DOI: 10.1111/j.1467-8659.2007.01038.x

Volume and Isosurface Rendering with GPU-Accelerated Cell Projection*

R. Marroquim, A. Maximo, R. Farias and C. Esperança

Universidade Federal do Rio de Janeiro - COPPE/UFRJ, Brazil
{ricardo@lcg.ufrj.br}

Abstract

We present an efficient Graphics Processing Unit GPU-based implementation of the Projected Tetrahedra (PT) algorithm. By reducing most of the CPU–GPU data transfer, the algorithm achieves interactive frame rates (up to 2.0 M Tets/s) on current graphics hardware. Since no topology information is stored, it requires substantially less memory than recent interactive ray casting approaches. The method uses a two-pass GPU approach with two fragment shaders. This work includes extended volume inspection capabilities by supporting interactive transfer function editing and isosurface highlighting using a Phong illumination model.

Keywords: Direct volume rendering, Cell Projection, and Isosurface Rendering

ACM CCS: I.3.3 Computer Graphics: Picture/Image Generation I.3.6 [Computer Graphics]: Methodology and Techniques I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

1. Introduction

The volume rendering field has many useful and important applications, such as inspection of medical images, visualization of geological data and fluid simulation, among other. The input is three-dimensional (3D) data (possibly time-varying) representing a particular attribute of a given volume, such as density, heat, velocity, etc. In this paper, we are interested in volumes defined as scalar fields, decomposed in tetrahedra, where the field samples are attached to mesh vertices.

Volume data contains 3D information acquired from different source types: sampling, simulation and modeling. For instance, images sequences obtained from Magnetic Resonance Imaging (MRI) or Computed Tomography (CT) are real material samplings, while the second type may come from fluid or geological simulation results. The last source comes from the area of *volume graphics*, where the volume visualization techniques are used for modeling and manipulating instead of visualizing.

Recently, new algorithms were developed [WKME03b, WKME03a, CRZP04] aiming to exploit programmable graphics hardware. The so-called Graphics Processing Units (GPUs) are vectorial processing devices that allow a single code to be applied to multiple data in parallel. With this, common computers can be turned into vectorial machines specifically designed for volume rendering.

We present a GPU-based implementation of a well-known algorithm – the *Projected Tetrahedra* (PT) of Shirley and Tuchman [ST90] – for interactive volume visualization and manipulation. This work builds upon an earlier implementation (see [MMFE06]) by adding two key features: a rationale for discarding tetrahedra that do not contribute to the final image and the ability to blend Phong-lighted isosurfaces with the standard PT-based volume visualization.

The rest of this paper is divided as follows. In Section 2, we discuss related work on volume rendering techniques and corresponding GPU approaches. Next, we review the PT algorithm in Section 3. We present our algorithm in Section 4

*Based on "GPU-Based Cell Projection for Interactive Volume Rendering", by Ricardo Marroquim, André Maximo, Ricardo Farias and Claudio Esperança, which appeared in *Proceedings of SIGRAPI 2006: XIX Brazilian Symposium on Computer Graphics and Image Processing*, Manual M. Oliveira and Rodrigo L. Carceroni (Eds.), Manaus, AM, Brazil, 8–10 October 2006, IEEE Computer Society Press. © 2006 IEEE.

and its results in Section 5. Finally, in Section 6, conclusions and future work directions are provided.

2. Related Work

Volume rendering consists of a series of techniques for analysing volumetric data and extracting significant information [KM05]. These can be roughly divided in three categories: *object-order*, *image-order* and *domain-based*. In the object-order algorithms, the contributions of each cell are evaluated and combined to produce the final image. For instance, the *cell projection* method projects each cell on the image plane and composes them in visibility order. In image-order methods, for example *ray casting*, rays are cast from each pixel of the image through the volume data. In the domain-based methods, spatial data is transformed into another domain, for example, *frequency domain* and then the final image is produced directly from the new domain.

One of the aspects which make volume visualization particularly challenging is the fact that 3D information must be efficiently summarized in a 2D image. For this purpose, volume data represented in scalar fields is usually mapped to color values by means of a *transfer function* [UK88]. The opacity value computation is based on the extinction coefficient, which models the light absorption inside the volume [Max95]. In the same manner as the color, the extinction coefficient is also mapped by the transfer function.

The volume rendering integral evaluates the physical interaction of light rays with the volume. This integral is an equation that computes the light color traversing the volume, using the ray's entry and exit values as well as the traversed distance inside each cell. Williams and Max [WM92] describe an exact method for computing this equation. However, due to computational limits, it is impractical for interactive applications.

Rather than computing integrals precisely by using small steps, a technique known as *pre-integration* uses a precomputed table which maps the volume rendering integral as a function of length traversed by the ray and the scalar values at entry and exit points. Moreland and Angel [MA04] introduced the partial pre-integration technique, where the precomputation of the integral equation does not depend on the transfer function and thus can be stored for future use. We make use of the ψ table, generated by Moreland and Angel, precompiled within our implementation. This table allows interactive transfer function editing together with the benefits of the pre-integration technique.

Lighting models play an important role in volume visualization by illustrating isosurfaces. Lum and Ma [LM04] use a multidimensional transfer function to enhance surfaces. As all methods that use gradients for lighting, their method is less effective for noisy data. Another problem arises near homogeneous regions where gradient directions are not well defined.

Some early methods for volume rendering [DCH88, Lev88, Sab88] describe a volume lighting algorithm which approximates isosurface normals by the gradient of the scalar field. In this paper, we use these concepts to achieve interactive volume rendering with fast isosurface shading.

Many algorithms have been proposed for volume rendering in the past. Ray casting [KH84] is perhaps the most common approach and several high-performance implementations have been developed. Most of these make use of GPUs, such as the Hardware-Based Ray Casting (HARC) [WKME03a, EC05]. Traditional cell projection algorithms [FGR85, ST90] have also been reimplemented aided by GPU programming [WMFC02, RKE00]. An interesting algorithm which combines ray casting and cell projection is the View-Independent Cell Projection (VICP) [WKME03b] of Weiler *et al.* Other common approaches include splatting algorithms [Wes90, CRZP04], sweeping [Gie92, SM97, FMS00] and 3D textures [WVW94, KW03].

Our algorithm is based on the PT technique introduced by Shirley and Tuchman [ST90]. The PT algorithm is a cell projection approach where each tetrahedron cell is projected and composed in the image plane. An overview of the PT method is given on Section 3.

