GPU-Based Cell Projection for Interactive Volume Rendering

André Maximo
Ricardo Marroquim
Ricardo Farias (orientador)
Universidade Federal do Rio de Janeiro - UFRJ / COPPE
{andre, ricardo, rfarias}@lcg.ufrj.br

Abstract

In this dissertation is presented a practical approach of the Projected Tetrahedra's (PT) algorithm for interactive volume rendering of unstructured data using programmable graphics cards. Unlike similar works reported earlier, the proposed method employs two fragment shaders, one for computing the tetrahedra projections and another for rendering the volume. The proposed algorithm achieve interactive rates by storing the model in texture memory and avoiding redundant projections of the earlier implementations using vertex shaders. The algorithm is capable of rendering over 2 millions tetrahedra per second on current graphics hardware, making it competitive with recent ray casting approaches, while occupying a substantially smaller memory footprint.

1. Introduction

Volume rendering is a technique used for inspecting volumetric data by extracting meaningful information [15]. Volumetric data contains three-dimensional (3D) information acquired from different sources. There are mainly three source types of volumetric data: *sampling*, *simulation* and *modeling*. For instance, medical images are obtained by sampling. While results of simulations, such as fluid dynamics, are also visualized as volume data. Recently, Computer-Aided Design (CAD) applications have been exploiting volume graphics techniques to generate better models.

Volumetric data acquired from one of these three processes may represent a **scalar** field, for example, density or heat, or a **vector** field, for example, velocity. In general, the volume data can be sparsely located, that is, the scalar or vector values can be at any point in space. This type of volume distribution is called **unstructured** data. The goal of this dissertation is to visualize unstructured scalar fields, where the volume is decomposed in a tetrahedra mesh and the scalar values correspond to its vertices.

Volume rendering can be employed in many important applications, such as inspection of medical images, visualization of geological data and fluid simulation, among other. Earlier methods approximate the volume as *iso-surfaces*, simplifying the rendering process by using only geometric primitives supported by the graphics card. However, the volumetric information is essentially lost. To address this problem the *direct volume rendering* technique was developed to generate two-dimensional (2D) images from 3D volume data.

Volume rendering 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 yielding the final image. For instance, the *cell projection* technique projects each cell on the screen space and composes them in visibility order. In **image-order** methods, for example, *ray casting*, rays are cast from each image pixel through the volume data. In the **domain-based** methods, spatial data is transformed into another domain, for example, *frequency domain*, and the final image is produced directly from this domain.

Recently, new algorithms were developed [34, 33, 4] aiming to exploit the Graphics Processing Units (**GPU**) programmability. GPUs are vectorial processing devices that allow a single code to process multiple data in parallel. With this, common computers can be turned into vectorial machines specifically designed for high performance volume rendering.

The proposal of this dissertation is to create an interactive method for volume rendering, using the cell projection technique and programmable graphics hardware resources. The remainder of this dissertation is organized as follows. In Section 2, related work are presented as well as an overview of volume rendering. The base algorithm of this dissertation is detailed in Section 3. Implementation details of the proposed algorithm are given in Section 4 and their results in Section 5. Finally, conclusions are given in Section 6.

2. Related Work

One of the aspects which makes the area of volume rendering particularly challenging is the fact that 3D information must be efficiently consolidated in a 2D image. For this purpose, volume data represented as scalar fields is usually associated to color values by means of an one-dimensional (1D) function called: **transfer function** [31]. The opacity value computation is based on the *extinction coefficient*, which models the light absorption inside the volume [19]. In the same manner as the color, the extinction coefficient is also associated to the transfer function.

The **volume rendering integral** computes the physical interaction of light rays with the volume [2]. This integral is an equation that evaluates the light color, using the ray's entry and exit values as well as the distance traversed inside each cell (see Figure 1). Williams and Max [37] describe an exact method for computing this equation. However, due to computational limits, it is impractical for interactive applications. Some works [26, 23, 40] improve the performance of their visualization methods by simplifying this integral. The simplification considers the viewing ray instead of the lighting ray, and the integral is computed using three values: the distance length l traversed inside the cell, called **thickness**; the scalar front s_f and back s_b of the ray's entry and exit points.

