# Projecting Tetrahedra without Rendering Artifacts

Martin Kraus* Purdue University
Wei Qiao† Purdue University
David S. Ebert‡ Purdue University

**ABSTRACT**

Hardware-accelerated direct volume rendering of unstructured volumetric meshes is often based on tetrahedral cell projection, in particular, the Projected Tetrahedra (PT) algorithm and its variants. Unfortunately, even implementations of the most advanced variants of the PT algorithm are very prone to rendering artifacts.

In this work, we identify linear interpolation in screen coordinates as a cause for significant rendering artifacts and implement the correct perspective interpolation for the PT algorithm with programmable graphics hardware. We also demonstrate how to use features of modern graphics hardware to improve the accuracy of the coloring of individual tetrahedra and the compositing of the resulting colors, in particular, by employing a logarithmic scale for the pre-integrated color lookup table, using textures with high color resolution, rendering to floating-point color buffers, and alpha dithering. Combined with a correct visibility ordering, these techniques result in the first implementation of the PT algorithm without objectionable rendering artifacts. Apart from the important improvement in rendering quality, our approach also provides a test bed for different implementations of the PT algorithm that allows us to study the particular rendering artifacts introduced by these variants.

**CR Categories:** I.3.3 [Computer Graphics]: Picture/Image Generation—Display algorithms, Bitmap and framebuffer operations; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

**Keywords:** volume visualization, volume rendering, cell projection, projected tetrahedra, perspective interpolation, dithering, programmable graphics hardware

## 1 INTRODUCTION AND PREVIOUS WORK

There are several approaches to direct volume rendering of unstructured meshes, e.g., ray casting and cell projection. However, most implementations for OpenGL graphics hardware are based on cell projection; more specifically, the Projected Tetrahedra (PT) algorithm published by Shirley and Tuchman [18].

The PT algorithm exploits the triangle rasterization performance of graphics hardware by decomposing the projected silhouette of a tetrahedron into three or four triangles, as in Figure 1. Previous research has focused on two aspects of this algorithm: the color computation for these triangles [6, 14, 16, 19] and the computation of a visibility ordering of the tetrahedra [2, 3, 10]. In this work, we are not concerned with the latter and assume that a correct visibility ordering is available. In fact, we employ the extension of the meshed polyhedra visibility ordering (MPVO) algorithm for non-convex meshes suggested by Williams [22], which computes a correct ordering for most tetrahedral meshes.

*e-mail: kraus@purdue.edu
†e-mail: qiaow@purdue.edu
‡e-mail: ebertd@purdue.edu

IEEE Visualization 2004
October 10-15, Austin, Texas, USA
0-7803-8788-0/04/$20.00 ©2004 IEEE

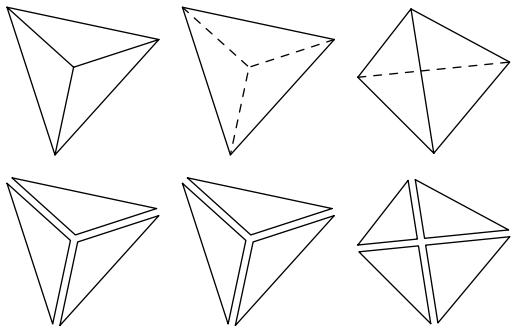


Figure 1: Classification of non-degenerate projected tetrahedra (top row) and the corresponding decompositions (bottom row).

To compute the colors of triangles generated by the PT algorithm, Shirley and Tuchman [18] suggested computing correct colors only for the triangles' vertices. Thus, the efficient linear interpolation of vertex colors provided by graphics hardware can be exploited. Unfortunately, this results in rendering artifacts even for uniform attenuation coefficients as noted by Stein, Becker, and Max [19]. Their solution was to interpolate the thickness and the average attenuation coefficient of the projected tetrahedron for each fragment by means of texture coordinate interpolation. Based on these two interpolated values, the opacity of each fragment is determined by a two-dimensional texture lookup. This opacity is either modulated with a constant color, or with an interpolated vertex color. Unfortunately, the linear interpolation of the average attenuation coefficient and the thickness is only correct for orthogonal projections and leads to rendering artifacts for perspective projections.

An efficient computation of the correct perspective interpolation has been published by Heckbert and Moreton [7] and independently by Blinn [1]. Moreover, correct perspective interpolation is usually offered by modern graphics hardware supporting OpenGL. However, the perspective interpolation employed in OpenGL [17] cannot directly cure this problem for the PT algorithm because it was designed for an interpolation of values on triangles—not within tetrahedra.

Another well-known disadvantage of the method by Stein et al. [19] is the restriction of the attenuation coefficients to linear functions within each tetrahedron. This problem was addressed by a software-based computation for arbitrary attenuation transfer functions published by Max, Hanrahan and Crawfis [13]. The availability of hardware-supported three-dimensional texture maps allowed Röttger, Kraus, and Ertl [16] to implement a generalization of this method in graphics hardware. This technique is now known as pre-integrated cell projection. The basic idea is to employ linear interpolation of texture coordinates to interpolate the thickness of the tetrahedron and the scalar data value on the front and back faces of the tetrahedron for each fragment. Hardware-accelerated three-dimensional texture mapping is then exploited to perform a lookup of the color for a particular fragment. Note that this particu-

lar implementation of pre-integrated cell projection by Röttger et al. is also restricted to orthogonal projections and generates rendering artifacts for perspective projections. Pre-integrated cell projection was further improved [6, 14]; however, the artifacts caused by perspective projections were never addressed.

An alternative to cell projection is ray casting in unstructured meshes. The basic algorithm for traversing cells of a tetrahedral mesh along viewing rays and an implementation in software were published by Garrity [5]. More recently, Weiler et al. [20, 21] published a cell projection algorithm based on the idea of ray casting single tetrahedra in graphics hardware and a hardware-based implementation of a pre-integrated variant of Garrity's algorithm using floating-point color buffers. With respect to rendering artifacts, there are two advantages of the ray casting approach in contrast to the PT algorithm: the use of floating-point precision to composite colors and the absence of any interpolation errors due to perspective projection. In order to achieve a similar rendering quality with the PT algorithm, we derive the correct perspective interpolation for projected tetrahedra and discuss its implementation with the help of programmable graphics hardware in Section 2.

For the coloring of individual tetrahedra, Weiler et al. [20] employed a pre-integrated lookup table implemented by a three-dimensional floating-point RGBA texture. Unfortunately, the lack of trilinear interpolation in these textures and the limited resolution result in rendering artifacts. Our solution to these problems is the use of textures with 16 bits per color component and the implementation of a logarithmic scale for the pre-integrated lookup table. These enhancements are described in Section 3.