Wylie *et al.* [WMFC02] developed the GATOR algorithm, an adaptation of the PT algorithm for GPU using a vertex shader. By creating a *basis graph*, they were able to redefine the different projection classes in a manner that one vertex shader can treat all cases in the same way. The GATOR algorithm is fast yet redundant, since for each vertex computation, all of the other tetrahedron vertices must be made available. Another issue of this method is that the final color is poorly approximated.

The main drawback of the PT algorithm is the need of a visibility ordering of the cells. This problem has been specifically addressed on many references, e.g. [Wil92, SBM94, CKM*99, CICS05], and is not focused in this paper. Our main contribution resides on a high performance realization of the PT algorithm, which is executed almost entirely on GPU, allowing interactive visualization of tetrahedral meshes. In particular, our implementation performs on a par with recently reported ray casting algorithms while being significantly more thrifty in memory usage.

3. Projected Tetrahedra Algorithm

Basically, the PT algorithm consists of projecting the tetrahedra to screen space and composing them in visibility order. The first implication of this method is that the 3D data must be *tetrahedralated*. In the case of regular volumes, this can be accomplished by subdividing each lattice cell into five tetrahedra.

The projected tetrahedra are decomposed into triangles according to a classification scheme. For each of the four different classes a tetrahedron is decomposed into a specific number of triangles. For example, class 1 projections are decomposed into three triangles while class 2 are decomposed into four. It should be noted that classes 3 and 4 are degenerate cases of classes 1 and 2, where a vertex is projected onto an edge or another vertex (see Figure 1).

The *thick* vertex is defined for each projection as the point of the ray segment that traverses the maximum distance through the tetrahedron. Analogously, the other vertices are called *thin* vertices, since no distance is covered. For class 2 projections, the thick vertex is computed as the intersection between the front and back edges, while for the other classes it is one of the projected vertices. The scalar values of the ray's entry and exit points are named as s_f and s_b (see Figure 2). Thin vertices have the same values for s_f and s_b, while for thick vertices these values may have to be interpolated from those of the thin vertices. The distance traversed by the ray segment is defined as the thickness l of the cell.

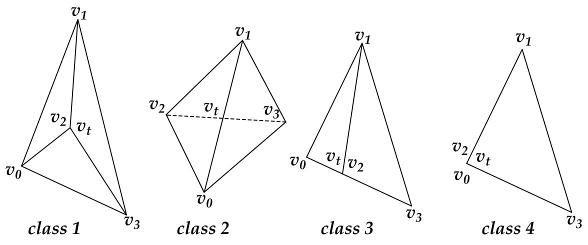

Figure 1: *The different classifications of the projected tetrahedra algorithm, where v_t is the thick vertex, and v_i are the projected vertices.*

Before rendering the triangles, color values must be assigned to each vertex. A transfer function is used to map the scalar values into chromaticity and opacity values. For a scalar value s, a table is looked up in order to obtain the RGB chromaticity values $[C(s)]$ and extinction coefficient $[\tau(s)]$, which is directly associated with the opacity. Thin vertices are rendered with zero opacity and the original chromaticity from the transfer function. On the other hand, the thick vertex is rendered with the average color between the front and back scalar values and opacity $\alpha = 1 - e^{-\tau_{avg} l}$, where τ_{avg} is the average of $\tau(s_f)$ and $\tau(s_b)$.

Finally, the rendered triangles are rasterized into pixel fragments by interpolating the thick and thin vertices color and opacity values. The fragments are composited in back-to-front order, and, for each new color added to the framebuffer, the new final pixel color is computed by $I_{new} = \alpha C + (1 - \alpha) I$, where I is the previous color stored in the framebuffer, and C and α are the interpolated color and opacity values.

4. A Two-Pass GPU Approach

The algorithm is executed mostly in the GPU and is divided in two main parts. In the first step, all relevant information of

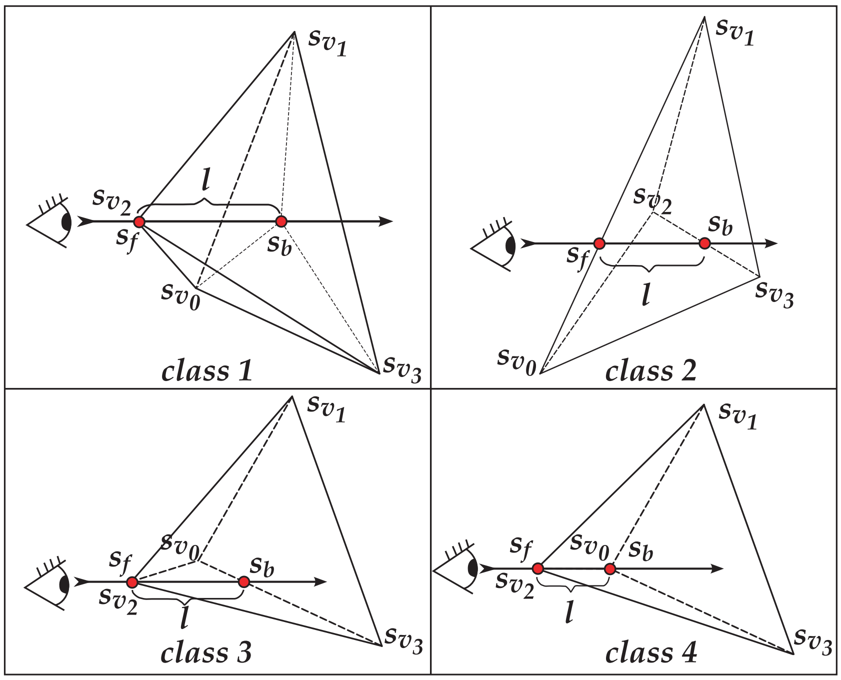

Figure 2: *One example for each projection class illustrating the tetrahedron in 3D space and the viewing ray intersecting it. The scalar value is defined as s_{v_i} for each projected vertex v_i, while s_f and s_b are the values at the ray's entry and exit points in the thick vertex, respectively.*