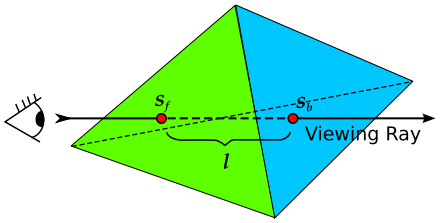

Figure 1. Simplified model of the volume rendering integral.

Many volume rendering algorithms have been proposed in the past. *Ray casting* [14] is perhaps the most common approach and several high-performance implementations of this idea have been developed. Most of these make use of GPUs, such as the *Hardware-Based Ray Casting* (**HARC**) [33, 9]. Traditional *cell projection* algorithms [11, 26] have also been reimplemented using GPU programming [23, 40]. An interesting algorithm which combines ray casting and cell projection is the *View-Independent Cell Projection* (**VICP**) [34] of Weiler et al. Other related approaches include splatting algorithms [35, 4], sweeping [12, 27, 10] and 3D textures techniques [38, 17]. In Section 5, the proposed algorithm is compared to some of these approaches.

The cell projection technique, also called *direct projection*, generates images by projecting the cells on the screen. The projection is determined by converting 3D cells in 2D geometric primitives (see Figure 2). Then, the rasterization process fills the primitives with fragments, which are combined during the composition process yielding the final image.

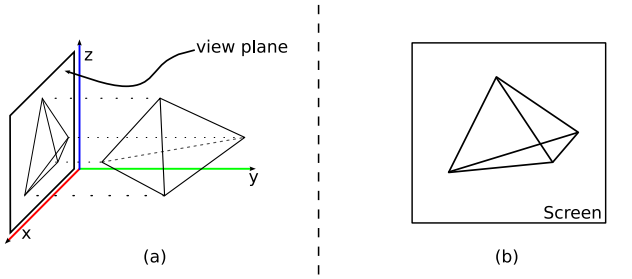

Figure 2. Cell projection technique example. The projection of one volume cell is shown in (a) and the result in (b).

The algorithm proposed on this dissertation is based on the Projected Tetrahedra (**PT**) method introduced by Shirley and Tuchman [26]. The PT algorithm is a cell projection approach where each tetrahedron cell is projected and composed in the final image. An overview of the PT algorithm is given in Section 3.

Wylie et al. [40] developed the *GPU Accelerated Tetrahedra Renderer* (**GATOR**) algorithm, an extension 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 such a way that the same vertex shader can treat all cases. The GATOR algorithm is fast yet redundant, since for each vertex computation, all the tetrahedron vertices must be made available. Another issue of this method is that the final color is poorly approximated.

3. Projected Tetrahedra Algorithm

The *Projected Tetrahedra* algorithm (*PT*) by Shirley and Tuchman [26] was developed in 1990 and became the base of several works since then. The PT algorithm consists of projecting the tetrahedra to the image plane and composing them in visibility order. The first implication is that the 3D data must be *tetrahedralated*. In the case of regular volumes, this can be accomplished by subdividing each hexahedral cell into 5 tetrahedra.

The projected tetrahedra are decomposed into 1 to 4 triangles according to a classification scheme. This classification is based on a comparison of the tetrahe-