While the compositing of color contributions is performed with floating-point precision by most ray casters, hardware-accelerated cell projection with programmable graphics hardware has been restricted to 8-bit fixed-point color components until recently. Even the currently available hardware support for floating-point color buffers does not allow us to blend small, overlapping triangle primitives without rendering artifacts. Since the PT algorithm tends to generate many small triangles, it is prone to these artifacts. In Section 4, we show how to avoid them at the cost of rendering performance. Moreover, we present an alternative approach for 8-bit color components similar to the alpha dithering technique suggested by Williams, Frank, and LaMar [23, 11].

A comparison of the common rendering artifacts of implementations of the PT algorithm is given in Section 5. The rendering performance of our enhanced variants of the PT algorithm are also discussed in this section. Section 6 presents our conclusions and plans for future work.

## 2 Perspective Interpolation

Before discussing the interpolation of vertex attributes with perspective correction for the PT algorithm in Section 2.2, we will introduce our notation and derive the required equations.

### 2.1 Interpolation in Normalized Device Coordinates

Our notation is based on the OpenGL specification [17]; in particular, the homogeneous coordinates of a vertex $\mathbf{v}_o$ in object space are denoted by $(x_o, y_o, z_o, w_o)^T$. Assuming $w_o$ is not equal to 0, this four-dimensional vector represents a three-dimensional vector $(x_o/w_o, y_o/w_o, z_o/w_o)^T$. The model-view matrix $M$ maps a vector $\mathbf{v}_o$ from object space to a vector $\mathbf{v}_e = (x_e, y_e, z_e, w_e)^T$ in eye space, i.e., $\mathbf{v}_e = M\mathbf{v}_o$. The mapping from eye space to clip space is performed by the projection matrix $P$: $\mathbf{v}_c = P\mathbf{v}_e$ with $\mathbf{v}_c = (x_c, y_c, z_c, w_c)^T$. Finally, we define the normalized device coordinates as the components of the four-dimensional vector $\mathbf{v}_d = (x_d, y_d, z_d, 1)^T$ obtained by a "perspective division" from $\mathbf{v}_c$, i.e, $\mathbf{v}_d = \mathbf{v}_c/w_c$.

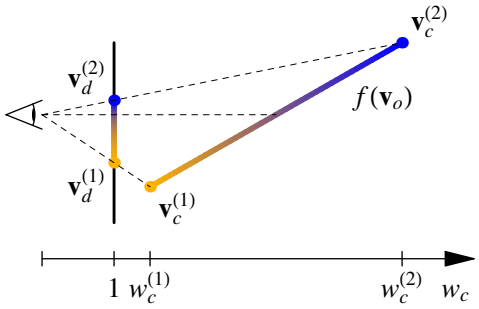



Figure 2: The projection of a linear function $f(\mathbf{v}_o)$ onto the view plane results in a nonlinear function. In the depicted case of a linear color interpolation between vertices $\mathbf{v}_c^{(1)}$ and $\mathbf{v}_c^{(2)}$, the center point between $\mathbf{v}_c^{(1)}$ and $\mathbf{v}_c^{(2)}$ and its color are not projected to the center point between $\mathbf{v}_d^{(1)}$ and $\mathbf{v}_d^{(2)}$ as illustrated by the centered dashed line.

For the derivation of the perspective interpolation[1], we have to consider linear functions in object space. In homogeneous coordinates, any scalar function $f(\mathbf{v}_o)$ that is linear in the three-dimensional coordinates $x_o/w_o$, $y_o/w_o$, and $z_o/w_o$ is of the form

$$f(\mathbf{v}_o) = c_x\frac{x_o}{w_o} + c_y\frac{y_o}{w_o} + c_z\frac{z_o}{w_o} + c_w = \mathbf{c}\cdot\frac{\mathbf{v}_o}{w_o}$$

with a constant four-dimensional vector $\mathbf{c} = (c_x, c_y, c_z, c_w)^T$. Note that the projection to normalized device coordinates will turn this linear function into a nonlinear function as illustrated in Figure 2.

Assuming $w_c$ is not equal to 0 and the matrix product $PM$ has an inverse, we can also write:

$$f(\mathbf{v}_o) = \mathbf{c}\cdot(PM)^{-1}PM\frac{\mathbf{v}_o}{w_c}\frac{w_c}{w_o}.$$

With the constant vector $\mathbf{c}' = \left((PM)^{-1}\right)^T\mathbf{c}$ and the equality $PM\mathbf{v}_o/w_c = \mathbf{v}_d$ we obtain

$$f(\mathbf{v}_o)\frac{w_o}{w_c} = \mathbf{c}'\cdot\mathbf{v}_d.$$

This equation implies that $f(\mathbf{v}_o)w_o/w_c$ is a linear function of the normalized device coordinates $x_d$, $y_d$, and $z_d$. Therefore, given any function $f(\mathbf{v}_o)$ that is linear in the three-dimensional coordinates $x_o/w_o$, $y_o/w_o$, and $z_o/w_o$, we may linearly interpolate values of $f(\mathbf{v}_o)w_o/w_c$ in normalized device coordinates.

An important example is $f(\mathbf{v}_o) \equiv 1$. Since 1 is constant, it is also a linear "function" of $x_o/w_o$, $y_o/w_o$, and $z_o/w_o$. In this particular case, the result from above implies that $w_o/w_c$ is a linear function of $x_d$, $y_d$, and $z_d$; therefore, we may linearly interpolate values of $w_o/w_c$ in normalized device coordinates.

These results were exploited by Heckbert and Moreton [7] and independently by Blinn [1] for the perspective interpolation of attributes between vertices, e.g., color or texture coordinates. If two vertices are connected by a line, the corresponding attributes of the two vertices define a linear function on the line segment. Analogously, if three vertices are connected by a triangle, the corresponding attributes define a linear function on the triangle. The domain

[1] Compare Equations 3.4 and 3.6 in the OpenGL 1.5 specification [17].

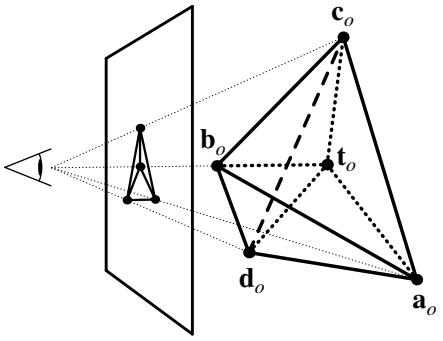



Figure 3: Decomposition of the tetrahedron $(\mathbf{a}_o, \mathbf{b}_o, \mathbf{c}_o, \mathbf{d}_o)$ into three smaller tetrahedra corresponding to the triangles generated by the PT algorithm for class 1a (see Figure 1). The new vertex $\mathbf{t}_o$ is determined by the intersection of triangle $(\mathbf{a}_o, \mathbf{c}_o, \mathbf{d}_o)$ with the extension of the line from the eye point to $\mathbf{b}_o$. The three tetrahedra are $(\mathbf{a}_o, \mathbf{b}_o, \mathbf{c}_o, \mathbf{t}_o)$, $(\mathbf{c}_o, \mathbf{b}_o, \mathbf{d}_o, \mathbf{t}_o)$, and $(\mathbf{d}_o, \mathbf{b}_o, \mathbf{a}_o, \mathbf{t}_o)$.