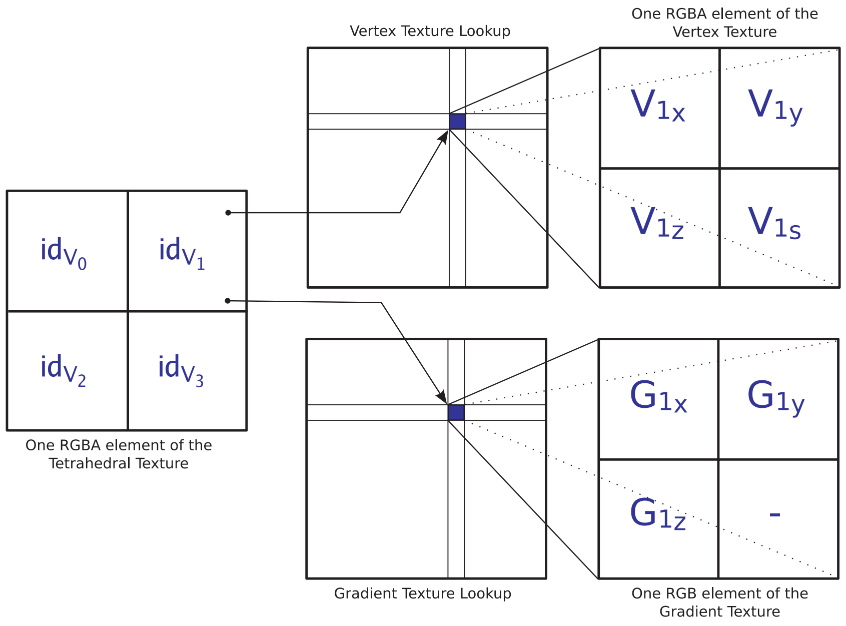

Figure 3: *Vertex data retrieval in the first fragment shader. Each texel of the Tetrahedral Texture contains the indices of its four vertices in the Vertex and Gradient Textures.*

each tetrahedron is computed, that is, the projection class, the thick vertex properties, and the z coordinate of the tetrahedron's centroid. During the second step, vertices' scalar and gradient values are interpolated to compute chromaticity and opacity values for each fragment. If an isosurface is detected, it is rendered with Phong shading using the gradient as the normal vector, otherwise the volume ray integral is applied.

To increase the frame rates, we make use of vertex arrays and the primitives are drawn as triangle fans. Each fan is drawn according to an order and number of vertices determined in the first step and passed on to the second, as described later. In addition, a discard test is applied beforehand to avoid rendering tetrahedra that do not contribute to the final image.

4.1. First Pass – Projecting the Tetrahedra

The first pass consists of computing all data per tetrahedron in the fragment shader. All relevant information has to be made accessible to the shader. This is accomplished by storing four different textures in the GPU memory: the Tetrahedral Texture, the Vertex Texture, the Gradient Texture and the Classification Texture. The first three textures have 32 bits per component and the fourth has 8 bits per component. The Gradient Texture has three components per texel, while the others are *RGBA* textures with four components per texel.

Each tetrahedron is associated with one texel of the Tetrahedral Texture, which contains four index values for retrieving data for the four vertices. The coordinates and associated scalar value of each vertex are stored in the Vertex Texture as one texel, where *RGB* fields store the x, y and z coordinates and the A field stores the scalar s. The Gradient Texture has the same size as the Vertex Texture and is accessed by the same indices. Each of its texels stores the precomputed gradient vector at the vertex (see Figure 3). These three texture lookups eliminate the need for vertex attributes and reduce the data transfer overhead from CPU to GPU. Even though the cost of passing attributes is replaced by that of texture fetches, it is still faster, since all operations remain exclusively inside the GPU.

To execute the fragment shader once per tetrahedron, the Tetrahedral Texture is rendered as a quadrilateral with the same size as the screen space, so that the number of texels is equal to the number of pixels (approximately the number of tetrahedra). This method is often used in so-called General Purpose GPU (GPGPU) algorithms. For each tetrahedron, the following values are computed inside the shader:

- s_f and s_b: front and back scalar values.
- g_f and g_b: front and back gradient vectors.
- l: cell thickness.
- c_z: cell centroid.
- Vertex order and number of triangles in the fan.

To determine the projection class of a tetrahedron, vertex coordinates are first projected to screen space. With four simple tests it is possible to determine not only the projection class, but also how the vertices are layed out in relation to each other. This classification process is very similar to Wylie's [WMFC02] method, except that we also treat degenerate cases. In addition, our method avoids computational redundancy by performing the tests once per tetrahedron rather than once per vertex.

Each test t_i is an evaluation of a cross-product computed with the projected vertices v_{proj_i} (see Figure 4). For each test there are three possible results (0, 1 or 2) depending on whether the z coordinate of each cross-product is negative, zero or positive, respectively. The test results are combined together to produce an unique index to the Classification Texture. This 1D texture contains a Ternary Truth Table with all possible test result permutations. On top of Wylie's 14 class 1 and 2 cases, we have added 24 class 3 cases and 12 class 4 cases covering all degenerate projections. With one lookup operation, the correct order to compute the intersection parameters is retrieved.

The number of triangles of the degenerate cases are directly associated with the number of results equal to 1. If one such result is detected, then a class 3 tetrahedron is projected as two triangles. If two such results are detected, then a class 4 tetrahedron is projected as one triangle. Three or four tests yielding 1 indicate degenerate tetrahedra which can be discarded.

$vec_1 = v_{proj_1} - v_{proj_0}$	$t_1 = sign((vec_1 \times vec_2).z) + 1$
$vec_2 = v_{proj_2} - v_{proj_0}$	$t_2 = sign((vec_1 \times vec_3).z) + 1$
$vec_3 = v_{proj_3} - v_{proj_0}$	$t_3 = sign((vec_2 \times vec_3).z) + 1$
$vec_4 = v_{proj_1} - v_{proj_2}$	$t_4 = sign((vec_4 \times vec_5).z) + 1$
$vec_5 = v_{proj_1} - v_{proj_3}$	

Figure 4: *Tests performed in fragment shader for projection classification. The GLSL built-in function sign returns −1, 0 or 1 depending on whether the argument is less than, equal to or greater than zero, respectively.*

The first pass is also responsible for computing the scalar values for the thick vertex as discussed in Section 3. Gradient values for the entry and exit points (g_f and g_b) are interpolated in much the same way.

Finally, all computed data is sent back to the CPU using multiple render targets (MRT) with four frame buffer color attachments. Each attachment is a 2D *RGBA* texture with 32 bits per component. The first texture receives the intersection vertex coordinates x_{v_i} and y_{v_i} (used only for class 2), the centroid of the tetrahedron c_z, and the index to the Ternary Truth Table id_{TTT}. The second texture contains the scalar values s_f and s_b, the thickness l and the number of triangles generated by the projection *count*. The third and fourth textures holds the front and back gradient vectors g_f and g_b, respectively. This scheme is depicted in Figure 5.

