Tomograms," *Medical Physics*, vol. 15, no. 3, pp. 64-72, May 1988.

[2] M. de Berg, M. Overmars, and O. Schwarzkopf, "Computing and Verifying Depth Orders," *SIAM J. Computing*, vol. 23, pp. 437-446, Apr. 1994.

[3] J. Foley, A. van Dam, S. Feiner, and J. Hughes, *Computer Graphics Principles and Practice*, second ed. Addison-Wesley, 1990.

[4] R. Gallagher and J. Nagtegaal, "An Efficient 3D Visualization Technique for Finite Element Models and Other Coarse Volumes," *Computer Graphics*, vol. 23, no. 3, pp. 185-192, July 1989.

[5] M.P. Garrity, "Raytracing Irregular Volume Data," *Computer Graphics*, vol. 24, no. 5, pp. 35-40, Nov. 1990.

[6] C. Giertsen, "Volume Visualization of Sparse Irregular Meshes," *Computer Graphics*, vol. 12, no. 2, pp. 40-48, Mar. 1992.

[7] C. Giertsen and A. Tuchman, "Fast Volume Rendering with Embedded Geometric Primitives," *Visual Computing—Integrating Computer Graphics with Computer Vision*, T.L. Kunii, ed., pp. 253-271. Springer Verlag, 1992.

[8] R. Haimes, "Visual3: Interactive Unsteady Unstructured 3D Visualization," *AIAA Paper 91-0794*, Reno Nev., Jan. 1991.

[9] M.S. Karasick, D. Lieber, L.R. Nackman, and V.T. Rajan, "Visualization of Three-Dimensional Delaunay Meshes," *Algorithmica*, vol. 19, pp. 114-128, 1997.

[10] K. Koyamada, "Volume Visualization for the Unstructured Data," *SPIE Vol. 1259 Extracting Meaning from Complex Data: Processing, Display, Interaction*, 1990.

[11] W.E. Lorensen and H.E. Cline, "Marching Cubes: A High Resolution 3-D Surface Construction Algorithm," *Computer Graphics*, vol. 21, no. 4, pp. 163-169, July 1987.

[12] B.A. Lucas, "A Scientific Visualization Renderer," *Proc. Visualization '92*, pp. 227-234, Boston, Oct. 1992.

[13] K-L. Ma, "Parallel Volume Ray-Casting for Unstructured-Grid Data on Distributed-Memory Architectures," *Proc. ACM Parallel Rendering Symp.*, pp. 23-30, Oct 1995.

[14] R.H. MacNeal, "Finite Elements: Their Design and Performance," New York: Marcel Dekker, 1994.

[15] X. Mao, "Splatting of Nonrectilinear Volumes Through Stochastic Resampling," *IEEE Trans. Visualization and Computer Graphics*, vol 2, no. 2, pp. 156-170, June 1996.

[16] N. Max, P. Hanrahan, and R. Crawfis, "Area and Volume Coherence for Efficient Visualization of 3D Scalar Functions," *Computer Graphics*, vol. 24, no. 5, pp. 27-33, Nov. 1990.

[17] N. Max, "Optical Models for Direct Volume Rendering," *IEEE Trans. Visualization and Computer Graphics*, vol. 1, no. 2, pp. 99-108, June 1995.

[18] K. Mueller and R. Yagel, "Fast Perspective Volume Rendering with Splatting Utilizing a Ray-Driven Approach," *Proc. Visualization '96*, pp. 65-72, Oct 1996.

[19] M. Newell, R. Newell and T. Sancha, "Solution to the Hidden Surface Problem," *Proc ACM Nat'l Conf.*, pp. 443-450, 1972.

[20] M. Newell, "The Utilization of Procedure Models in Digital Image Synthesis," PhD thesis, Univ. of Utah, 1974 (UTEC-CSc-76-218 and NTIS AD/A 039 008/LL).

[21] K. Novins and J. Arvo, "Controlled Precision Volume Integration," *Proc. 1992 Workshop Volume Visualization*, pp. 83-89, Boston, Oct. 1992.

[22] J. O'Rourke, *Computational Geometry in C*. Cambridge Univ. Press, 1995.

[23] C.E. Prakash, "Parallel Voxelization Algorithms for Volume Rendering of Unstructured Grids," PhD Thesis, Supercomputer Centre, Indian Inst. of Science, 1996.

[24] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery, *Numerical Recipes in Fortran*. Cambridge Univ. Press, 1992.

[25] D.F. Rogers, *Procedural Elements for Computer Graphics*. New York: McGraw-Hill, 1985.

[26] G.B. Rybicki, "Dawson Integral and the Sampling Theorem," *Computers in Physics*, vol. 3, no. 2, pp. 85-87, 1989.

[27] P. Sabella, "A Rendering Algorithm for Visualizing 3D Scalar Fields," *Computer Graphics*, vol. 22, no. 4, pp. 51-58, Aug. 1988.