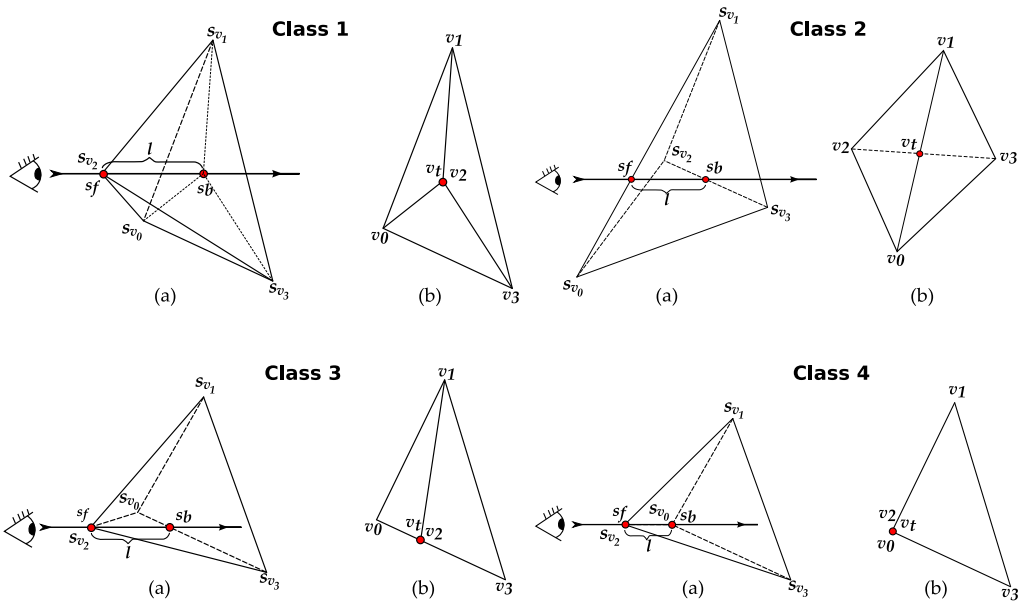

Figure 3. An example of each projection class. The drawing illustrates the tetrahedron and the viewing vector (a) and the projected tetrahedron (b).

dron's face normals with the **viewing vector**. Three different results can be expected from this product $\{+, -, 0\}$ indicating if a face is front facing, back facing, or orthogonal to the viewing vector, respectively. For each of the four different classes a tetrahedron is decomposed into a specific number of triangles (see Figure 3). For example, class 1 projections are decomposed into three triangles while class 2 are decomposed into four.

The **thick** vertex is defined 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 from the intersection between the front and back projected edges, for that reason called *intersection vertex*, while for the other classes it is one of the projected vertices. Thin vertices have the same s_f and s_b values, while for thick vertices these values may have to be interpolated from those of the thin vertices.

Figure 3 depicts one example case for each projection class. In the first case (of class 1), $s_f = s_{v_2}$ and s_b is computed by a trilinear interpolation of s_{v_0}, s_{v_1} and s_{v_3}. In the second case (of class 2), the thick vertex v_t is the intersection of the two interior segments, and the scalar values s_f and s_b are computed by interpolation on segments $\overline{v_0v_1}$ and $\overline{v_2v_3}$, respectively. The third case (of class 3) has $s_f = s_{v_2}$ and s_b is computed by interpolating s_{v_0} and s_{v_3}. In the fourth case (of class 4) s_f and s_b are equal to s_{v_2} and s_{v_0}, respectively.

Before rendering the projected triangles, color values must be assigned to each vertex. A transfer function is used to map the scalar values into chromaticity (*RGB*) and opacity (*A*) values. For a scalar value s, a table is looked up in order to obtain the chromaticity values ($C(s)$) and extinction coefficient ($\tau(s)$), which measure the amount of light absorbed by the cell and is directly associated with the opacity value. 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 is computed by $\alpha = 1 - e^{-\tau_{avg} l}$, where τ_{avg} is the average of $\tau(s_f)$ and $\tau(s_b)$.

Finally, the rendered triangles are rasterized into fragments by interpolating the thick and thin vertices color and opacity values. The fragments are composed in back-to-front order, and, for each new color added to the framebuffer, the new pixel color is computed as $C_{i+1} = \alpha_i C_i + (1 - \alpha_i) C_{i-1}$, where C_{i-1} is the previous color stored in the framebuffer, and C_i and α_i are the interpolated color and opacity values. The new opacity value α_{i+1} is computing by accumulating α_{i-1} with α_i. An illustration of the color composition is shown in Figure 4.

