A Fast High Accuracy Volume Renderer for Unstructured Data

Kenneth Moreland*
Sandia National Laboratories

Edward Angel†
University of New Mexico

ABSTRACT

In this paper, we describe an unstructured mesh volume renderer. Our renderer is interactive and accurately integrates light intensity an order of magnitude faster than previous methods. We employ a projective technique that takes advantage of the expanded programmability of the latest 3D graphics hardware. We also analyze an optical model commonly used for scientific volume rendering and derive a new method to compute it that is very accurate but computationally feasible in real time. We demonstrate a system that can accurately produce a volume rendering of an unstructured mesh with a first-order approximation to any classification method. Furthermore, our system is capable of rendering over 300 thousand tetrahedra per second yet is independent of the classification scheme used.

CR Categories: I.3.7 [Computing Methodologies]: Computer Graphics—Three-Dimensional Graphics and Realism

Keywords: volume rendering

1 INTRODUCTION

Commodity graphics hardware directly supports only zero-, one-, and two-dimensional primitives (points, lines, and polygons). The reason is simple. An opaque solid object is visually indistinguishable from a hollow surface when viewed from the outside. However, a photorealistic scene may involve any number of translucent volumetric objects such as clouds, dust, steam, or fog. In addition, **direct volume rendering** is a popular technique for visualizing volumetric data from sources such as scientific simulations and medical scanners. Light in a translucent volume behaves very differently from light against a surface comprising polygons.

Interest in volumetric rendering has spawned several techniques for utilizing 3D commodity graphics hardware. Texture-based sampling [1] for rectilinear grids (i.e. 3D arrays) and projection [21] for unstructured meshes (i.e. collections of cells with no restrictions on cell type or connectivity) are the two most popular techniques. Our focus is on unstructured meshes. Unstructured meshes tend to be coarser than their rectilinear counterparts as freedom in cell size, shape, and connectivity allow modelers to fit a mesh more accurately with fewer cells. However, the irregular cell connections make rendering more challenging and errors more noticeable compared to regular data.

Color calculations for volumes are more computationally intensive than those for surfaces. The calculations are so complicated that, until now, no volume rendering system is capable of performing them interactively for even a first-order approximation of luminance and attenuation. Present interactive systems have to resort to rough approximations or a finite set of precomputed values attached to a simple transfer function. In this paper, we describe a means of performing this lighting computation that is fast enough to compute in real time and can be implemented in a fragment program. We do this computation by adaptively sampling the volume and performing quick, but close, approximations for lighting calculations.

This paper proceeds as follows. First, we briefly summarize current hardware accelerated unstructured mesh volume rendering. Second, we discuss the ray casting and ray integration, respectively, of our volume rendering system. Third, we discuss the results we achieved with our system and draw some conclusions.

*e-mail: kmorel@sandia.gov
†e-mail: angel@cs.unm.edu

IEEE Symposium on Volume Visualization and Graphics 2004
October 11-12, Austin, Texas, USA

2 PREVIOUS WORK

For optimal rendering performance, we leverage unstructured mesh volume rendering work that takes advantage of the highly optimized and readily available commodity graphics hardware. Shirley and Tuchman [21] presented the **projected tetrahedra** algorithm for rendering unstructured volumes over a decade ago. It has since been a popular method for volume rendering as it is effective, easy to implement, and applicable to any unstructured mesh (as any type of cell can be decomposed into tetrahedra).

One variation of the projected tetrahedra algorithm [28] projects hexahedra instead of tetrahedra. Another variation [32] modifies the projection classes to calculate them within a vertex processor. Weiler, Kraus, and Ertl [25] present a unique projection-based rendering algorithm called **view independent cell projection**. Unlike the projected tetrahedra algorithm, which renders triangular regions with linearly varying depth, Weiler's algorithm renders the front faces intact. The algorithm uses the rasterizer to also interpolate the volume properties of and distance to the other three faces of the tetrahedron. The algorithm then chooses the correct distance and back face intersection during fragment processing. Weiler and colleagues [27] modify the algorithm to compute the distance between front and back faces in the fragment processor rather than the vertex processor.

When using cell projection, a rendering system must also sort the cells. There are many approaches to cell sorting, but one of the most popular is the MPVO algorithm by Williams [29]. Over the years, researchers have made many improvements to the algorithm [2, 14, 22].

Weiler and colleagues [26] show that the current generation of 3D graphics cards can also perform ray casting of unstructured meshes. However, the ray casting approach relies heavily on the graphics card's fragment processor. We believe that we can more effectively balance the computation among the CPU, GPU vertex processor, and GPU fragment processor by using cell projection rather than ray casting.

The model we use for light transport through a volumetric cloud is the absorption and emission model given by Max [16]. Given a single ray of light passing through a volume parameterized such that light enters the volume at $s = 0$ and exits the volume at $s = D$, the intensity of light emanating from the volume is

$$I_D = I_0 e^{-\int_0^D \tau(t)dt} + \int_0^D L(s)\tau(s)e^{-\int_s^D \tau(t)dt}ds \qquad (1)$$

where I_0 is the intensity of light as it enters the volume, $\tau(s)$, the **attenuation coefficient**, describes the density of the volume, and

$L(s)$, the **luminance**, describes the amount of light emitted per object density. We refer to Equation 1 as the **volume rendering integral**.

Williams and Max [30] were the first to solve the volume rendering integral with linearly interpolated attributes. Solving Equation 1 for $L(s) = L_\mathrm{b}(1-s/D) + L_\mathrm{f}(s/D)$ and $\tau(s) = \tau_\mathrm{b}(1-s/D) + \tau_\mathrm{f}(s/D)$, we get the following complicated equation.