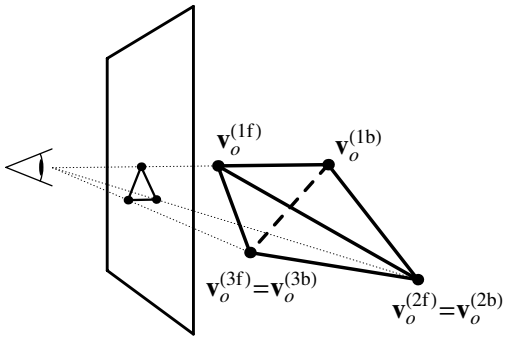



Figure 4: The tetrahedron $(\mathbf{d}_o, \mathbf{b}_o, \mathbf{a}_o, \mathbf{t}_o)$ from Figure 3 in our notation for triangle vertices. Note that $\mathbf{v}_o^{(1f)}$ and $\mathbf{v}_o^{(1b)}$ are projected to the same point on the view plane.

of these linear functions $f(\mathbf{v}_o)$ may be extended to the whole three-dimensional object space without any complications. The projection of these functions is, however, not linear in normalized device coordinates (see Figure 2). Therefore, instead of linearly interpolating vertex attributes, the values of $f(\mathbf{v}_o)w_o/w_c$ and $w_o/w_c$ are interpolated separately for each vertex, and these values are interpolated linearly in normalized device coordinates for each fragment. The value of $f(\mathbf{v}_o)$ is then reconstructed by one division per fragment:

$$f(\mathbf{v}_o) = \frac{\left(f(\mathbf{v}_o)\frac{w_o}{w_c}\right)_{\text{interpolated}}}{\left(\frac{w_o}{w_c}\right)_{\text{interpolated}}}.$$

Assuming $w_o$ is equal to 1 (or at least all $w_o$'s are the same for all vertices), this leads directly to Equations 3.4 and 3.6 in the OpenGL 1.5 specification [17].

### 2.2 Interpolation for Projected Tetrahedra

The original PT algorithm [18] decomposes the (non-degenerate) projection of a tetrahedron into three or four triangles (see Figure 1), computes colors for all triangle vertices, and employs hardware-accelerated triangle rasterization to interpolate the fragment color. Note that a correct perspective interpolation of colors is impossible for this algorithm because each rasterized fragment corresponds to a viewing ray segment through the tetrahedron with a range of $w_c$ coordinates instead of a single $w_c$ coordinate.

Instead of decomposing the projection of a tetrahedron into three or four triangles, we can also decompose the tetrahedron itself into three or four smaller tetrahedra. In this case, the decomposition is performed in object coordinates instead of normalized device coordinates. As illustrated in Figure 3, each of these smaller tetrahedra is projected to one of the triangles of the original PT decomposition. Moreover, each of the smaller tetrahedra features one pair of vertices, which are projected to the same two-dimensional vertex in the view plane. In Figure 4, for example, $\mathbf{v}_o^{(1f)}$ (the "front" vertex) and $\mathbf{v}_o^{(1b)}$ (the "back" vertex) are projected to the same vertex in the view plane. In order to process all three vertices of the triangles in the view plane in exactly the same way (as required by our implementation), we duplicate the other two three-dimensional vertices of the tetrahedron. For example, $\mathbf{v}_o^{(2f)}$ and $\mathbf{v}_o^{(2b)}$ in Figure 4 are identical copies of one vertex. Analogously, $\mathbf{v}_o^{(3f)}$ and $\mathbf{v}_o^{(3b)}$ are identical.

Once the six vertices $\mathbf{v}_o^{(1f)}$, $\mathbf{v}_o^{(1b)}$, $\mathbf{v}_o^{(2f)}$, $\mathbf{v}_o^{(2b)}$, $\mathbf{v}_o^{(3f)}$, and $\mathbf{v}_o^{(3b)}$ are computed, either the front facing triangle spanned by the vertices $\mathbf{v}_o^{(1f)}$, $\mathbf{v}_o^{(2f)}$, and $\mathbf{v}_o^{(3f)}$ or the back facing triangle spanned by the vertices $\mathbf{v}_o^{(1b)}$, $\mathbf{v}_o^{(2b)}$, and $\mathbf{v}_o^{(3b)}$ can be rasterized because both triangles will cover the same pixels. During the rasterization of either triangle, we can interpolate any vertex attribute either on the front facing triangle by interpolating between vertices $\mathbf{v}_o^{(1f)}$, $\mathbf{v}_o^{(2f)}$, and $\mathbf{v}_o^{(3f)}$, or on the back facing triangle by interpolating (with the same weights) between vertices $\mathbf{v}_o^{(1b)}$, $\mathbf{v}_o^{(2b)}$, and $\mathbf{v}_o^{(3b)}$. The correct perspective interpolation can be performed as described in Section 2.1.

This approach avoids rendering artifacts caused by an incorrect (nonperspective) interpolation in several important variants of the PT algorithm. For example, in the variant of the PT algorithm published by Stein et al. [19], we can interpolate the attenuation coefficients $\tau^{(\mathrm{f})}$ and $\tau^{(\mathrm{b})}$ with perspective correction on the front facing and the back facing triangle, respectively, while rasterizing either one. Similarly, we can interpolate the scalar data values $s^{(\mathrm{f})}$ and $s^{(\mathrm{b})}$ on the two triangles in the pre-integrated variant of the PT algorithm suggested by Röttger et al. [16].

These two variants of the PT algorithm also require the thickness of the tetrahedron for the rasterized fragment. This thickness $l$ may be computed as the distance (in three-dimensional eye space) between the points on the two triangles corresponding to the rasterized fragment:

$$l = \left| \frac{\mathbf{v}_e^{(\mathrm{b})}}{w_e^{(\mathrm{b})}} - \frac{\mathbf{v}_e^{(\mathrm{f})}}{w_e^{(\mathrm{f})}} \right|.$$

All coordinates of $\mathbf{v}_e^{(\mathrm{f})}/w_e^{(\mathrm{f})}$ and $\mathbf{v}_e^{(\mathrm{b})}/w_e^{(\mathrm{b})}$ are linear functions of the three-dimensional object coordinates for any model-view matrix $M$ with a fourth row vector of $(0,0,0,1)^T$; therefore, $\mathbf{v}_e^{(\mathrm{f})}/w_e^{(\mathrm{f})}$ and $\mathbf{v}_e^{(\mathrm{b})}/w_e^{(\mathrm{b})}$ can be interpolated with perspective correction on the front facing and back facing triangle, respectively, as described above.