4. Proposed Algorithm

The algorithm is executed mostly in the GPU and is divided in two main parts. In the **first step**, all relevant information of each tetrahedron is computed, that is, the projection class, the thick vertex properties, and the z coordinate

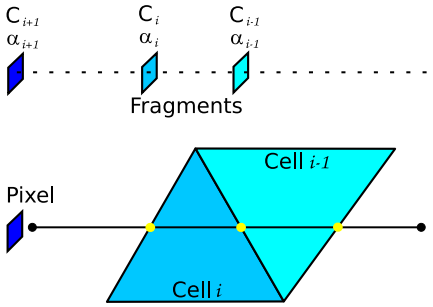

Figure 4. Fragment composition for computing the final pixel color by adding the contribution of each cell in back-to-front order.

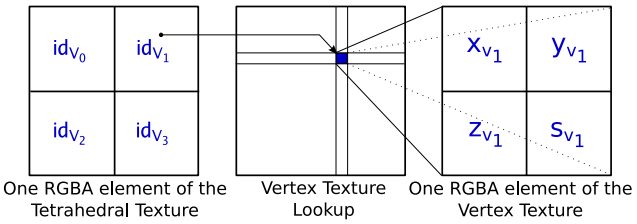

Figure 5. Vertex data retrieval in the first fragment shader. Each texel of the Tetrahedral Texture contains the indices of its four vertices in the Vertex Texture.

of the tetrahedron's centroid. During the **second step**, vertices' scalar values are interpolated to compute chromaticity and opacity values for each fragment.

To increase performance, the proposed algorithm makes 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.

4.1. First Step

The first fragment shader computes the scalar value at the thick vertex, the cell's thickness and centroid, determines the vertex order and number of triangles in the fan. All data used in this shader is stored in GPU memory by creating three different *RGBA* textures: the Tetrahedral Texture, the Vertex Texture and the Classification Texture. The first two textures have 32 bits per component and the third has 8 bits per component. Textures are passed as uniform variables, meaning that they are global constants to all fragments.

The coordinates of each vertex and associated scalar value are stored in the Vertex Texture as a *RGBA* texel: the *RGB* fields store the x, y and z coordinates and the *A* field stores the scalar s.

The Tetrahedral Texture stores one tetrahedron per texel, each of which contains four values which are used for retrieving the four vertices of the tetrahedron from the Vertex Texture, as illustrated in Figure 5. These two texture lookups eliminate the need for vertex attributes and reduce the data transfer overhead from CPU to GPU. Even though this procedure adds the cost of texture fetching, it is still faster than passing attributes.

To execute the fragment shader once per tetrahedron, the Tetrahedral Texture is rendered as 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 [13].

The vertices are then projected to screen coordinates and the projection class is determined by means of four tests described below. This classification process is very similar to Wylie's [40] method, except that degenerate cases are also treated. In addition, the proposed method avoids computational redundancy by performing the tests once per tetrahedron rather than once per vertex.

Each test is an evaluation of a cross product computed with the projected vertices v_0, v_1, v_2 and v_3, shown in Figure 6. The test results are used in a texture lookup operation to determine the class of the projection. The 1D Classification Texture is loaded in the fragment shader and contains a *ternary truth table* (see Table 1) with the different test result permutations. On top of Wylie's 14 cases, the proposed table added 24 class 3 cases and 12 class 4 cases. Each texel thus represents a singular case and contains the correct order to compute the intersection vertex.

$$\begin{aligned} vec1 &= v_1 - v_0 \\ vec2 &= v_2 - v_0 \\ vec3 &= v_3 - v_0 \\ vec4 &= v_1 - v_2 \\ vec5 &= v_1 - v_3 \end{aligned}$$

$$\begin{aligned} test1 &= sign((vec1 \times vec2).z) + 1 \\ test2 &= sign((vec1 \times vec3).z) + 1 \\ test3 &= sign((vec2 \times vec3).z) + 1 \\ test4 &= sign((vec4 \times vec5).z) + 1 \end{aligned}$$

Figure 6. 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.

id_{ttt}	$test_{1234}$	*Case*	*RGBA*
0	0 0 0 0	12	0-3-2-1
1	0 0 0 1	18	0-3-2-1
2	0 0 0 2	6	0-3-2-1
3	0 0 1 0	25	1-0-3-2
4	0 0 1 1	40	2-3-1-0
.	.	.	.
.	.	.	.
.	.	.	.
80	2 2 2 2	11	0-1-2-3

Table 1. Some entries of the ternary truth table used by the proposed algorithm.