$$I_D = \begin{cases} I_0 e^{-D\frac{\tau_\mathrm{b}+\tau_\mathrm{f}}{2}} + L_\mathrm{f} - L_\mathrm{b} e^{-D\frac{\tau_\mathrm{b}+\tau_\mathrm{f}}{2}} \\ \quad + (L_\mathrm{b} - L_\mathrm{f}) \frac{1}{\sqrt{D(\tau_\mathrm{b}-\tau_\mathrm{f})}} e^{\frac{D}{2(\tau_\mathrm{b}-\tau_\mathrm{f})}\tau_\mathrm{f}^2} \sqrt{\frac{\pi}{2}} \\ \quad \left[\mathrm{erf}\left(\tau_\mathrm{b}\frac{\sqrt{D}}{\sqrt{2(\tau_\mathrm{b}-\tau_\mathrm{f})}}\right) - \mathrm{erf}\left(\tau_\mathrm{f}\frac{\sqrt{D}}{\sqrt{2(\tau_\mathrm{b}-\tau_\mathrm{f})}}\right)\right] & \tau_\mathrm{b} > \tau_\mathrm{f} \\ I_0 e^{-D\frac{\tau_\mathrm{f}+\tau_\mathrm{b}}{2}} + L_\mathrm{f} - L_\mathrm{b} e^{-D\frac{\tau_\mathrm{f}+\tau_\mathrm{b}}{2}} \\ \quad + (L_\mathrm{b} - L_\mathrm{f}) \frac{1}{\sqrt{D(\tau_\mathrm{f}-\tau_\mathrm{b})}} e^{-\frac{D}{2(\tau_\mathrm{f}-\tau_\mathrm{b})}\tau_\mathrm{f}^2} \sqrt{\frac{\pi}{2}} \\ \quad \left[\mathrm{erfi}\left(\tau_\mathrm{f}\frac{\sqrt{D}}{\sqrt{2(\tau_\mathrm{f}-\tau_\mathrm{b})}}\right) - \mathrm{erfi}\left(\tau_\mathrm{b}\frac{\sqrt{D}}{\sqrt{2(\tau_\mathrm{f}-\tau_\mathrm{b})}}\right)\right] & \tau_\mathrm{b} < \tau_\mathrm{f} \\ I_0 e^{-\tau_\mathrm{b} D} + L_\mathrm{b}\left(\frac{1}{\tau_\mathrm{b} D} - \frac{1}{\tau_\mathrm{b} D} e^{-\tau_\mathrm{b} D} - e^{-\tau_\mathrm{b} D}\right) \\ \quad + L_\mathrm{f}\left(1 + \frac{1}{\tau_\mathrm{b} D} e^{-\tau_\mathrm{b} D} - \frac{1}{\tau_\mathrm{b} D}\right) & \tau_\mathrm{b} = \tau_\mathrm{f} \end{cases} \quad (2)$$

As we can see, Equation 2 has many terms and is expensive to compute. Furthermore, Equation 2 is susceptible to numerical inaccuracies. (Williams, Max, and Stein [31] give transformations that increase the numerical stability.) Consequently, calculating Equation 2 for real-time or interactive systems is seldom feasible.

Shirley and Tuchman [21] use an approximation in which they linearly interpolate the attenuation but use a constant color. The luminance they use is the mean luminance of the segment. In this case, the volume rendering integral reduces to

$$I_D \approx I_0 e^{-D\frac{1}{2}(\tau_\mathrm{b}+\tau_\mathrm{f})} + \frac{1}{2}(L_\mathrm{b} + L_\mathrm{f})\left(1 - e^{-D\frac{1}{2}(\tau_\mathrm{b}+\tau_\mathrm{f})}\right) \quad (3)$$

Max, Hanrahan, and Crawfis [17] generalize this approach for a constant luminance and any integrable function for attenuation. They use the mean luminance when it is not constant (again resulting in Equation 3). Kniss and colleagues [12] solve the volume rendering equation for attenuation with a normal distribution. They also use a weighted sum to approximate the luminance with a constant value.

To alleviate the high computational overhead of integrating viewing rays through a volume, Röttger, Kraus, and Ertl [20] introduce **pre-integration**. Pre-integration performs the classification and ray integration through a simple texture lookup. The classification performed during pre-integration is limited to 1D transfer functions although shading can be estimated [18].

Because the textures in pre-integration perform classification, an application must repopulate them for every transfer function change, which can be a serious computational overhead. Subsequent research accelerates transfer function construction by reducing the table size needed [7], utilizing the GPU [19], or performing incremental computations [26].

3 Cell Projection

In this section, we describe how we take a collection of cells and determine how each viewing ray intersects each cell.

3.1 Balanced Cell Projection

Our cell projection algorithm starts with the approach from Weiler, Kraus, and Ertl [25] called View Independent Cell Projection (VICP). Like all other projection algorithms, VICP determines the intersections of all viewing rays with a cell at once. Since VICP deals exclusively with tetrahedra, which are by definition convex, finding this intersection reduces to finding the intersection of the viewing ray with two of the tetrahedron's faces.

VICP finds the intersections of the viewing ray with the front faces simply by rasterizing them. It then determines the back face intersection on a per fragment basis. To find the back face intersection, VICP intersects the viewing ray with the plane of each face that is not the front face. In the example shown in Figure 1, VICP finds the viewing ray intersection of face f_0 by rasterizing that face. VICP also intersects the viewing ray with the planes of faces f_1, f_2, and f_3. The rear face will always be the intersection that is nearest and behind the front face. In Figure 1, this is clearly face f_1.

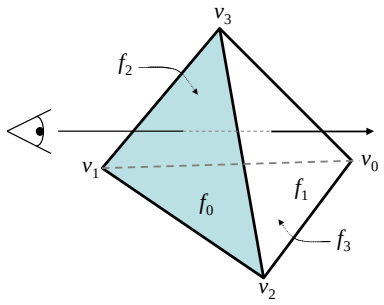


Figure 1: Viewing-ray–plane intersections performed by view independent cell projection.

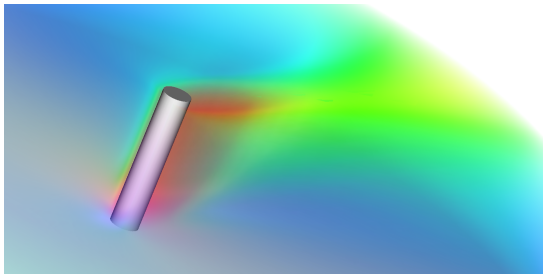
At the time of the VICP algorithm's creation, the fragment processor was not yet powerful enough to compute the intersection of a ray with three faces. So instead, VICP performs these intersections at the vertices in the vertex processor and then uses the rasterizer to interpolate the correct intersections for each viewing ray.

Weiler, Kraus, and Ertl [25] report that the transfer of data from CPU to GPU is a major bottleneck for VICP. Consequently, they use their cell projection with a ray integration method that is order independent. Therefore, they do not have to sort the cells based on the view and can keep the data stored directly on the graphics card. Nevertheless, even when transferring data between CPU and GPU every frame, we have found VICP to be competitive with other hardware accelerated cell projection methods [21, 32].