[28] R. Sedgewick, *Algorithms in C++*, pp. 359–371. Addison-Wesley 1992.

[29] P. Shirley and A. Tuchman, "A Polygonal Approximation to Direct Scalar Volume Rendering," *Computer Graphics*, vol. 24, no. 5, pp. 63-70, Nov. 1990.

[30] C. Silva and J.S.B. Mitchell, "The Lazy Sweep Ray Casting Algorithm for Rendering Irregular Grids," *IEEE Trans. Visualization and Computer Graphics*, vol. 3, no. 2, pp. 142-157, Mar.-June 1997.

[31] C. Stein, B. Becker, and N. Max, "Sorting and Hardware Assisted Rendering for Volume Visualization," *Proc. 1994 Symp. Volume Visualization*, pp. 83-90, Washington, D.C., Oct. 1994.

[32] L. Westover, "Interactive Volume Rendering," *Proc. 1989 Workshop Volume Visualization*, pp. 9-16, Chapel Hill, N.C., May 1989.

[33] J. Wilhelms and A. Van Gelder, "A Coherent Projection Approach for Direct Volume Rendering," *Computer Graphics*, vol. 25, no. 4, pp. 275-284, July 1991.

[34] J. Wilhelms, "Pursuing Interactive Visualization of Irregular Grids," *The Visual Computer*, vol. 9, pp. 450-458, 1993.

[35] J. Wilhelms, A. Van Gelder, P. Tarantino, and J. Gibbs, "Hierarchical and Parallelizable Direct Volume Rendering for Irregular and Multiple Grids," *Proc. Visualization '96*, pp. 57-64, Oct. 1996.

[36] P.L. Williams, "Visibility Ordering Meshed Polyhedra," *ACM Trans. Graphics*, vol. 11, no. 2, pp. 103-126, Apr. 1992.

[37] P.L. Williams, "Interactive Splatting of Nonrectilinear Volumes," *Proc. Visualization '92*, pp. 37-44, Boston, Oct. 1992.

[38] P.L. Williams and N.L. Max, "A Volume Density Optical Model," *Proc. 1992 Workshop Volume Visualization*, pp. 61-68, Boston, Oct. 1992.

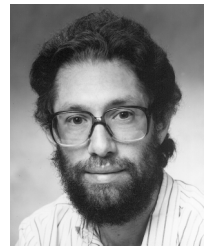
[39] P.L. Williams and S.A. Uselton, "Metrics and Generation Specifications for Comparing Volume Rendered Images," NASA-Ames TR NAS-96-021, Dec. 1996, http://science.nas.nasa.gov/Pubs/TechReports/NASreports/NAS-96-021/NAS-96-021.html.

[40] P.H. Winston and B.K.P. Horn, *Lisp*, second edition. Reading, Mass.: Addison Wesley, 1984.

[41] R. Yagel, D. Reed, A. Law, P-W. Shih, and N. Shareef, "Hardware Assisted Volume Rendering of Unstructured Grids by Incremental Slicing," *Proc. 1996 Symp. Volume Visualization*, pp. 55-62, Nov. 1996.

[42] *Handbook of Chemistry and Physics*. Cleveland, Ohio: Chemical Rubber Publishing.

Peter L. Williams received the PhD in computer science from the University of Illinois at Urbana-Champaign, and the BS in engineering-physics from the University of California at Berkeley. He has taught computer science at Vassar College, the University of Connecticut at Storrs, and Harvey Mudd College. He is currently a visiting scientist at the IBM T.J. Watson Research Center, where he is a member of the IBM Data Explorer research group. At IBM, Dr. Williams is developing a distributed visualization system for use on the IBM SP-2 for the National Laboratories' terascale computing effort. His research interests include graphics, scientific visualization, volume rendering (especially unstructured data), and high performance parallel and distributed computing.



Nelson L. Max has research interests in the areas of scientific visualization, volume and flow rendering, computer animation, molecular graphics, and realistic computer rendering, including shadow and radiosity effects. Since 1977, he has been a computer scientist at Lawrence Livermore National Laboratory, and has been teaching part time at the University of California, Davis, currently as a 50 percent professor of applied Science. Dr. Max has taught mathematics and computer science at the University of California at Berkeley, the University of Georgia, Carnegie Mellon University, and Case Western Reserve University. He was director of the U.S. National Science Foundation supported Topology Films Project in the early 1970s, which produced computer animated educational films on mathematics. He has worked in Japan for three and a half years as codirector of two Omnimax (hemisphere screen) stereo films for international expositions, showing the molecular basis of life. For the past eight years, he has concentrated on volume, vector field, and flow visualization for 3D simulations.

Clifford M. Stein graduated from Harvey Mudd College and received his PhD in computer science from the University of California at Davis in 1998. He was employed at Lawrence Livermore National Laboratory from 1992 to 1997. His current research interests include animation, rendering, and physically-based modeling.