4.2. Preparing the Arrays for Rendering

Before rendering the primitives, the data must be retrieved from the output textures, sorted and stored in arrays. Our implementation currently uses a simple bucket-based sort when the model is being rotated and the STL-implemented merge-sort for a static model. Triangle fans are rendered with the optimized OpenGL function *glMultiDrawElements*, whose arguments reference global arrays storing vertex information. In particular, we make use of four different global arrays for storing vertex coordinates, color values and gradient values (see Figure 6).

The four global arrays contain the tetrahedra grouped in five elements: the thick vertex plus the four original vertices. Since for each change in the view direction only the position,

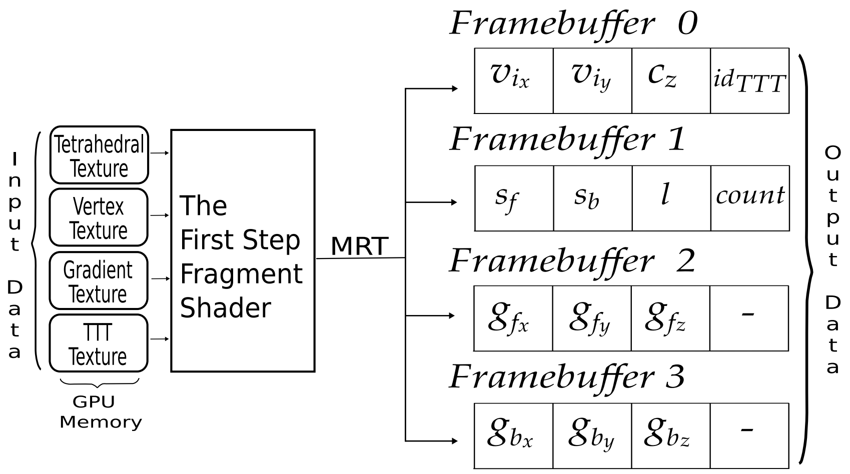

Figure 5: *Fragment shader input/output scheme, where TTT is the Ternary Truth Table.*

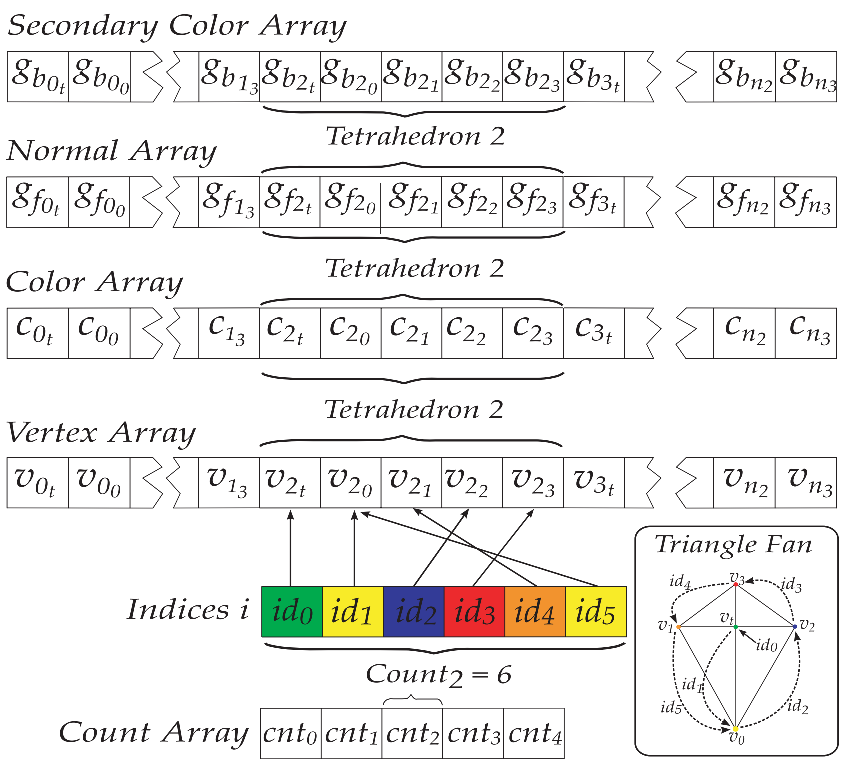

Figure 6: *Arrays data structure. The indices illustrate a class 1 case, where the correct order to draw tetrahedron i is* $v_{i_t} - v_{i_0} - v_{i_2} - v_{i_3} - v_{i_1} - v_{i_0}$.

color and gradients of the thick vertices are updated, most of these arrays are constant. This implies that OpenGL is able to maintain most of the information in the GPU memory, avoiding data transfer overhead.

The vertex array contains the coordinates $\{x, y, z\}$ of each vertex. The color array contains values $\{s_f, s_b, l\}$ rather than actual colors, which will be computed on-the-fly by the second fragment shader. Finally, gradients are stored in the normal and secondary color arrays. Note that for thin vertices $s_f = s_b$, $l = 0$ and $g_f = g_b$.

The two arrays passed as arguments to *glMultiDrawElements* are constructed as follows. The *indices* array is divided into n groups, where n is the number of tetrahedra. For each tetrahedron, the correct order to render the triangle fan is stored as six integers indexing its vertices. The *cnt* array contains the number of vertices in each fan. Recall that the maximum number of vertices in a fan is six (class 2) with four triangles. For cases with less than six vertices the *indices* array is only accessed up to position *cnt*. Refer to Figure 6 for further details.

4.3. Second Pass – Rendering the Primitives

The second shader computes the final color of each fragment. The vertex colors are linearly interpolated inside each triangle using the values of a thick and two thin vertices. Since the color is actually passed as $\{s_f, s_b, l\}$, these interpolated values are used in the fragment shader to compute the final color. The simplest way to compute the final color is using the average scalar values as described in Section 3.

To compute the final opacity value, we make use of a 1D texture. This texture contains values for e^{-u}, where u is regularly sampled over interval [0, 1]. The lookup is done by passing $u = \tau l$ to obtain the final exponential opacity value. Our experiments show that using this 1D texture is slightly faster than computing the exponential function in the fragment shader.

4.4. Partial Pre-Integration

Rather than estimating the final color using average scalar values, we used the so-called ψ table of the partial pre-integration technique [MA04]. Since this table does not depend on any attribute of the visualization, it is pre-compiled within our implementation.