The degenerate cases are perceived when one or more of the tests yield a value of 1. If one test returns 1, then it is a class 3 case, if two tests return 1, then it is a class 4 case. For these cases most intersection and interpolation computations can be skipped or replaced by simpler operations. If more than two tests return 1 all four points were projected onto a line, therefore the original model contains degenerate tetrahedra and this projection is discarded. Identifying this degenerated cases is important since not treating them can lead to artifacts in the resulting image.

The fragment data is output by using *multiple render targets* (*MRT*) with two *framebuffer color attachments*. Each attachment is a 2D *RGBA* texture with 32 bits per component. The first output texture contains the intersection vertex coordinates x_{v_i} and y_{v_i} (used only for class 2 tetrahedra), the tetrahedron's centroid z_c and the index of the ternary truth table id_{TTT}. The second output texture contains s_f, s_b, the thickness l and the number of vertices of the triangle fan $count$. This scheme is depicted in Figure 7.

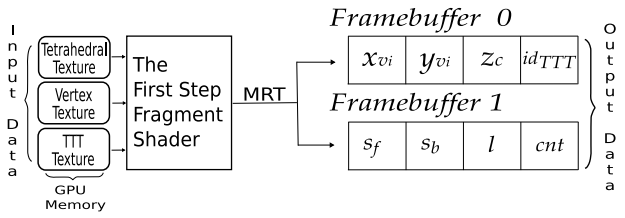

Figure 7. Fragment shader input/output scheme.

4.2. Sort and Setup Arrays

Before rendering the primitives, the data must be retrieved from the output textures, sorted and stored in arrays. The proposed algorithm runs both intermediate steps in CPU, preparing the GPU to the second step.

As mentioned in Section 4.1, the first fragment shader computes the centroids of the tetrahedra and passes these values to a CPU algorithm which sorts them in depth order. In fact, only the z coordinate of each centroid in normalized projected space is used by the sorting algorithm.

Two sorting algorithms are used in this dissertation: a quicker but less precise **bucket sort** is used whenever the viewing transformation is changing, whereas a standard $O(nlogn)$ **merge sort** algorithm is employed for still frames.

The bucket sort is based on slabs perpendicular to the viewing vector, i.e., the tetrahedra are divided into groups (around 20 of them) so that all tetrahedra in a given group have roughly the same depth with respect to the viewer. This means that all tetrahedra within a group may be rendered in any order with little impact on the correctness of the final image. The slabs themselves are visited in back-to-front order as usual. On the other hand, when the user is not manipulating the model, a standard merge sort is used in order to obtain an accurate depth ordering of the centroids.

It must be mentioned, however, that sorting the centroids is not guaranteed to produce 100% correct results in all cases. If this level of accuracy is needed, a more sophisticated approach should be used for ordering the tetrahedra, such as the Williams' MVPO [36] or the k-buffer of Callahan et al. [3].

After the cell sorting, the retrieved information is organized to be rendered with the optimized OpenGL function *glMultiDrawElements*, whose arguments reference global arrays storing vertex information. In particular, the proposed algorithm makes use of two different global arrays for storing vertex coordinates and color values (see Figure 8).

The two 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 and color 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.

Each vertex array element contains the coordinates $\{x, y, z\}$ of the vertex. The color array contains values $\{s_f, s_b, l\}$ for each vertex, rather than actual colors, which will be computed on-the-fly by the second fragment shader. Notice that, for thin vertices, $s_f = s_b$ and $l = 0$.