Usually all $w_e$ coordinates are 1. Furthermore, we can exploit the fact that the three-dimensional points represented by $\mathbf{v}_e^{(\mathrm{f})}$ and $\mathbf{v}_e^{(\mathrm{b})}$ are on one line with the origin:

$$l = \left| \mathbf{v}_e^{(\mathrm{b})} - \mathbf{v}_e^{(\mathrm{f})} \right| = \left| \mathbf{v}_e^{(\mathrm{f})} \right| \left| \frac{z_e^{(\mathrm{b})} - z_e^{(\mathrm{f})}}{z_e^{(\mathrm{f})}} \right|.$$

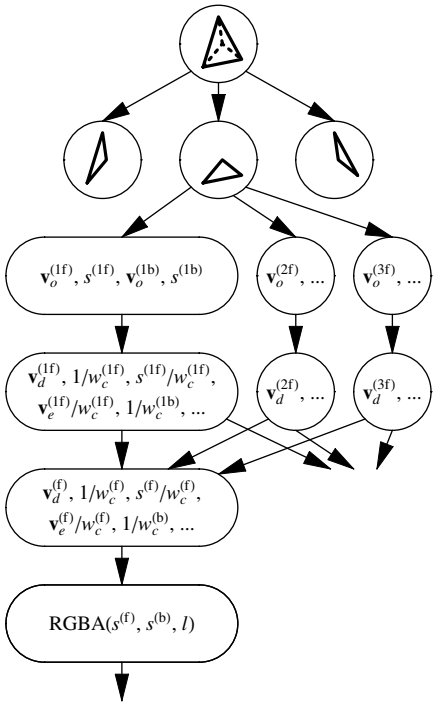



Figure 5: Data flow in our implementation of perspective interpolation for the PT algorithm.

Note that $|\mathbf{v}_e^{(f)}|$ denotes the Euclidean vector norm of the three-dimensional eye space vector represented by $\mathbf{v}_e^{(f)}$. The latter equation is also employed in our implementation, which is presented in the next section.

### 2.3 Implementation with Programmable Graphics Hardware

Our implementation is based on the OpenGL 1.5 ARB extensions for vertex programs and fragment programs [17]. Figure 5 illustrates the data flow for perspective interpolation in our implementation of the PT algorithm for pre-integrated cell projection.

As discussed in Section 2.2, each tetrahedron is decomposed into three or four smaller tetrahedra corresponding to the triangles of the original PT algorithm. For each of the smaller tetrahedra, the six vertices $\mathbf{v}_o^{(1f)}$, $\mathbf{v}_o^{(1b)}$, $\mathbf{v}_o^{(2f)}$, $\mathbf{v}_o^{(2b)}$, $\mathbf{v}_o^{(3f)}$, and $\mathbf{v}_o^{(3b)}$ are computed. Moreover, six corresponding scalar data values $s^{(1f)}$, $s^{(1b)}$, $s^{(2f)}$, $s^{(2b)}$, $s^{(3f)}$, and $s^{(3b)}$ are determined. We rasterize the front facing triangle spanned by the vertices $\mathbf{v}_o^{(1f)}$, $\mathbf{v}_o^{(2f)}$ and $\mathbf{v}_o^{(3f)}$. Apart from these coordinates, each triangle vertex is also provided with the object coordinates of the corresponding "back" vertex and the scalar data values for the front and back vertex. For example, for the $i$-th vertex with object coordinates $\mathbf{v}_o^{(if)}$ we specify the scalar data values $s^{(if)}$ and $s^{(ib)}$, and the vector $\mathbf{v}_o^{(ib)}$ as additional vertex attributes.

These vertex attributes are the input parameters for our vertex program. For the $i$-th vertex, the vertex program computes clip coordinates $\mathbf{v}_c^{(if)} = PM\mathbf{v}_o^{(if)}$ and normalized device coordinates $\mathbf{v}_d^{(if)} = \mathbf{v}_c^{(if)}/w_c^{(if)}$. By performing the perspective division and returning normalized device coordinates instead of clip coordinates, we ensure that OpenGL does *not* use perspective interpolation but linear interpolation since we specify that $w_c$ is equal to 1.

Apart from the projected position, the output parameters of our vertex program for the $i$-th vertex are $1/w_c^{(if)}$, $s^{(if)}/w_c^{(if)}$, $\mathbf{v}_e^{(if)}/w_c^{(if)}$, $1/w_c^{(ib)}$, $s^{(ib)}/w_c^{(ib)}$, and $z_e^{(ib)}/w_c^{(ib)}$. Note that we set all $w_o$'s to 1; thus, they do not appear in these quantities.

These output vertex attributes are linearly interpolated between the three vertices of each triangle. We denote the results of these interpolations by $1/w_c^{(f)}$, $s^{(f)}/w_c^{(f)}$, $\mathbf{v}_e^{(f)}/w_c^{(f)}$, $1/w_c^{(b)}$, $s^{(b)}/w_c^{(b)}$, and $z_e^{(b)}/w_c^{(b)}$, respectively. These are input parameters for our fragment program, which completes the perspective interpolation by dividing interpolated values:

$$s^{(f)} = \frac{s^{(f)}/w_c^{(f)}}{1/w_c^{(f)}}, \mathbf{v}_e^{(f)} = \frac{\mathbf{v}_e^{(f)}/w_c^{(f)}}{1/w_c^{(f)}}, s^{(b)} = \frac{s^{(b)}/w_c^{(b)}}{1/w_c^{(b)}}, z_e^{(b)} = \frac{z_e^{(b)}/w_c^{(b)}}{1/w_c^{(b)}}.$$

The thickness $l$ of the tetrahedron is computed from these quantities as described in Section 2.2. Note that it is often possible to simplify the computation of $l$ without introducing visible rendering artifacts with the help of the approximation $l \approx |z_e^{(b)} - z_e^{(f)}|$.

This completes the perspective interpolation of $s^{(f)}$, $s^{(b)}$, and the computation of $l$. Based on these parameters, the color of the fragment is determined by the fragment program as discussed in the next section.

## 3 Accurate Coloring

In the PT algorithm, many tetrahedra can contribute to the color of a single pixel; therefore, even small color contributions of individual tetrahedra can sum up to a significant contribution to the final image. Thus, in order to avoid rendering artifacts, the accuracy of the color computation for a single tetrahedron has to exceed the color accuracy of the final image.

Fortunately, arithmetic computations in fragment programs may be performed with floating-point precision. Therefore, it is beneficial to replace, for example, the texture lookup for the correct exponential attentuation suggested by Stein et al. [19] by a more accurate computation in a fragment program. For some optical models, the three-dimensional texture lookup for pre-integrated cell projection can also be replaced by a fragment program as suggested by Guthe et al. [6]. In general, however, the pre-integrated lookup can only be replaced by an expensive numerical integration.