In the second fragment shader, the colors associated with s_f and s_b are retrieved from a second 1D transfer function texture. The C_f and C_b together with the thickness value l are used to compute the indices of the ψ table, stored in a

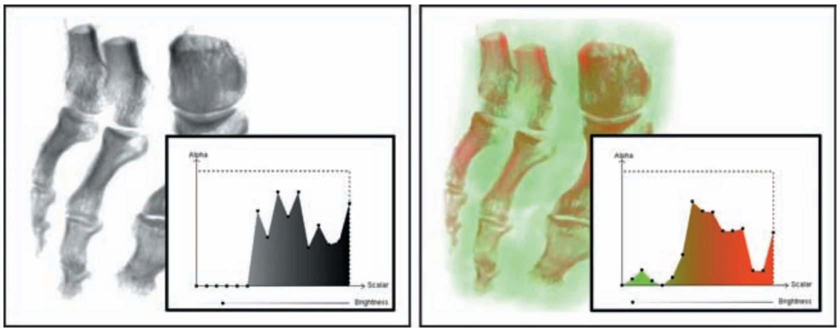

Figure 7: *Foot data set with two different transfer functions.*

2D texture. The value retrieved from this table is then used to compute the final fragment color.

In contrast with the original PT method, the partial pre-integration method is slower than computing the final color using the average method, but faster than using full pre-integration.

4.5. Transfer Function Editing

The use of the partial pre-integration method allows the transfer function to be interactively edited. Every time a change occurs, the transfer function texture is recomputed and uploaded again to the fragment shader.

Control points are placed over the color map to change the opacity levels in the selected neighborhood. In addition, the user can cycle through different colormaps varying from gray-scale maps to multicolored maps. The brightness of the image can also be adjusted and acts as a global opacity factor. An example of two different transfer functions for the Foot model is shown on Figure 7.

An early discard test is employed to bypass tetrahedra that have zero opacity and will add no contribution to the final image. Only tetrahedra with all vertices having zero opacity are discarded, that is, their scalar values are mapped to colors with zero opacity on the transfer function.

The arrays from Section 4.2 are recreated each time a control point of the transfer function either reaches or leaves the zero opacity level. Even though this process reduces the interactivity of the transfer function editing, it greatly increases the rendering rates in terms of *fps*.

4.6. Rendering Isosurfaces

Apart from the transfer function interactive editing tool, the user also has control over isosurfaces. An isovalue is associated with one scalar value as a function $f(s)$. For each fragment it is determined if the chosen isosurface crosses the respective tetrahedron. Since each fragment contains the

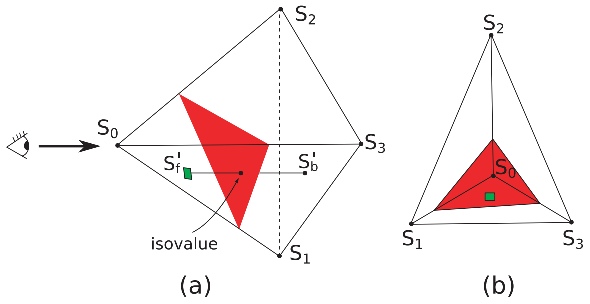

Figure 8: *(a) An isosurface crossing a tetrahedron. (b) The projected tetrahedron with the rendered contribution of the isosurface rendered.*

front and back interpolated scalar values, the test consists of simply checking if the isovalue is inside the given range. If so, the surface cuts the current tetrahedron and the color computation is managed differently (see Figure 8).

This approach is similar to [KSE04]. However, we are not computing the isosurface per tetrahedron explicitly, but per fragment taking advantage of our rendering pipeline. Hence, there is no need to worry about different cases of how the isosurface cuts a tetrahedron, as in the GPU based marching tetrahedra algorithm [Pas04].

The isosurfaces are rendered using a Phong illumination method. The interpolated gradients act as the normal vector for computing the diffuse light reflection and specular highlights. Illumination controls permit the adjustment of diffuse, specular and ambient coefficients. In addition, the opacity of each rendered surface can also be controlled.

The front and back gradients are combined with weights relative to the distance between the isosurface and the front and back intersection points. This means that if the isosurface is closer to the ray's exit point, for example, the back gradient has more weight on the normal vector computation than the front gradient.

Table 1. *Time comparison between our algorithm (PTINT) and others approaches.*

Algorithm	Blunt Fin	Oxygen Post
PTINT	11.30 *fps*	4.49 *fps*
GATOR	4.07 *fps*	1.51 *fps*
VICP (GPU)	5.20 *fps*	1.93 *fps*
VICP (CPU)	1.82 *fps*	0.57 *fps*
VICP (Balanced)	4.10 *fps*	1.11 *fps*
HARC	4.47 *fps*	8.63 *fps*
HARC (INT)	4.94 *fps*	5.93 *fps*
HAVIS	2.36 *fps*	0.79 *fps*
HAVS (k = 2)	6.09 *fps*	3.09 *fps*
HAVS (k = 6)	3.45 *fps*	2.09 *fps*

This technique allows a hybrid visualization of the volume with multiple isosurfaces. The isosurfaces can be rendered with fully opaque or semi-transparent fragments. Since all isosurface computation is performed in the second fragment shader, no additional rendering pass is needed. Furthermore, if no isosurface or lighting are desired, the algorithm can regain significant performance by leaving out all gradient computations including: computing g_f and g_b and rendering to framebuffers 2 and 3 on the first step, updating the two gradient arrays on the intermediate CPU pass, and interpolating the gradients and computing the normals on the second pass.

5. Results

Our prototype was programmed in C++ using OpenGL 2.0 with GLSL under Linux. Performance measurements were made on a Intel Pentium IV 3.6 GHz, 2 GB RAM, with a nVidia GeForce 6800 256 MB graphics card and a PCI Express 16x bus interface.

The data sets used were: Blunt Fin (blunt), Combustion Chamber (comb), Liquid Oxygen Post (post), Super Phoenix (spx), Fuel Injection (fuel) and Electron Distribution Probability (neghip).

The timings are given using a 512^2 pixel viewport and considering that the model is constantly rotating. Table 1 compares our algorithm (PTINT) with others volume rendering algorithms:

- PTINT – The proposed approach: PT with partial pre-integration.
- GATOR – GPU Accelerated Tetrahedra Renderer [WMFC02].
- VICP – View-Independent Cell Projection (implemented in GPU and CPU) [WKME03b];
- VICP (Balanced) – VICP with GPU-CPU balancing [MA04].