To manage the correct vertex order of the triangle fans, two additional arrays are needed. The *indices* array is divided into groups, each one with six elements which store the correct vertex order of the fan used to render a tetrahedron. The *count* array contains the number of vertices in each fan. Recall that the maximum number of vertices in a

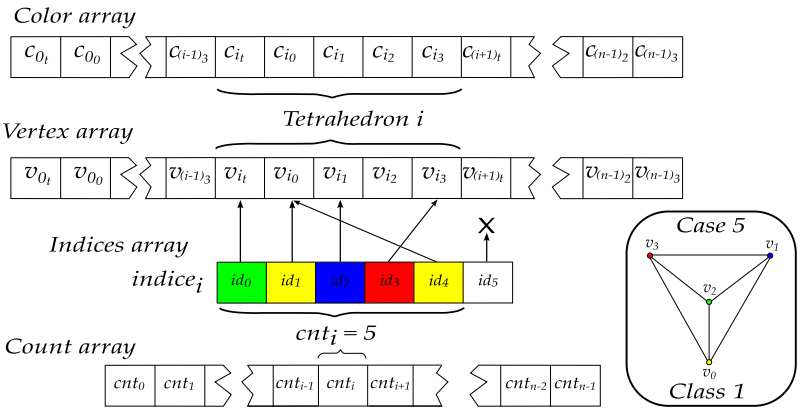

Figure 8. Array data structure. The indices illustrate the case 5 (of class 1), where the correct order to draw tetrahedron i is $v_{i_t} - v_{i_0} - v_{i_1} - v_{i_3} - v_{i_0}$. Note that v_{i_2} is the thick vertex and its coordinates are copied to v_{i_t}.

fan is six (cases of class 2). All the arrays used are updated in CPU by using the data retrieved from the first shader. Note that for the i^{th} primitive, group $indice_i$ is used only up to position cnt_i. See Figure 8 for further details.

4.3. Second Step

When the second shader program is processed, fragments will correspond to linearly interpolated vertex colors. Each final fragment color is computed as described in Section 3 using the values $\{s_f, s_b, l\}$ linearly interpolated from one thick vertex and two thin vertices. An interpolation example is shown in Figure 9.

The transfer function table is passed as a 1D texture to the shader in order to determine the final color and opacity values of each fragment. For the simple average scalar method, where $s = \frac{s_f + s_b}{2}$, the lookup operation using s returns a *RGBA* texel where *RGB* is the final color and A is the extinction coefficient τ.

Rather than estimating the final color using average scalar values, the proposed algorithm uses the partial pre-integration technique (ψ table) of Moreland et al. [21, 20] to achieve better render quality. Since this table does not depend on any attribute of the visualization, it is pre-compiled within the proposed implementation. The $C(s_f)$ and $C(s_b)$ together with the thickness value l are used to compute the indices of the ψ table, stored in a 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 [28, 23, 7]. Moreover, the use of partial pre-integration, instead of pre-integration, allows interactive transfer function editing. Every time the function is updated, the texture is reloaded into GPU memory.

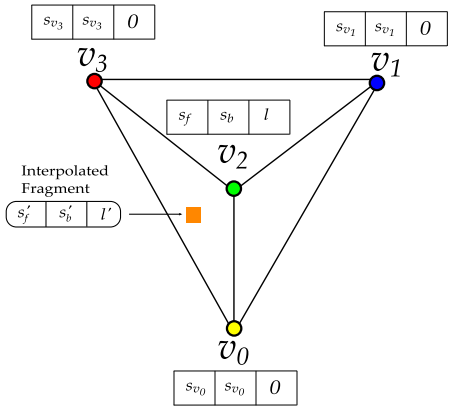

Figure 9. The interpolation values for the case 5 (of class 1) shown in Figure 8. Note that, except for the thick vertex, all others are rendered with the original values of the color array.

To compute the final opacity value, the proposed algorithm makes use of another 1D texture. This texture contains sample values obtained by $\text{Tex1D}(u) = e^{-u}$, for u sampled over interval $[0, 1]$. The lookup is done by passing $u = \tau l$ to obtain the final exponential opacity value. Experimental results indicate that using this 1D texture is slightly faster than computing the exponential function in the fragment shader.