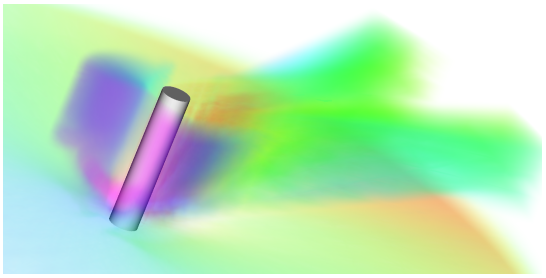
Because we plan to use a ray integration method that requires us to sort cells based on viewpoint (and therefore pass data to the GPU at every frame), there is no penalty for us to perform some calculations on the CPU. So rather than move operations from the vertex processor to the fragment processor as is done in [27], we move operations back into the CPU to minimize the amount of data that must be transferred.

Specifically, we perform ray-face intersections on the CPU. The ray-face intersection determines the distance from a vertex to a face along the viewing ray and the scalar value on that face. To perform this intersection (as [25] describes), we need the plane equation for the face. Furthermore, we need the gradient of the scalar to compute the scalar value at the ray-face intersection, totaling at least six floating-point values. If we perform the ray-face intersection on the CPU, we need to pass only the scalar and the distance-to-vertex values (two floating-point values).

As Weiler, Kraus, and Ertl [25] point out, at each vertex we need to do only one ray-face intersection. A vertex is shared by three of the four faces of the tetrahedron. For example in Figure 1, vertex v_1

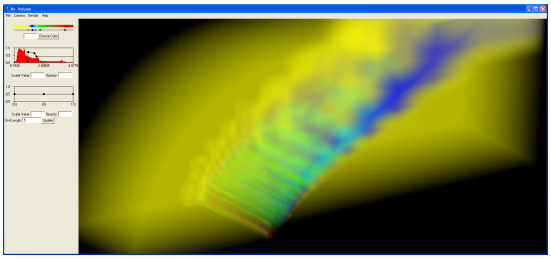


(a) Volume rendering with a traditional 1D transfer function

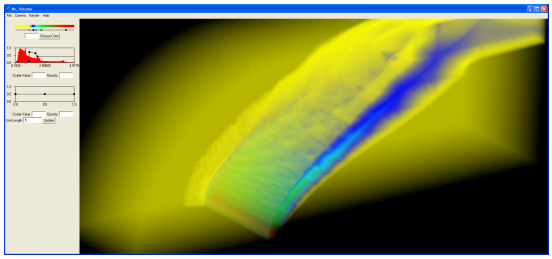


(b) Volume rendering with opacity based on boundary estimation from first and second derivatives.

Figure 2: Comparison of renderings with the oxygen post data set. The surface of the post is added for context. Changing the opacity based on the derivatives gives a clearer view of the turbulence around the post.



(a) Mesh sampled at vertices.



(b) Mesh adaptively sampled.

Figure 3: The effect of aliasing when sampling a mesh. Both images have the same simple transfer function classification, which has a sharp opacity transition to highlight an isosurface. The rendering on the left samples only at the vertices of the cells, which induces aliasing. The rendering on the right adaptively samples the cells.

touches faces f_0, f_2, and f_3. Therefore, the intersection of any ray through the vertex and these three faces is trivially the vertex itself; we need neither to perform these ray-face intersections nor pass the resulting data to the GPU.

As prescribed by Weiler [25], when rasterizing a face, we need to interpolate the scalar of the face, the scalars of the intersections of the three other faces (as a three-tuple), and the distance to all three other faces (as a three-tuple). In our implementation, we interpolate the intersection values of all four faces as two four-tuples. Obviously, the interpolation of the fourth value in these tuples is a wasted computation, but due to the redundancy in the graphics hardware, the added computation does not affect the execution time.

The advantage of using four-tuples for the distance and scalar vectors is that the indexing will be consistent among the tetrahedron's faces. Therefore, rather than pass data for each vertex separately for each face, we can pass the vertex data once per vertex and use indexed mode to draw the triangles, reducing the total amount of vertex data by one-third. We also need to load index data, but these indices are consistent among all viewpoints, such that they can be loaded just once at application startup using an OpenGL extension such as vertex buffer objects.

Ultimately, we reduced the number of floats passed per tetrahedron to 28: 7 floats per vertex (3 for the position, 1 for the scalar value, 1 for the distance to the opposite face, 1 for the scalar value at the opposite face, and 1 identifying the opposite face) times 4 vertices. Compare our required bandwidth to the approach given by Weiler [27] that requires 192 floats per tetrahedron: 16 floats per vertex times 12 vertices.[1]

3.2 Adaptive Sampling

The original implementation of View Independent Cell Projection [25], as well as our implementation of Balanced Cell Projection, use pre-integration [20] to perform the ray integration. However, pre-integration currently limits the classification to 1D transfer functions with some shading. Pre-integration allows neither the opacity to be scaled by the gradient of the scalars nor the highlighting of scalar boundaries based on gradients [9]. Pre-integration is also incapable of supporting multidimensional transfer functions [11, 12], non-photorealistic rendering effects [5, 10], or global illumination [3, 8, 13, 33]. There is also a large body of research dedicated to classification without transfer functions [4], and Tzeng, Lum, and Ma [24] demonstrate how to apply a general N-dimensional classification algorithm to volume rendering. Figure 2 gives a simple example showing the utility of one of these classification schemes. There are many more examples in the cited literature. Consequently, we move away from pre-integration.

We can overcome this limitation by simply passing the classified colors and opacities to the graphics card and performing ray integration there. However, this introduces aliasing. Although it

[1] The vertex information changes per face, so you need to submit the data of each vertex three times, once for each face to which it is connected. Our count assumes that you send the data redundantly to facilitate the use of vertex arrays.

is often appropriate to linearly interpolate the mesh data within the cells, it is, in general, incorrect to apply this interpolation to post-classification colors. As an example, consider Figure 3(a). Because the transfer function is sampled only at cell vertices, the renderer completely misses sharp transitions in the transfer function within cells, leaving a blocky, blurry mess. Compare this result to the appropriate transfer function sampling in Figure 3(b). The aliasing does not occur with pre-integration because the original scalars are being interpolated, not the final colors [6].