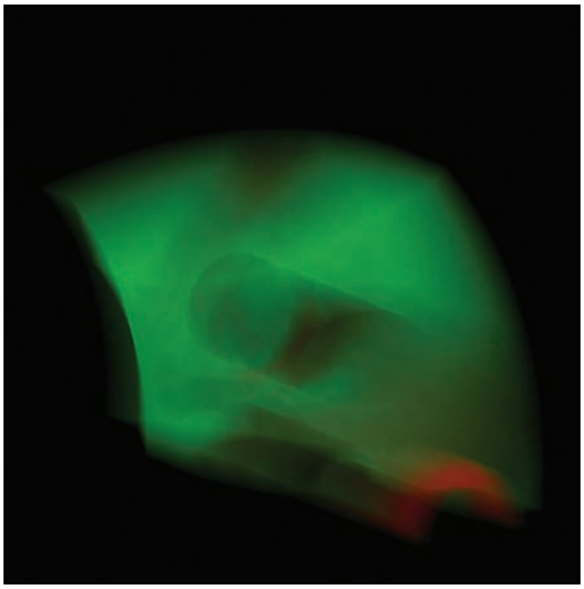
Figure 9: *Spx data set.*

- HARC – Hardware-Based Ray Casting [WKME03a];
- HARC (INT) – HARC with partial pre-integration [EC05].
- HAVIS – Hardware-Accelerated Volume and Isosurface Rendering Based on Cell-Projection [RKE00].
- HAVS - Hardware Assisted Visibility Sorting [CICS05].

Table 2 further specifies the number of vertices (*# Verts*) and tetrahedra (*# Tet*) for each data set, and the average number of frames per second (*fps*) and tetrahedra per second (*Tet/s*) of our algorithm in three variants: average scalar method (*basic*); partial pre-integration technique (*pre-integration*); and using both partial pre-integration and isosurface rendering (*isosurfaces*).

The timings given in Table 1 consider the basic integration approach but without the tetrahedra discard method, therefore the *fps* is lower than the ones shown in Table 2. It is important to note the differences between the tested data sets. For the Blunt Fin, our algorithm performs better than all others approaches. However, for the Oxygen Post data set, it loses to the ray casting algorithms. This can be attributed to the fact that, for some view points, the model has small pixel area, while for cell projection approaches this pixel area size is irrelevant.

The Spx data set is shown in Figure 9 rendered with the basic average scalar method. In Figure 11, the Oxygen Post (a) image was generated with only the pre-integration technique while for Neghip (b) pre-integration and volume shading was used.

Table 2: *Data set sizes and average times of our algorithm in three variants: basic average scalar method; partial pre-integration technique; and partial pre-integration and isosurface rendering.*

Data	Size		*Basic*		*+Pre-integration*		*+Isosurfaces*	
sets	*# Verts*	*# Tet*	*fps*	*M Tet/s*	*fps*	*M Tet/s*	*fps*	*M Tet/s*
Blunt	40 K	187 K	12.75	2.38	10.76	2.01	4.97	0.93
Comb	47 K	215 K	11.12	2.38	8.98	1.92	3.71	0.79
Post	110 K	513 K	4.91	2.51	4.35	2.23	2.09	1.07
Spx	150 K	828 K	4.68	2.55	4.52	2.47	1.23	1.02
Fuel	262 K	1.25 M	26.01	2.10	22.99	1.86	9.95	0.80
Neghip	262 K	1.25 M	3.59	2.34	3.11	2.03	1.27	0.83

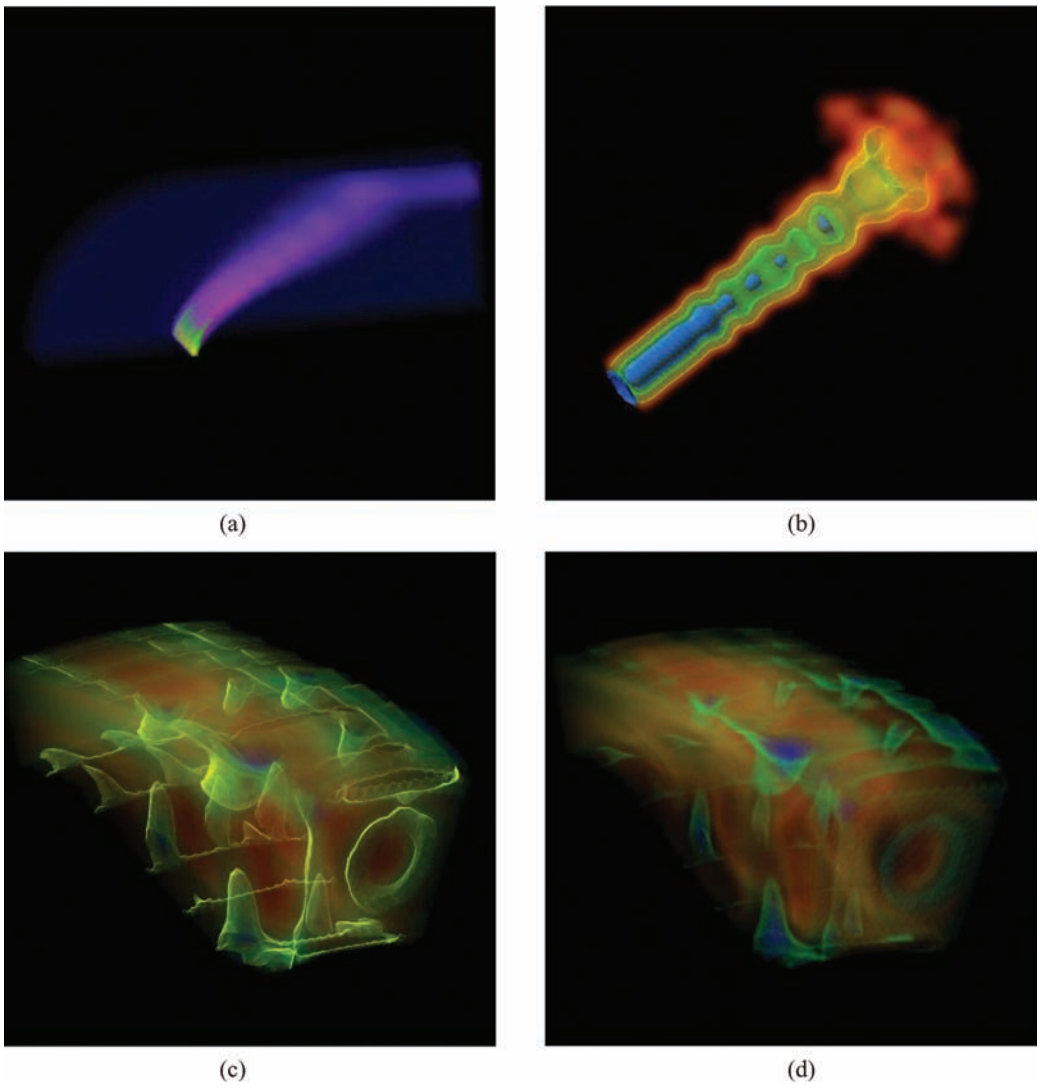

Figure 10: *Data sets: Blunt Fin (a), Fuel Injection (b) and Combustion Chamber (c and d).*