5. Results

The implementation of the proposed algorithm was programmed in C++ using OpenGL 2.0 [25, 39, 24] with GLSL [1] 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 tested data sets were: the Blunt Fin (blunt) and Liquid Oxygen Post (post) from NASA's NAS website [22] and converted to tetrahedra; the Combustion Chamber (comb) from the Visualization Toolkit (Vtk) [16]; the Spx from Lawrence Livermore National Lab [18]; and the Fuel Injection and Brain tomography from [32]. Table 2 further specifies the number of vertices (*# Verts*) and tetrahedra (*# Tet*) for each data set, and the average frames per second (*fps*) and tetrahedra per second (*Tet/s*). Values indicated with K represent thousands and with M millions.

Data set	*# Verts*	*# Tet*	*fps*	*K Tet/s*
Blunt	40 K	187 K	11.30	2119.7
Comb	47 K	215 K	9.32	2005.4
Post	110 K	513 K	4.49	2384.4
Spx	150 K	828 K	3.04	2526.9
Fuel	262 K	1.5 M	1.49	2246.0
Brain	950 K	5.5 M	0.46	2560.8

Table 2. Data set sizes and average times.

The timings are given using a 512^2 pixel viewport and considering that the model is constantly rotating. Table 3 compares the proposed algorithm (PTINT) with others volume rendering algorithms:

- PTINT – The proposed approach: PT with partial pre-integration;
- GATOR – GPU Accelerated Tetrahedra Renderer [40];
- VICP – View-Independent Cell Projection (implemented in GPU and CPU) [34];
- VICP (Balanced) – VICP with GPU-CPU balancing [21, 20];
- HARC – Hardware-Based Ray Casting [33];
- HARC (INT) – HARC with partial pre-integration [8, 9];
- HAVIS – Hardware-Accelerated Volume and Isosurface Rendering Based on Cell-Projection [23];
- HAVS - Hardware Assisted Visibility Sorting [3].

The six tested data sets are show in Figure 14. Another data set rendered with the proposed algorithm is shown in Figure 10.

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*

Table 3. Time comparison between the proposed algorithm (PTINT) and others approaches.

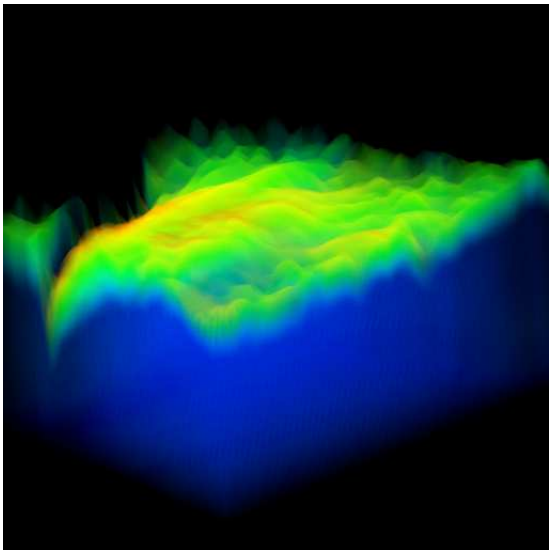

Figure 10. Ocean data set.

It is important to note the differences between the tested data sets. For the Blunt Fin, the proposed 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. Figure 11 illustrates a graph of the average rendering time of the Oxygen Post data set for different screen resolution sizes.