Therefore, instead of replacing the pre-integrated texture lookup, we improve its accuracy by employing textures with 16-bit color components (so called "HILO textures" [9]), which support trilinear interpolation. Since HILO textures are only available with two color components, we have to split the RGBA lookup texture into two HILO textures and perform two texture lookups to get all four color components. Note that we avoid floating-point textures because they only permit nearest-neighbor "interpolation."

Apart from the accuracy of the colors tabulated in the three-dimensional lookup texture, we also have to consider its minimum dimensions. The three coordinates for this texture lookup correspond to the scalar data value at the front $s^{(f)}$, the scalar data value at the back $s^{(b)}$, and the thickness $l$ [16]. The texture coordinates are usually computed by a linear mapping of the whole range of $s^{(f)}$, $s^{(b)}$, and $l$, respectively, to the range of texture coordinates. In our implementation, the range of a particular texture coordinate is $[(2n)^{-1}, 1-(2n)^{-1}]$, where $n$ is the dimension of the texture in the corresponding direction. Note that $(2n)^{-1}$ and $1-(2n)^{-1}$ specify the coordinates of the centers of the 0-th and the $(n-1)$-th texel, respectively, in OpenGL textures.

For the dimensions corresponding to $s^{(f)}$ and $s^{(b)}$ a resolution corresponding to the Nyquist sampling rate of the transfer functions will avoid most visible artifacts. In contrast to the scalar data values, the range of the thickness $l$ depends on the tetrahedral mesh. More specifically, the minimum thickness is zero and the maximum

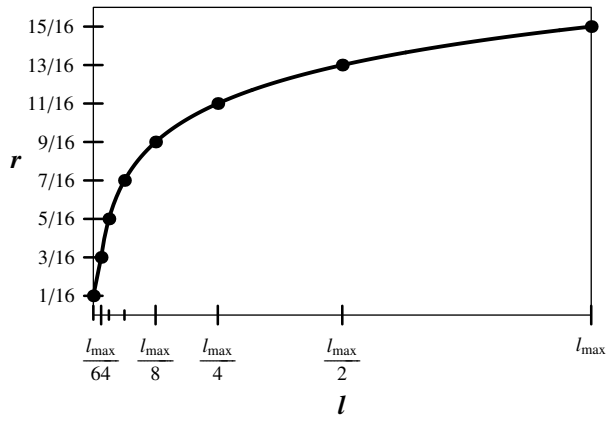



Figure 6: Mapping of $l \in [0, l_{\max}]$ to texture coordinate $r \in [(2n_r)^{-1}, 1-(2n_r)^{-1}]$ for dimension $n_r = 8$. Note that the mapping is linear instead of logarithmic for $0 \le l \le 2^{-(n_r-2)} l_{\max} = l_{\max}/64$.

thickness is equal to the length $l_{\max}$ of the longest edge of all cells in eye space. If the model-view matrix does not include any scaling, $l_{\max}$ is also the length of the longest edge in object space. Note that fragments on the silhouette of a tetrahedral cell usually require a thickness very close to zero; thus, the pre-integrated lookup table has to cover the whole range $[0, l_{\max}]$. The dependency of the color on the thickness $l$ is in general a very smooth function since the most relevant dependencies are the exponential attentuation and the linear accumulation of emitted light. Therefore, previous implementations of pre-integrated cell projection have often chosen a rather low resolution for the thickness.

However, this is not appropriate for data sets featuring a very high ratio between the lengths of the longest and the shortest edges of the mesh. In these cases, the thickness of many small tetrahedra will be close to zero. Therefore, on the one hand, these data sets require a very high resolution of the lookup table for small values of $l$; on the other hand, the thickness is still a smooth function for large values of $l$; thus, a coarse resolution for large $l$ is sufficient.

Therefore, a logarithmic mapping of the thickness $l$ to the texture coordinate $r$ is more appropriate than a linear mapping. Note that a thickness of $l = 0$ should be mapped to the texture coordinate of the 0-th texel $r = (2n_r)^{-1}$ where $n_r$ is the $r$-dimension of the texture. In order to satisfy these constraints, we chose a mapping of $l$ to $r$, which is linear for $0 \le l \le \Delta l$ with $\Delta l = 2^{-(n_r-2)} l_{\max}$ and logarithmic for $l \ge \Delta l$; see Figure 6 and Equation 1.

$$r = \left( \max\left( \log_2 \frac{l}{\Delta l}, 0 \right) + \min\left( \frac{l}{\Delta l}, 1 \right) \right) \Big/ n_r + \frac{1}{2n_r} \qquad (1)$$

A straightforward implementation of Equation 1 using the OpenGL ARB extension for fragment programs requires seven instructions. (Implementing this mapping by means of a 1D texture lookup would usually require a prohibitively large texture size.) To compute the pre-integrated lookup table efficiently, we employ an adapted variant of the incremental pre-integration method [20] that incrementally computes two-dimensional slices of the table for $l = 0, \Delta l, 2\Delta l, 4\Delta l, 8\Delta l, \ldots, 2^{n_r-2}\Delta l$.

## 4 Accurate Compositing

In order to avoid rendering artifacts in volume rendering algorithms, colors have to be composited with a higher accuracy than most graphics adapters offer for frame buffers. Unfortunately, the current hardware support for floating-point color buffers has some crucial limitations, which prevent their efficient use in the context of the PT algorithm as discussed in detail in Section 4.1. Section 4.2 describes an approach based on randomized dithering using frame buffers with 8-bit color components to improve the quality of the final image.

### 4.1 Floating-Point Color Buffer

Current graphics adapters (based on NVIDIA's NV3x and ATI's R3xx chipsets) allow fragment programs to write to and read from floating-point textures at the same time. Although the results of these texture read operations are undefined, this technique has been successfully employed to implement color blending for slice-based volume visualization [8].

For the small triangle primitives of the PT algorithm, however, the results of these texture read operations are in fact erroneous—presumably due to caching of texture data. If the compositing computation in the fragment program is based on outdated data, severe caching artifacts can appear. For the NVIDIA Quadro FX 3000 graphics board used in this work, we found several ways to avoid all these caching artifacts. The two most important are:

1. rasterizing a large point primitive after each triangle fan, or
2. binding the color buffer texture *and* sending a fence (see the GL_NV_fence extension [9]) after each triangle fan.[2]

Although rather slow, the latter approach performed better and was, therefore, used for our measurements in Section 5.

The recently released NVIDIA GeForce FX 6800 (NV40 chipset) supports rendering to 16-bit floating-point color buffers with alpha blending; thus, there is no need to simultaneously read from and write to the same texture. Our preliminary measurements indicate that this allows us to use floating-point color buffers at interactive frame rates for the PT algorithm.