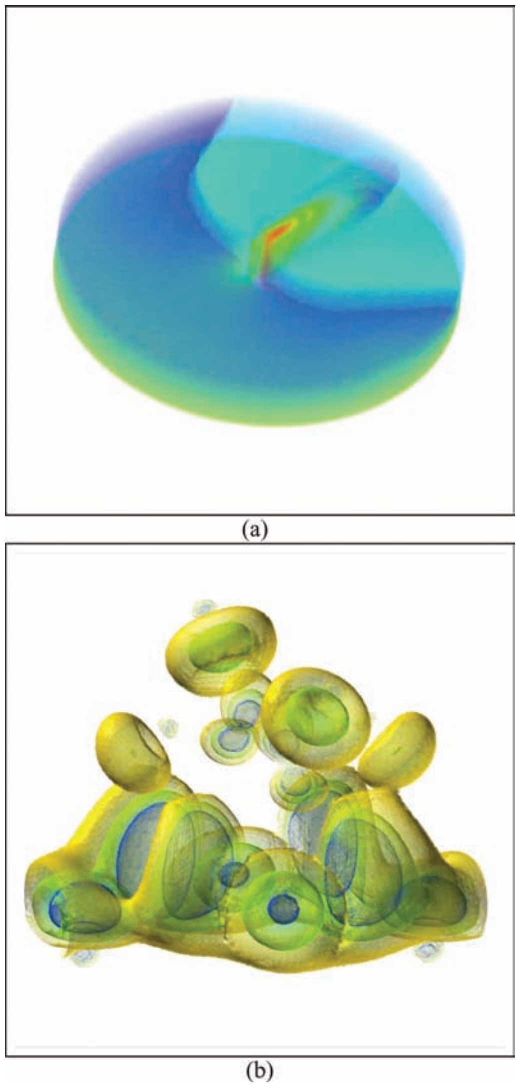

Figure 11: *Liquid Oxygen Post (a) and Electron Distribution Probability (b) data sets.*

Three models rendered with our algorithm are shown in Figure 10. The Blunt Fin (a) and Comb (d) data sets were rendered with only the pre-integration technique. While the Fuel Injection (b) and Comb (c) data sets have been rendered with isosurface highlighting.

6. Conclusions and Future work

We have presented a hybrid isosurface and cell projection volume rendering method that takes full advantage of modern graphics hardware. The algorithm achieves interactive frame rates by eliminating much of the bus transfer overhead. By keeping the whole model in GPU memory, we limit the data access to internal texture fetches. Further performance improvements were obtained by employing an early tetrahedral discard test.

By mixing direct volume rendering with isosurface rendering, more comprehensive visualizations can be obtained. For example, by making some isosurfaces fully opaque, underlying volumes can be hidden. On the other hand, superimposing layers can be enhanced with other isosurfaces while still allowing other details to be shown by means of direct volume rendering.

Better visualization of the isosurfaces is possible using a Phong shading model. Diffuse, specular and ambient coefficients can be controlled for better model inspection. The only drawback being the considerable performance loss due to gradient computation and interpolation.

Furthermore, the interactive transfer function and illumination control tools give the user a fast way to manipulate and inspect the model. It is possible to cycle through different color maps and manipulate a varying number of controls points. The isosurface inspection is made simple with slide controls for determining the isovalues and their respective opacity values, as well the Phong illumination coefficients.

Finally, our implementation requires less storage space than Ray-Casting algorithms. This is mainly due to the fact that no cell connectivity is needed, freeing the GPU memory of heavy data structures. In our implementation, each texture element occupies 16 bytes, which means that a data set with n vertices and m tetrahedra will require $16(2n + m)$ texture bytes. Assuming an average of 4 tets per vertex, the storage space needed per tet is 24 bytes. Roughly, one million cells occupy 24 MB of GPU memory (20 MB without illumination). While, the HARC PPI [EC05] requires 96 bytes/tet and the original HARC implementation requires 144 bytes/tet.

As one improvement for the current work, we are currently researching semi-automatic transfer functions design ideas. Even though interactive control over the transfer function is desirable, it is important to allow more intuitive ways to manipulate and generate them.

Another topic of interest is the improvement of the sorting algorithm, as none of the two used in this work guarantee a correct visibility order for all cases. This can be accomplished by using algorithms such as the Hardware-Assisted Visibility Sorting (HAVS) by Callahan *et al.* [CICS05]. However, merging this approach with our algorithm is not trivial. As a faster alternative for the current implementation, there are other GPU-based sorting approaches such as the bitonic merge sort [PDC*03] or Govindaraju *et al's.* algorithm [GHLM05]. The latter claims to order the cells in close to linear time when taking advantage of spatial coherence. However, it has not yet been tested with volume rendering applications to our knowledge.

Acknowledgments

We would like to thank Rodrigo Espinha and Waldemar Celes for providing us with their implementation of HARC algorithm. We also acknowledge the grant of the first author provided by Brazilian agency CNPq (National Counsel of Technological and Scientific Development).

References

[CICS05] CALLAHAN S. P., IKITS M., COMBA J. L. D., SILVA C. T.: Hardware-assisted visibility sorting for unstructured volume rendering. *IEEE Transactions on Visualization and Computer Graphics 11*, 3 (May/June 2005), 285–298.

[CKM*99] COMBA J., KLOSOWSKI J. T., MAX N. L., MITCHELL J. S. B., SILVA C. T., WILLIAMS P. L.: Fast polyhedral cell sorting for interactive rendering of unstructured grids. *Computer Graphics Forum 18*, 3 (1999), 369–376.

[CRZP04] CHEN W., REN L., ZWICKER M., PFISTER H.: Hardware-accelerated adaptive EWA volume splatting. In *VIS' 04: Proceedings of the conference on Visualization'04* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 67–74.

[DCH88] DREBIN R. A., CARPENTER L., HANRAHAN P.: Volume rendering. In *SIGGRAPH'88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1988), ACM Press, pp. 65–74.