Williams, Max, and Stein [31] solve the aliasing problem by splitting cells. They define their transfer functions as piecewise linear functions. Each **control point**, a point where the transfer function is nonlinear, defines an isosurface. They split the cells on these isosurfaces, yielding a linear interpolation of colors within the split cells. The problem with the Williams, Max, and Stein approach is that it introduces a high overhead when the transfer function changes.

Our approach is similar to that of Williams, Max, and Stein in that we split cells with respect to the surfaces implied by sharp changes in the classification. However, instead of splitting a cell geometrically, we clip the cell on the graphics card. When rendering a cell, we allow it to be clipped by up to two parallel planes. Clipped cells are rendered piece-by-piece in back-to-front order.

Ideally, we would like to use the clipping hardware of the graphics card to modify the geometry. Unfortunately, the clipping hardware can clip only polygons and the vertex processor cannot generate the extra vertices necessary to clip tetrahedra. Instead, we perform the clipping on each fragment. The clipping planes are identified by the distance from each vertex along the viewing ray to the plane. These distances are then interpolated by the rasterizer to get the distance from each point in the front face. These distances, plus the already known distance between the front and back faces, are sufficient to locate the clipping planes.

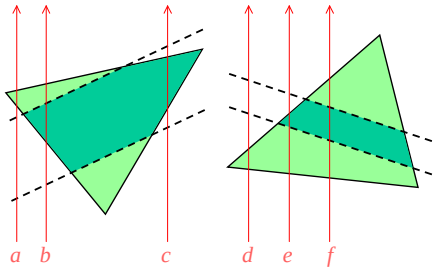


Figure 4: Tetrahedral clipping (reduced to a 2D example). Here we show two example triangles that we clip on a per fragment basis between the two clipping planes. Six example viewing rays (labeled *a*–*f*) are given.

Figure 4 shows examples of clipped viewing-ray segments. If the entire segment falls in between the two planes, as in ray *c*, then we do not clip the segment at all. If the front part of the ray is behind the far plane or the back part of the ray is in front of the near plane, as in rays *a* and *d* respectively, then the segment is discarded. Otherwise, if the front part of the segment is in front of the near plane, as in rays *e* and *f*, we clamp the front scalar value to that of the front plane and appropriately shorten the length of the segment. We likewise clip the cell when the back part of the segment is behind the far plane, as in rays *b* and *f*.

Unfortunately, our method for rendering adaptively sampled, pre-classified volumes imposes extra load in several ways. First, extra data must be passed to the GPU. Replacing scalars with colors and passing distances to cutting planes bumps the data transfer up to 60 floats per tetrahedron. Second, the tetrahedral clipping requires extra fragment program instructions: 18 instructions for the clipping plus any required for the ray integration. Third, clipped cells must be rendered multiple times.

The number of extra cells rendered is dependent on both the mesh and the classification used. For three example meshes, we counted the number of extra cells that need to be rendered for a variety of control points for a 1D transfer function. Figure 5 displays these counts in 1D scatter plots. The number of splits required for a randomly selected control point is on average about 2.5 percent.

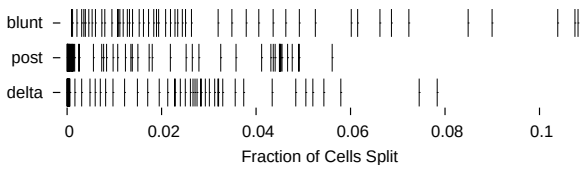
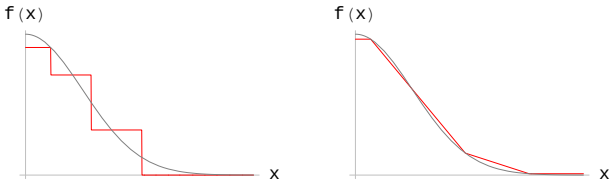


Figure 5: One dimensional scatter plots of the number of cells that would be intersected by a particular isosurface. Points in the plots are stretched into lines for easier viewing.

4 Ray Integration

As noted in Section 2, there are many solutions to the volume rendering integral, although most require a piecewise constant luminance [12, 17, 21]. Williams and Max [30] solve the integral for piecewise linear luminance (Equation 2), but their solution takes far more computation than the others. As is demonstrated in Figure 6, a piecewise linear function is more accurate than a piecewise constant function, even when estimating nonlinear functions. Thus, we propose a novel method of evaluating Equation 2 that requires far less computation than previous approaches [23, 30, 31].



(a) Stair (piecewise constant) function.

(b) Piecewise linear function.

Figure 6: Comparison of normal distribution estimations. The error of the stair function (as measured by the area between the estimation and actual curve) is 3 times that of the piecewise linear function.

We start with the general volume rendering integral, Equation 1. We plug in a linear form for $L(s)$, $L(s) = L_\mathrm{b}(1-\frac{s}{D}) + L_\mathrm{f}\frac{s}{D}$, and then group terms containing the parameters for luminance (L_b and L_f) obtaining

$$I_D = I_0 e^{-\int_0^D \tau(t)dt} + \int_0^D \left(L_\mathrm{b}\left(1-\frac{s}{D}\right) + L_\mathrm{f}\frac{s}{D}\right)\tau(s)e^{-\int_s^D \tau(t)dt}ds$$

$$I_D = I_0 e^{-\int_0^D \tau(t)dt} + L_\mathrm{b}\int_0^D \left(1-\frac{s}{D}\right)\tau(s)e^{-\int_s^D \tau(t)dt}ds + L_\mathrm{f}\int_0^D \frac{s}{D}\tau(s)e^{-\int_s^D \tau(t)dt}ds$$

We can further resolve the integrals through integration by parts.

$$I_D = I_0 e^{-\int_0^D \tau(t)dt} + L_\mathrm{b}\left(\left(1-\frac{s}{D}\right)e^{-\int_s^D \tau(t)dt}\Big|_0^D - \int_0^D -\frac{1}{D}e^{-\int_s^D \tau(t)dt}ds\right) + L_\mathrm{f}\left(\frac{s}{D}e^{-\int_s^D \tau(t)dt}\Big|_0^D - \int_0^D \frac{1}{D}e^{-\int_s^D \tau(t)dt}ds\right)$$

$$I_D = I_0 e^{-\int_0^D \tau(t)dt} + L_\mathrm{b}\left(-e^{-\int_0^D \tau(t)dt} + \frac{1}{D}\int_0^D e^{-\int_s^D \tau(t)dt}ds\right) + L_\mathrm{f}\left(1-\frac{1}{D}\int_0^D e^{-\int_s^D \tau(t)dt}ds\right) \quad (4)$$