We found it extremely useful to employ floating-point color buffers to generate reference images because these visualizations avoid all artifacts due to color quantization during color compositing. Thus, we are able to reveal rendering artifacts that would otherwise be hidden by color quantization artifacts.

### 4.2 Alpha Dithering

For frame buffers with very limited color resolution, e.g., 8 bits per color component, Williams, Frank, and LaMar [23, 11] suggested *alpha dithering* as an efficient way to overcome quantization artifacts in volume rendering algorithms.

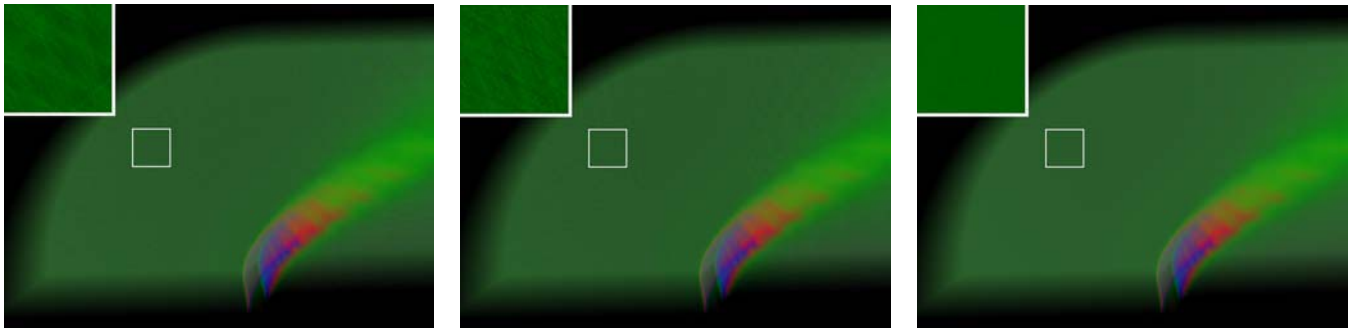
Our variant of alpha dithering customizes the 8-bit color quantization of the color result of the fragment program. The default quantization on our graphics hardware maps an output $\alpha$-component between 0 and 1 to $\lfloor 255\alpha + 1/2 \rfloor / 255$ with the floor function $\lfloor x \rfloor$ denoting the largest integer smaller than or equal to $x$. In contrast to this default quantization, we implement the following quantization at the very end of our fragment program:

$$\alpha \mapsto \begin{cases} (\lfloor 255\alpha \rfloor + 1)/255 & \text{if } 255\alpha - \lfloor 255\alpha \rfloor > q \\ \lfloor 255\alpha \rfloor / 255 & \text{otherwise} \end{cases}$$

with a pseudo-random number $q \in [0,1]$, which is determined by texture lookups in tables of random numbers. In other words, we round up with a probability equal to the fractional part of $255\alpha$, otherwise we round down.

Our randomized rounding performs very well in many cases; however, it does not avoid all rendering artifacts since the graphics

[2] This method was suggested by Nick Triantos (NVIDIA).



(a) (b) (c)

Figure 7: Rendering of the NASA blunt fin data set: (a) without HILO lookup textures, (b) without floating-point color buffer, and (c) with both HILO lookup textures and floating-point color buffer.

Table 1: Total rendering times per frame (including cell sorting) and overhead introduced by our enhancements for the blunt fin data set.

| rendering technique/enhancement | time in secs |
|---|---|
| basic pre-integrated PT algorithm (i) | 0.195 |
| overhead for perspective interpolation (ii) | 0.048 |
| overhead for HILO lookup textures (iii) | 0.031 |
| overhead for logarithmic lookup (iv) | 0.040 |
| total for frame buffer (i+ii+iii+iv) | 0.314 |
| total with alpha dithering | 0.621 |
| total for floating-point color buffer | 1.781 |

Table 2: Number of tetrahedra and total frame buffer rendering times per frame for different data sets.

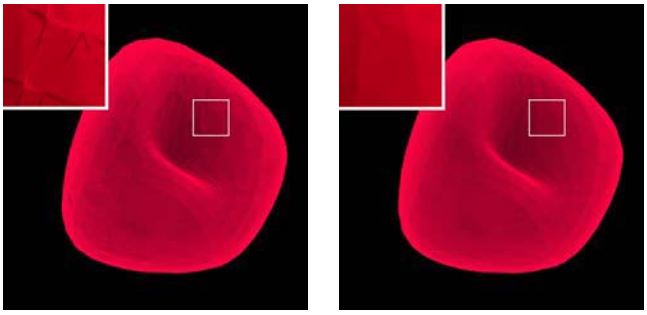
| data set | no. tets | time in secs | tets per sec |
|---|---|---|---|
| heat sink | 121,668 | 0.252 | 483K |
| blunt fin | 187,318 | 0.314 | 597K |
| cylinder | 624,960 | 0.929 | 673K |
| X38 | 1,943,483 | 3.07 | 633K |

hardware performs a second quantization after the blending operation of the fixed-function OpenGL pipeline. Unfortunately, this blending operation is not customizable; thus, we cannot avoid artifacts introduced at this point. The only exception to this rule is purely additive blending. In this case, the result of the blending is guaranteed to be quantized, and the second quantization is, therefore, ineffective. This is also the only case in which a randomized rounding of the red, green, and blue components is preferable to the randomized rounding of $\alpha$.

## 5 Results

### 5.1 Timings

We tested our implementation on a Windows XP PC with an AMD Athlon 64 3400+ processor (2.4 GHz), an AGP 8× bus, and an NVIDIA Quadro FX 3000 graphics adapter with 256 MB of video memory. As discussed in Section 2.3, our implementation takes advantage of the programmable vertex and fragment processing provided by the graphics hardware. Many factors influence the rendering performance, for example, the number of tetrahedra, the image dimensions, and the depth complexity. The transfer function does not affect the rendering performance since different transfer functions are achieved by modifying the pre-integrated lookup table.



(a) (b)

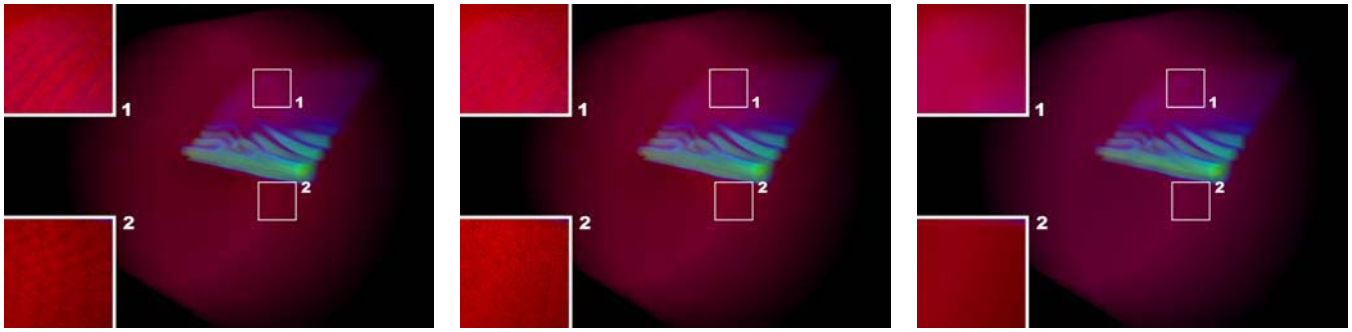
Figure 8: Isosurface visualization of the heat sink data set: (a) without and (b) with perspective interpolation.