[EC05] ESPINHA R., CELES W.: High-quality hardware-based ray-casting volume rendering using partial pre-integration. In *SIBGRAPI*'05: *Proceedings of the XVIII Brazilian Symposium on Computer Graphics and Image Processing* (2005), IEEE Computer Society, p. 273.

[FGR85] FRIEDER G., GORDON D., REYNOLDS A.: Back-to-front display of voxel-based objects. In *IEEE Computer Graphics and Applications* (1985), IEEE Press, pp. 52–60.

[FMS00] FARIAS R., MITCHELL J. S. B., SILVA C. T.: ZSWEEP: an efficient and exact projection algorithm for unstructured volume rendering. In *VVS'00: Proceedings of the 2000 IEEE Symposium on Volume visualization* (New York, NY, USA, 2000), ACM Press, pp. 91–99.

[GHLM05] GOVINDARAJU N. K., HENSON M., LIN M. C., MANOCHA D.: Interactive visibility ordering and transparency computations among geometric primitives in complex environments. In *SI3D'05: Proceedings of the 2005 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2005), ACM Press, pp. 49–56.

[Gie92] GIERTSEN C.: Volume visualization of sparse irregular meshes. *IEEE Comput. Graph. Appl. 12*, 2 (1992), 40–48.

[KH84] KAJIYA J. T., HERZEN B. P. V.: Ray tracing volume densities. In *SIGGRAPH'84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1984), ACM Press, pp. 165–174.

[KM05] KAUFMAN A., MUELLER K.: Overview of volume rendering. *Chapter for The Visualization Handbook* (2005).

[KSE04] KLEIN T., STEGMAIER S., ERTL T.: Hardware-accelerated reconstruction of polygonal isosurface representations on unstructured grids. In *PG'04: Proceedings of the Computer Graphics and Applications, 12th Pacific Conference on (PG'04)* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 186–195.

[KW03] KRUGER J., WESTERMANN R.: Acceleration techniques for GPU-based volume rendering. In *VIS'03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)* (Washington, DC, USA, 2003), IEEE Computer Society, p. 38.

[Lev88] LEVOY M.: Display of surfaces from volume data. In *IEEE Computer Graphics and Applications* (1988), IEEE Press, pp. 29–37.

[LM04] LUM E. B., MA K.-L.: Lighting transfer functions using gradient aligned sampling. In *VIS'04: Proceedings of the conference on Visualization'04* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 289–296.

[MA04] MORELAND K., ANGEL E.: A fast high accuracy volume renderer for unstructured data. In *VVS'04: Proceedings of the 2004 IEEE Symposium on Volume visualization and graphics* (Piscataway, NJ, USA, 2004), IEEE Press, pp. 13–22.

[Max95] MAX N.: Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics 1*, 2 (1995), 99–108.

[MMFE06] MARROQUIM R., MAXIMO A., FARIAS R., ESPERANCA C.: GPU-Based cell projection for interactive volume rendering. In *SIBGRAPI'06: Proceedings of the XIX Brazilian Symposium on Computer Graphics and Image Processing* (Los Alamitos, CA, USA, 2006), IEEE Computer Society, pp. 147–154.

[Pas04] PASCUCCI V.: Isosurface computation made simple: Hardware acceleration, adaptive refinementand tetrahedral stripping. In *Eurographics/IEEE TVCG Symposium on Visualization (VisSym)* (2004), EG/IEEE, pp. 293–300.

[PDC*03] PURCELL T. J., DONNER C., CAMMARANO M., JENSEN H. W., HANRAHAN P.: Photon mapping on programmable graphics hardware. In *HWWS'03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (Aire-la-Ville, Switzerland, 2003), Eurographics Association, pp. 41–50.

[RKE00] ROETTGER S., KRAUS M., ERTL T.: Hardware-accelerated volume and isosurface rendering based on cell-projection. In *VIS'00: Proceedings of the conference on Visualization'00* (Los Alamitos, CA, USA, 2000), IEEE Computer Society Press, pp. 109–116.

[Sab88] SABELLA P.: A rendering algorithm for visualizing 3D scalar fields. In *SIGGRAPH'88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1988), ACM Press, pp. 51–58.

[SBM94] STEIN C., BECKER B., MAX N.: Sorting and hardware assisted rendering for volume visualization. In *1994 Symposium on Volume Visualization* (1994),Kaufman A., Krueger W., (Eds.), pp. 83–90.

[SM97] SILVA C. T., MITCHELL J. S. B.: The lazy sweep ray casting algorithm for rendering irregular grids. *IEEE Transactions on Visualization and Computer Graphics 3*, 2 (1997), 142–157.

[ST90] SHIRLEY P., TUCHMAN A. A.: Polygonal approximation to direct scalar volume rendering. In *Proceedings San Diego Workshop on Volume Visualization, Computer Graphics* (1990), vol. 24(5), pp. 63–70.

[UK88] UPSON C., KEELER M.: V-buffer: visible volume rendering. In *SIGGRAPH'88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1988), ACM Press, pp. 59–64.

[Wes90] WESTOVER L.: Footprint evaluation for volume rendering. In *SIGGRAPH'90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1990), ACM Press, pp. 367–376.

[Wil92] WILLIAMS P. L.: Visibility-ordering meshed polyhedra. *ACM Trans. Graph. 11*, 2 (1992), 103–126.

[WKME03a] WEILER M., KRAUS M., MERZ M., ERTL T.: Hardware-based ray casting for tetrahedral meshes. In *VIS'03: Proceedings of the 14th IEEE conference on Visualization'03* (2003), pp. 333–340.

[WKME03b] WEILER M., KRAUS M., MERZ M., ERTL T.: Hardware-based view-independent cell projection. *IEEE Transactions on Visualization and Computer Graphics 9*, 2 (2003), 163–175.

[WM92] WILLIAMS P. L., MAX N. L.: A volume density optical model. In *1992 Workshop on Volume Visualization* (1992), pp. 61–68.

[WMFC02] WYLIE B., MORELAND K., FISK L. A., CROSSNO P.: Tetrahedral projection using vertex shaders. In *VVS'02: Proceedings of the 2002 IEEE Symposium on Volume visualization and graphics* (Piscataway, NJ, USA, 2002), IEEE Press, pp. 7–12.

[WVW94] WILSON O., VANGELDER A., WILHELMS J.: Direct volume rendering via 3d textures. Tech. rep., University of California at Santa Cruz, Santa Cruz, CA, USA, 1994.