There is significant repetition of terms in Equation 4. We define the following two terms, each of which appears twice.

$$\zeta_{D,\tau(t)} \equiv e^{-\int_0^D \tau(t)dt} \quad (5)$$

$$\Psi_{D,\tau(t)} \equiv \frac{1}{D}\int_0^D e^{-\int_s^D \tau(t)dt}ds \quad (6)$$

Substituting Equation 5 and Equation 6 into Equation 4 results in the following.

$$I_D = I_0\zeta_{D,\tau(t)} + L_\mathrm{b}\left(\Psi_{D,\tau(t)} - \zeta_{D,\tau(t)}\right) + L_\mathrm{f}\left(1-\Psi_{D,\tau(t)}\right) \quad (7)$$

Given $\zeta_{D,\tau(t)}$ and $\Psi_{D,\tau(t)}$, Equation 7 is straightforward and fast to compute. We now describe how to evaluate $\zeta_{D,\tau(t)}$ and $\Psi_{D,\tau(t)}$ with linear attenuation.

Solving for $\zeta_{D,\tau(t)}$ is straightforward. Using a linear form for $\tau(t)$, $\tau(t) = \tau_\mathrm{b}(1-\frac{t}{D}) + \tau_\mathrm{f}\frac{t}{D}$, and plugging into Equation 5, we get the following.

$$\begin{aligned}\zeta_{D,\tau_\mathrm{b},\tau_\mathrm{f}} &= e^{-\int_0^D\left(\tau_\mathrm{b}\left(1-\frac{t}{D}\right)+\tau_\mathrm{f}\frac{t}{D}\right)dt} \\ &= e^{-\left(\tau_\mathrm{b}\left(D-\frac{D}{2}\right)+\tau_\mathrm{f}\frac{D}{2}\right)} \\ &= e^{-\frac{D}{2}(\tau_\mathrm{b}+\tau_\mathrm{f})}\end{aligned} \quad (8)$$

Because Equation 8 resolves to such a simple expression, we can compute it directly in the programmable fragment units (of DirectX9-class graphics hardware [15]) with few instructions.

In contrast, using a linear form for $\tau(s)$ does not resolve $\Psi_{D,\tau(s)}$ to a simple, easily computed form.

$$\Psi_{D,\tau_\mathrm{b},\tau_\mathrm{f}} = \frac{1}{D}\int_0^D e^{-\int_s^D\left(\tau_\mathrm{b}\left(1-\frac{t}{D}\right)+\tau_\mathrm{f}\frac{t}{D}\right)dt}ds \quad (9)$$

If we can reduce Ψ to few enough variables, we can store pre-integrated values in a table. Consider what happens when we change the limits of the integrals in Equation 9 to range between 0 and 1.

$$\Psi_{D,\tau_\mathrm{b},\tau_\mathrm{f}} = \int_0^1 e^{-D\int_s^1(\tau_\mathrm{b}(1-t)+\tau_\mathrm{f}t)dt}ds$$

Next, we distribute D within the inner integral.

$$\Psi_{\tau_\mathrm{b}D,\tau_\mathrm{f}D} = \int_0^1 e^{-\int_s^1(\tau_\mathrm{b}D(1-t)+\tau_\mathrm{f}Dt)dt}ds \quad (10)$$

Equation 10 demonstrates that we may store Ψ in a two-dimensional table by pre-computing Ψ for all applicable $(\tau_\mathrm{b}D, \tau_\mathrm{f}D)$ pairs. Before a lookup into this table may occur, we must compute the products $\tau_\mathrm{b}D$ and $\tau_\mathrm{f}D$. We can perform both multiplications in a single fragment-program vector operation, and we can reuse the products to compute ζ if we rewrite Equation 8 as $e^{-\frac{1}{2}(\tau_\mathrm{b}D+\tau_\mathrm{f}D)}$, so the multiplications are essentially free.

However, there is a problem with storing $\Psi_{\tau_\mathrm{b}D,\tau_\mathrm{f}D}$ in a table. The quantities τD are not bound. Furthermore, because cell sizes in unstructured meshes can vary by several orders of magnitude, finding an appropriate finite domain for $\Psi_{\tau_\mathrm{b}D,\tau_\mathrm{f}D}$ is problematic.

We solve this problem by again changing the variables we use to index Ψ. First, we define the variable γ as

$$\gamma \equiv \frac{\tau D}{\tau D+1} \quad (11)$$

We choose γ for its resemblance to $\Psi_{\tau_\mathrm{b}D,0}$. Solving Equation 11 for τD,

$$\tau D = \gamma/(1-\gamma)$$

and substituting into Equation 10, we get the following.

$$\Psi_{\gamma_\mathrm{b},\gamma_\mathrm{f}} = \int_0^1 e^{-\int_s^1\left(\frac{\gamma_\mathrm{b}}{1-\gamma_\mathrm{b}}(1-t)+\frac{\gamma_\mathrm{f}}{1-\gamma_\mathrm{f}}t\right)dt}ds \quad (12)$$

The principle advantage of using γ over τD is that valid values of γ range only over $[0,1)$. It is therefore possible to store the entire domain of Ψ into a single table. Figure 7 shows a plot of $\Psi_{\gamma_\mathrm{b},\gamma_\mathrm{f}}$ over its entire domain.