Table 1 shows the timing results for the NASA blunt fin data set, which is decomposed into 187,318 tetrahedra before rendering. All times were measured for images of $800 \times 600$ pixels. Since our system is fragment-bound, additional instructions in the fragment program increase the rendering time. Note that we could greatly improve the rendering performance by culling transparent tetrahedra in software; however, for benchmarking purposes we are projecting all tetrahedra.

Perspective interpolation is implemented using vertex and fragment programs with twelve and eleven arithmetic instructions, respectively (see Section 2.3). The use of HILO lookup textures requires an additional 3D-texture lookup per fragment (see Section 3). The logarithmic lookup is implemented with seven arithmetic instructions per fragment, as mentioned in Section 3. For the images in this paper we always employed lookup tables of dimensions $256 \times 256 \times 16$ where 16 is the dimension of the texture corresponding to the thickness $l$.

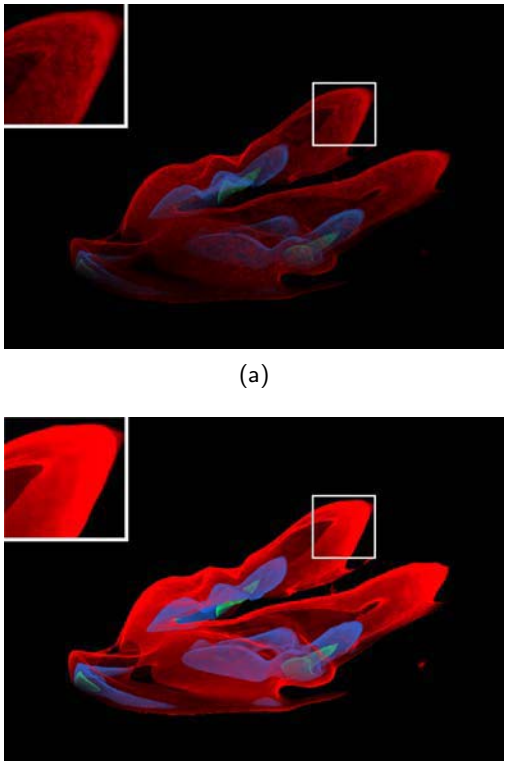
Ten arithmetic instructions and two 2D-texture lookups in pseudo-random number tables are needed per fragment for alpha dithering (see Section 4.2). Due to the additional operations required to eliminate the texture caching artifacts as discussed in Section 4.1, the color compositing in a floating-point color buffer is considerably slower than in a frame buffer.

Table 2 presents the rendering times per frame for different data sets rendered with perspective interpolation and logarithmic HILO lookup textures. The times are roughly linear in the number of projected tetrahedra but also depend strongly on the number of rasterized fragments.




(a) (b) (c)

Figure 9: Rendering of the NASA tapered cylinder data set: (a) with color compositing in the frame buffer, (b) with alpha dithering and color compositing in the frame buffer, and (c) with color compositing in a floating-point color buffer.



(a) (b)

Figure 10: Isosurface visualization of the pressure component of the NASA X38 data set: (a) with a uniform lookup texture and (b) with a logarithmic lookup texture.

Our preliminary measurements on an NVIDIA GeForce FX 6800 GT clocked at 350 MHz show only a slight increase in performance. For example, the rendering with all enhancements to an 8-bit frame buffer is about 22 % faster than with the NVIDIA Quadro FX 3000. This is presumably due to bottlenecks caused by the CPU or the AGP bus. There is, however, an important exception: the rendering with hardware-supported blending to a 16-bit floating-point color buffer is only less than 1 % slower than the rendering to an 8-bit frame buffer since no additional operations are necessary to avoid caching artifacts.

### 5.2 Comparison of Artifacts

One advantage of our implementation is that it allows us to study artifacts in terms of their causes and remedies. In the following discussion, we compare different renderings and relate artifacts to techniques that remove them.

Perspective interpolation of vertex attributes is required in the PT algorithm since an incorrect linear interpolation of texture coordinates causes erroneous coloring (see Section 3). In Figure 8, we show a volume visualization of the heat sink data set, which mimics an isosurface rendering by employing a transfer function with a sharp peak at the isovalue. Without perspective interpolation, artifacts in the form of cracks and overlaps occur along the intersections of the isosurface with cell faces, as shown in the inset in Figure 8a. In contrast, rendering with perspective interpolation shows no artifacts, as shown in the inset in Figure 8b. Note that all insets in Figures 7 to 10 are magnified and contrast-enhanced.

An 8-bit per component pre-integrated lookup texture does not offer sufficient color depth, as mentioned in Section 3. For example, the structured artifacts shown in Figure 7a result from the coarsely quantized colors of such a lookup texture. On the other hand, HILO lookup textures produce artifact-free renderings as shown in Figure 7c. Color compositing also requires a high color accuracy, as noted in Section 4.1. Figure 7b shows a rendering using a frame buffer with only 8 bits per component for color compositing. This limited color depth results in structured artifacts, which are similar to those in Figure 7a.

As explained in Section 4.2, alpha dithering alleviates rendering artifacts resulting from quantization errors to a certain extent. Figure 9a shows the NASA tapered cylinder data set rendered with an 8-bit per component frame buffer. Alpha dithering lessens artifacts in inset “1” and almost removes them in inset “2” as shown in Figure 9b. With floating-point precision for the color compositing, alpha dithering is unnecessary, as shown in Figure 9c.

As explained in Section 3, a uniform lookup texture is inappropriate if the ratio between the lengths of the longest and the shortest mesh edges is very high, which is the case in the NASA X38 data set. Very small tetrahedra have thicknesses close to zero; thus, they require lookup textures with an extremely high resolution for small thicknesses. Uniform lookup textures cannot offer such a high resolution at a feasible size and, therefore, lead to under-sampling errors for extremely small tetrahedra. This results in darker renderings with substantial edge artifacts (Figure 10a) as opposed to the artifact-free rendering with a logarithmic lookup texture (Figure 10b).

## 6 Conclusions and Future Work

We have identified and cured all major rendering artifacts that are common in implementations of the PT algorithm. These include incorrect interpolation, insufficient accuracy and dimensions of pre-integrated lookup tables, and insufficient accuracy of the frame buffer used for compositing. With our improvements, the PT algorithm is capable of achieving a rendering quality that was previously only possible with ray casting approaches.