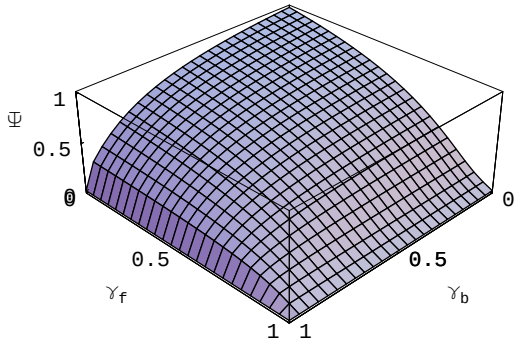


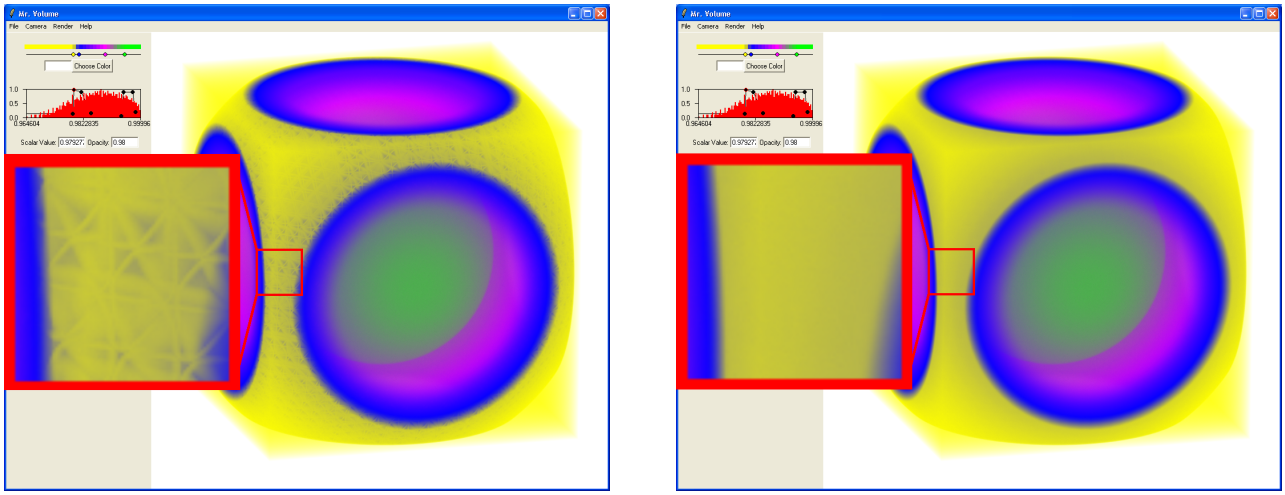
Figure 7: Plot of Ψ against γ_b and γ_f (Equation 12).

Because this method uses a table that holds the pre-integration of part of the volume rendering integral, we dub this method **partial pre-integration**. The major difference between our partial pre-integration and the pre-integration approach introduced by [20] is that values in our table do not rely on a transfer function. Thus, the approach is applicable to any classification system.

The construction of the Ψ table is very compute intensive and can take hours to perform. However, the time to compute the table is inconsequential because the table is ubiquitous. The Ψ table does *not* have to be recomputed for a transfer function change. The Ψ table does *not* have to be recomputed for a change in the mesh being rendered. The same Ψ table may be used for any number of programs.[2]

Figure 8 compares our partial pre-integration method with a ray integration method that uses constant luminance [17, 21]. Partial pre-integration clearly gives superior accuracy.

[2]You can download a Ψ table for your own applications from http://www.cs.unm.edu/~kmorel/documents/volvis2004/.



(a) Average color approximation.

(b) Partial pre-integration.

Figure 8: A comparison of partial pre-integration to previous work. In the image on the left, you can see blue bleeding through the yellow. In the image on the right, no such bleeding occurs. We have magnified a portion of each image.

5 Results

Rendering speeds of a volume rendering system can vary with the volume it is rendering, the classification being used, the viewing projection, and the image size. In this chapter, we give rendering times for several data sets taken from the NASA Advanced Supercomputing website[3] and converted to tetrahedra.

The timings given in this section are for 800 by 800 pixel images unless otherwise specified. For comparative testing purposes, we use 1D transfer functions with 9 control points for classification. The renderings rotate the camera around the center of the model. The frame times given are an average of the rendering speed over every frame through the rotation. We performed the tests on a 3.2 GHz Pentium 4 with 2 GB of RAM and a Quadro FX 3000 graphics card. The Quadro graphics card has 256 MB of its own memory and resides on an AGP 8X bus.

5.1 Cell Projection

Our cell projection for adaptive transfer function sampling clips tetrahedra and renders them multiple times. Before analyzing the rendering rate of this cell projection method, we must first understand how many more tetrahedra are rendered in order to perform the adaptive sampling. Table 1 gives, for each data set, the amount of extra tetrahedra rendered. All the transfer functions selected require the adaptive sampling method to render about 3 to 4 percent more tetrahedra per control point. This number is slightly higher than our estimate in Section 3.2, possibly due to a higher density of cells in interesting parts of the data.

Table 2 compares the cell projection methods introduced in this paper to previously developed projection methods. *Projected Tetrahedra*, *GATOR*, and *View Independent Cell Projection* were introduced by [21], [32], and [25], respectively. *Balanced Cell Projection* and *Adaptive Sampling* are the methods presented in Sections 3.1 and 3.2, respectively. To highlight the running times of each cell projection method, we used the least computationally intensive ray integration methods. For *Projected Tetrahedra*, *GATOR*,

[3] http://www.nas.nasa.gov/Research/Datasets/datasets.html

Table 1: Growth in data sets for adaptive sampling. This table gives the size of the original data set, the number of tetrahedra rendered by the adaptive sampling approach, and the growth in the number of tetrahedra rendered.

Data Set	Tetrahedra in Data Set	Tetrahedra Rendered	Growth
Blunt Fin	187,395	249,278	33%
Oxygen Post	513,375	662,625	29%
Delta Wing	1,005,675	1,373,010	36%

Table 2: Running times for various volume rendering cell projection approaches. Blue methods are previous work whereas green methods are introduced in this paper.

Model	Cell Projection Method	fps	tet/sec
Blunt Fin			
	Projected Tetrahedra	11.935	2236 K
	GATOR	2.618	490 K
	View Independent Cell Projection	3.051	572 K
	Balanced Cell Projection	4.090	766 K
	Adaptive Sampling	1.276	318 K
Oxygen Post			
	Projected Tetrahedra	4.872	2501 K
	GATOR	0.674	346 K
	View Independent Cell Projection	1.325	680 K
	Balanced Cell Projection	2.090	1073 K
	Adaptive Sampling	0.578	383 K
Delta Wing			
	Projected Tetrahedra	2.489	2503 K
	GATOR	0.316	318 K
	View Independent Cell Projection	0.702	706 K
	Balanced Cell Projection	1.396	1404 K
	Adaptive Sampling	0.408	561 K

and *Adaptive Sampling*, we averaged the color as prescribed by [21]. For *View Independent Cell Projection* and *Balanced Cell Projection*, we used pre-integration [20] with a table of size 128 by 128 by 256.

The comparative running times are close to what we expect. Our *Balanced Cell Projection* is a modified version of *View Independent Cell Projection*, and the rendering times suggest that these changes do indeed speed up the rendering. *Adaptive Sampling* is the same as *Balanced Cell Projection* with the added ability to render pre-classified and segmented cells. We expect the added overhead to transfer full volume properties plus the added computation for clipping cells to adversely impact performance, and the data shows that it does.

However, these results also differ somewhat from what we would expect. We would expect the improvements of the *Balanced Cell Projection* over the *View Independent Cell Projection* to be more dramatic. Furthermore, the penalty of the *Adaptive Sampling* is worse than we would expect. Surprisingly, *Projected Tetrahedra* runs faster than any of the other methods we implemented. We believe these results arise from the system being fragment-processing bound. If the fragment processing were the bottleneck, it would diminish improvements in the cell projection. Furthermore, the clipping performed in the *Adaptive Sampling* relies heavily on the fragment processor.

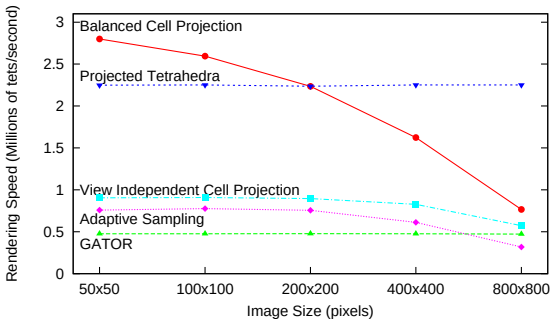


Figure 9: Impact of fragment processing on cell projection. All readings are taken from the Blunt Fin data set.

Figure 9 shows how the rendering rate changes as the image size (and consequently the number of fragments processed) increases. For most of the cell projection methods, the rendering rate holds nearly constant until the image size reaches 40 thousand pixels. After that, the renderer becomes fragment processor bound and the rendering rate steadily decreases as the image size increases. Only *Projected Tetrahedra* and *GATOR* are unaffected by the increased fragments to process. These are also the only two projection methods that require no fragment processing apart from the color computation.

5.2 Ray Integration

Table 3 compares the various methods for computing the volume rendering integral that we discuss and introduce in this paper. Note that we perform the ray integration for all of these methods exclusively in the fragment processor. In addition, recall from the previous section that the renderer is fragment-processing bound for these tests. Therefore, the comparative rates shown in Table 3 are good indicators of the relative performance of the different methods.

The *Average Color and Luminance* approach pioneered by Shirley and Tuchman [21] and Max, Hanrahan, and Crawfis [17] has one of the best frame rates, but, as we have shown, can have large errors due to color averaging. The *Linear Color and Luminance* computation developed by Williams, Max, and Stein [31] has superior image quality but abysmal rendering rates. In contrast, the *Partial Pre-Integration* method (using a 512 by 512 Ψ table) introduced in this paper has a rendering speed competitive with the Shirley and Tuchman method yet is visually indistinguishable from that of the Williams, Max, and Stein method. In fact, the *Partial Pre-Integration* method is an order of magnitude faster than the Williams, Max, and Stein method.

Table 3: Running times for various volume rendering ray integration approaches. Blue methods are previous work whereas green methods are introduced in this paper.

Model	Ray Integration Method	fps	tet/sec
Blunt Fin			
	Average Luminance and Attenuation	1.276	318 K
	Linear Luminance and Attenuation	0.094	22 K
	Partial Pre-Integration	0.867	216 K
Oxygen Post			
	Average Luminance and Attenuation	0.578	383 K
	Linear Luminance and Attenuation	0.044	29 K
	Partial Pre-Integration	0.398	264 K
Delta Wing			
	Average Luminance and Attenuation	0.408	561 K
	Linear Luminance and Attenuation	0.036	49 K
	Partial Pre-Integration	0.301	414 K

6 Conclusions

In this paper, we present partial pre-integration. Partial pre-integration is a volume ray integration implementation that neither exhibits the artifacts generated by the constant luminance approximation [17, 21] nor incurs the heavy computational overhead of calculating the volume rendering integral directly [23, 30, 31].

Our improvements on the cell projection algorithm are successful. We transfer far less data to the GPU per tetrahedron than previous methods of on card projected tetrahedra [25, 32] allowing more efficient streaming of cells to the graphics card than before. However, these improvements are mitigated by slow fragment processing.

Our implementation of tetrahedron clipping on the GPU is somewhat disappointing. We overestimated the speed of our fragment processor and consequently our per fragment clipping contributed to a bottleneck. This bottleneck causes our system to be slower than other current systems. However, as the computation happens entirely on the GPU, we have cycles remaining on the CPU that we can reclaim for cell visibility sorting, which itself is a computationally intensive problem. Furthermore, we expect the fragment processing power of 3D graphics cards to improve significantly in future generations.

Although the vertex and fragment programs on the GPU perform calculations with 32-bit floating-point precision, we place the results in an 8-bit buffer so that we may take advantage of the color blending hardware. Using an 8-bit color buffer does introduce quantization error. We assume that the next generation of 3D graphics cards is capable of performing blending for higher precision color buffers such that we may greatly reduce this quantization error.

Ultimately, as we are fragment processor bound, our method is slower than that of pre-integration [20], which requires little more than a texture lookup. However, unlike pre-integration, our system supports the rendering of data with any classification algorithm. Thus, we are capable of supporting gradient shading, gradient opacity, boundary highlights, silhouettes, multidimensional transfer functions, and any other classification scheme not yet de-

veloped or exploited by the volume rendering community.

7 Acknowledgments

We would like to thank Patricia Crossno, Brian Wylie, Philip Heermann, and Cláudio Silva for their technical and programmatic support. We would also like to thank the NVIDIA Corporation for their hardware donations and technical support.

The DOE Mathematics, Information, and Computer Science Office funded this research. The National Science Foundation also funded this research under grant number CDA-9503064. The work was performed at Sandia National Laboratories and the University of New Mexico. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under Contract DE-AC04-94AL85000.

References

[1] Brian Cabral, Nancy Cam, and Jim Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Proceedings of the 1994 Symposium on Volume Visualization*, pages 91–98, October 1994.

[2] João Comba, James T. Klosowski, Nelson Max, Joseph S. B. Mitchell, Cláudio T. Silva, and Peter L. Williams. Fast polyhedral cell sorting for interactive rendering of unstructured grids. *Computer Graphics Forum (Eurographics '99)*, 18(3):369–376, April/June 1999.