Our solution to the limited accuracy of frame buffers is the use of floating-point color buffers, which is well supported only by the latest graphics hardware. This hardware allows us to provide the highest rendering quality at interactive frame rates.

Several of our improvements are not restricted to the PT algorithm. For example, the correct perspective interpolation could also be applied to pre-integrated texture-based volume rendering [4], and logarithmic lookup textures could be used for hardware-accelerated ray casting algorithms with an adaptive sampling rate [15, 20].

In order to further improve the PT algorithm, we plan to integrate the volume lighting method described by Lum et al. [12] with correct perspective interpolation. Moreover, we intend to validate the volume visualizations generated by our system by means of a comparison with a software ray caster. This will allow us to study the effect of incorrect visibility orderings and to compare the numerical pre-integration with analytic solutions of the volume rendering integral for piecewise-linear transfer functions.

## 7 Acknowledgments

The authors would like to thank Cass Everitt, Randall Frank, Markus Hadwiger, Mark Kilgard, Eric LaMar, Kirk Riley, Nik Svakhine, Nick Triantos, Peter Williams, and the anonymous reviewers for many helpful discussions, comments, and suggestions. We would also like to thank Kelly Gaither for providing the NASA X38 dataset and NVIDIA for providing a pre-release GeForce FX 6800 board. This material is based upon work supported by the National Science Foundation under Grant Nos. 0222675, 0081581, 0121288, 0196351, and 0328984.

## References

[1] James F. Blinn. Jim Blinn's corner: Hyperbolic interpolation. *IEEE Computer Graphics and Applications*, 12(4):89–94, 1992.

[2] Paolo Cignoni, Claudio Montani, Donatella Sarti, and Roberto Scopigno. On the optimization of projective volume rendering. In R. Scanteni, J. van Wijk, and P. Zanarini, editors, *Visualization in Scientific Computing '95*, pages 58–71. Springer-Verlag Wien, 1995.

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

[4] Klaus Engel, Martin Kraus, and Thomas Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In William Mark and Andreas Schilling, editors, *Proceedings Graphics Hardware 2001*, pages 9–16. ACM Press, 2001.

[5] Michael P. Garrity. Raytracing irregular volume data. *ACM Computer Graphics (Proceedings San Diego Workshop on Volume Visualization 1990)*, 24(5):35–40, 1990.

[6] Stefan Guthe, Stefan Roettger, Andreas Schieber, Wolfgang Strasser, and Thomas Ertl. High-quality unstructured volume rendering on the pc platform. In Thomas Ertl, Wolfgang Heidrich, and Michael Doggett, editors, *Proceedings Graphics Hardware 2002*, pages 119–125. ACM Press, 2002.

[7] Paul S. Heckbert and Henry P. Moreton. Interpolation for polygon texture mapping and shading. In David F. Rogers and Rae A. Earnshaw, editors, *State of the Art in Computer Graphics: Visualization and Modeling*, pages 101–111. Springer-Verlag, 1991.

[8] Jens Krüger and Rüdiger Westermann. Acceleration techniques for gpu-based volume rendering. In Greg Turk, Jarke J. van Wijk, and Robert Moorhead, editors, *Proceedings IEEE Visualization 2003*, pages 287–292. IEEE Computer Society Press, 2003.

[9] Mark J. Kilgard. *NVIDIA OpenGL Extension Specifications*. NVIDIA Corporation, 2001.

[10] Martin Kraus and Thomas Ertl. Cell-projection of cyclic meshes. In Thomas Ertl, Kenneth Joy, and Amitabh Varshney, editors, *Proceedings IEEE Visualization 2001*, pages 215–222. IEEE Computer Society Press, 2001.

[11] Eric C. LaMar. On issues of precision for hardware texture-based volume visualization. In R. F. Erbacher, P. C. Chen, J. C. Roberts, and Craig M. Wittenbrink, editors, *Visual Data Exploration and Analysis*, pages 19–23. SPIE – The International Society for Optical Engineering, 2004.

[12] Eric B. Lum, Brett Wilson, and Kwa-Liu Ma. High-quality lighting and efficient pre-integration for volume rendering. In *Proceedings Joint Eurographics-IEEE TVCG Symposium on Visualization 2004 (VisSym '04)*, pages 25–34, 2004.

[13] Nelson Max, Pat Hanrahan, and Roger Crawfis. Area and volume coherence for efficient visualization of 3d scalar functions. *ACM Computer Graphics (Proceedings San Diego Workshop on Volume Visualization 1990)*, 24(5):27–33, 1990.

[14] Stefan Roettger and Thomas Ertl. A two-step approach for interactive pre-integrated volume rendering of unstructured grids. In Roger Crawfis, Chris Johnson, and Klaus Mueller, editors, *Proceedings Volume Visualization and Graphics Symposium 2002*, pages 23–28. ACM Press, 2002.

[15] Stefan Roettger, Stefan Guthe, Daniel Weiskopf, Thomas Ertl, and Wolfgang Strasser. Smart hardware-accelerated volume rendering. In *Proceedings Joint Eurographics-IEEE TCVG Symposium on Visualization (VisSym '03)*, pages 231–238, 2003.

[16] Stefan Röttger, Martin Kraus, and Thomas Ertl. Hardware-accelerated volume and isosurface rendering based on cell-projection. In Thomas Ertl, Bernd Hamann, and Amitabh Varshney, editors, *Proceedings IEEE Visualization 2000*, pages 109–116. IEEE Computer Society Press, 2000.

[17] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 1.5)*. Silicon Graphics, Inc., 2003.

[18] Peter Shirley and Allan Tuchman. A polygonal approximation to direct scalar volume rendering. *ACM Computer Graphics (Proceedings San Diego Workshop on Volume Visualization 1990)*, 24(5):63–70, 1990.

[19] Clifford M. Stein, Barry G. Becker, and Nelson L. Max. Sorting and hardware assisted rendering for volume visualization. In Arie Kaufman and Wolfgang Krueger, editors, *Proceedings 1994 Symposium on Volume Visualization*, pages 83–89. ACM Press, 1994.

[20] Manfred Weiler, Martin Kraus, Markus Merz, and Thomas Ertl. Hardware-based ray casting for tetrahedral meshes. In Greg Turk, Jarke J. van Wijk, and Robert Moorhead, editors, *Proceedings IEEE Visualization 2003*, pages 333–340. IEEE Computer Society Press, 2003.

[21] 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, 2003.

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

[23] Peter L. Williams, Randall J. Frank, and Eric C. LaMar. Alpha dithering to correct low-opacity 8 bit compositing errors. Lawrence Livermore National Laboratory Technical Report UCRL-ID-153185, March 2003.