[3] Yoshinori Dobashi, Kazufumi Kaneda, Hideo Yamashita, Tsuyoshi Okita, and Tomoyuki Nishita. A simple, efficient method for realistic animation of clouds. In *Proceedings of ACM SIGGRAPH 2000*, pages 19–28, July 2000.

[4] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification*. Wiley, second edition, 2001. ISBN 0-471-05669-3.

[5] David Ebert and Penny Rheingans. Volume illustration: Non-photorealistic rendering of volume models. In *Proceedings of IEEE Visualization 2000*, pages 195–201, October 2000.

[6] Klaus Engel, Martin Kraus, and Thomas Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In Stephen N. Spencer, editor, *2001 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 9–16, August 2001.

[7] Stefan Guthe, Stefan Röttger, Andreas Schrieber, Wolfgang Straßer, and Thomas Ertl. High-quality unstructured volume rendering on the PC platform. In *Proceedings of the SIGGRAPH/Eurographics Graphics Hardware Workshop '02*, pages 119–125, 2002.

[8] Mark J. Harris and Anselmo Lastra. Real-time cloud rendering. *Compute Graphics Forum (Eurographics 2001 Proceedings)*, 20(3):76–84, September 2001.

[9] Gordon Kindlmann and James W. Durkin. Semi-automatic generation of transfer functions for direct volume rendering. In *Proceedings of the 1998 IEEE Symposium on Volume Visualization*, pages 79–86, October 1998.

[10] Gordon Kindlmann, Ross Whitaker, Tolga Tasdizen, and Torsten Möller. Curvature-based transfer functions for direct volume rendering: Methods and applications. In *Proceedings of IEEE Visualization 2003*, October 2003.

[11] Joe Kniss, Gordon Kindlmann, and Charles Hansen. Multidimensional transfer functions for interactive volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):270–285, July/September 2002.

[12] Joe Kniss, Simon Premože, Milan Ikits, Aaron Lefohn, Charles Hansen, and Emil Praun. Gaussian transfer functions for multi-field volume visualization. In *Proceedings of IEEE Visualization 2003*, pages 497–504, October 2003.

[13] Joe Kniss, Simon Premoze, Charles Hansen, Peter Shirley, and Allen McPherson. A model for volume lighting and modeling. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):150–162, April/June 2003.

[14] Shankar Krishnan, Cláudio T. Silva, and Bin Wei. A hardware-assisted visibility-ordering algorithm with applications to volume rendering. In *Proceedings of Data Visualization (Eurographics/IEEE Symposium on Visualization)*, pages 233–242, 2001.

[15] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: A system for programming graphics hardware in a C-like language. *ACM Transactions on Graphics (ACM SIGGRAPH 2003)*, 22(3):896–907, July 2003.

[16] Nelson Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, June 1995.

[17] Nelson Max, Pat Hanrahan, and Roger Crawfis. Area and volume coherence for efficient visualization of 3D scalar functions. In *Computer Graphics (Proceedings of the San Diego Workshop on Volume Visualization)*, volume 24, pages 27–33, December 1990.

[18] Michael Meißner, Stefan Guthe, and Wolfgang Straßer. Interactive lighting models and pre-integration for volume rendering on PC graphics accelerators. In *Proceedings of Graphics Hardware 2002*, pages 209–218, May 2002.

[19] Stefan Röttger and Thomas Ertl. A two-step approach for interactive pre-integrated volume rendering of unstructured grids. In *Proceedings of IEEE Volume Visualization and Graphics Symposium 2002*, pages 23–28, October 2002.

[20] Stefan Röttger, Martin Kraus, and Thomas Ertl. Hardware-accelerated volume and isosurface rendering based on cell-projection. In *Proceedings of IEEE Visualization 2000*, pages 109–116, October 2000.

[21] Peter Shirley and Allan Tuchman. A polygonal approximation to direct scalar volume rendering. volume 24, pages 63–70, December 1990.

[22] Cláudio T. Silva, Joseph S. B. Mitchell, and Peter L. Williams. An exact interactive time visibility ordering algorithm for polyhedral cell complexes. In *IEEE Symposium on Volume Visualization*, pages 87–94, 1998.

[23] Clifford Stein, Barry Becker, and Nelson Max. Sorting and hardware assisted rendering for volume visualization. In *Proceedings of the 1994 Symposium on Volume Visualization*, pages 83–89, October 1994.

[24] Fan-Yin Tzeng, Eric B. Lum, and Kwan-Liu Ma. A novel interface for higher-dimensional classification of volume data. In *Proceedings of IEEE Visualization 2003*, pages 505–512, October 2003.

[25] Manfred Weiler, Martin Kraus, and Thomas Ertl. Hardware-based view-independent cell projection. In *Proceedings of IEEE Volume Visualization and Graphics Symposium 2002*, pages 13–22, October 2002.

[26] Manfred Weiler, Martin Kraus, Markus Merz, and Thomas Ertl. Hardware-based ray casting for tetrahedral meshes. In *Proceedings of IEEE Visualization 2003*, pages 333–340, October 2003.

[27] Manfred Weiler, Martin Kraus, Markus Merz, and Thomas Ertl. Hardware-based view-independent cell projection. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):163–175, April/June 2003.

[28] Jane Wilhelms and Allen van Gelder. A coherent projection approach for direct volume rendering. In *Computer Graphics (Proceedings of ACM SIGGRAPH 91)*, volume 25, pages 275–284, July 1991.

[29] P. L. Williams. Visibility ordering meshed polyhedra. *ACM Transactions on Graphics*, 11(2):103–126, April 1992.

[30] Peter Williams and Nelson Max. A volume density optical model. In *Computer Graphics (Proceedings of the 1992 Workshop on Volume Visualization)*, pages 61–68, October 1992.

[31] Peter L. Williams, Nelson L. Max, and Clifford M. Stein. A high accuracy volume renderer for unstructured data. *IEEE Transactions on Visualization and Computer Graphics*, 4(1), March 1998.

[32] Brian Wylie, Kenneth Moreland, Lee Ann Fisk, and Patricia Crossno. Tetrahedral projection using vertex shaders. In *Proceedings of IEEE Volume Visualization and Graphics Symposium 2002*, pages 7–12, October 2002.

[33] Caixia Zhang and Roger Crawfis. Shadows and soft shadows with participating media using splatting. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):139–149, April/June 2003.