Furthermore, the proposed implementation requires less bytes per tetrahedron than the compared Ray-Casting algorithms, which require storing heavy data structures in GPU memory. The PTINT approach requires 16 bytes per element of the tetrahedral and the vertex textures. Consider-

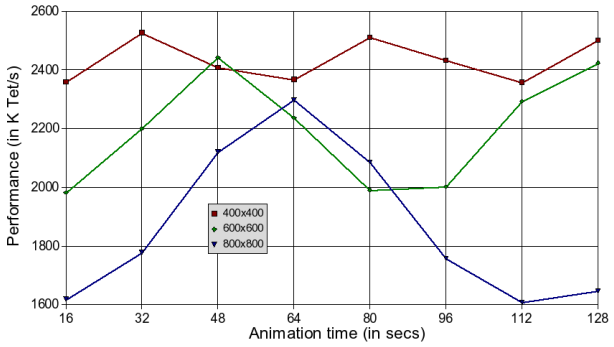

Figure 11. Average performance (in K tet/s) by animation time (in seconds) of the Oxygen Post data set for different resolutions.

ing an average of 4 tet per vertex, the proposed approach spends 20 bytes/tet. This means that one million cells occupy roughly 20 MB of GPU memory. In contrast, HARC (INT) [9] requires 96 bytes/tet, while the original HARC implementation requires 144 bytes/tet.

The interface of the proposed algorithm is shown in Figure 12. The volume rendering information of the Spx data set appears on the screen sides, and includes timing, size and rendering information. Other interface allows the interactive editing of the transfer function (see Figure 13).

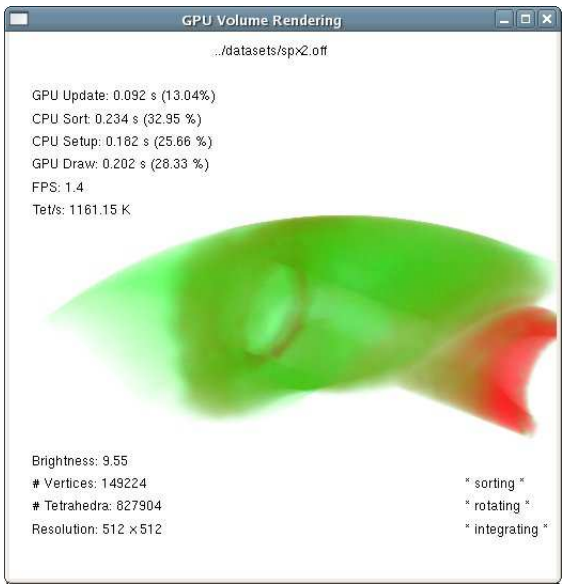

Figure 12. Volume rendering interface showing the Spx data set.

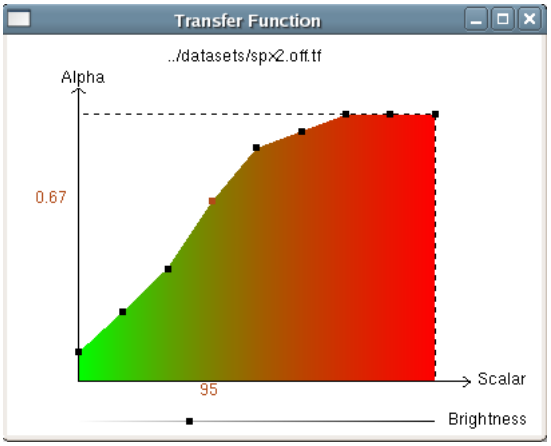

Figure 13. Transfer function editing interface.

6. Conclusions

In this dissertation was presented a cell projection volume rendering method [1] that takes full advantage of modern graphics hardware. The implementation achieves rates over 2.0 M Tet/s with high quality images, and, at the same time, offering interactive control over the transfer function.

The presented algorithm was published in *SIBGRAPI 2006* [29], and later selected to the Computer Graphics Forum [5] journal (to appear in 2007). Two extensions of this work was also accepted in a workshop of high-performance computing [30] in 2006, and in the International Conference on Computer Graphics Theory and Applications (*GRAPP*) [6] in 2007.

Differently from previous PT algorithms, the PTINT method incurs in no major bus transfer overhead since it keeps the whole model stored in the graphics card texture memory. Even though this may represent a limitation for very large models, modern hardware can have up to 1 GB of graphics memory. On the other hand, by not keeping any auxiliary data structure in texture memory, the space allocated per tetrahedron is very low. As a case in point, the proposed method were able to visualize a model with 5.5 million tetrahedra using a fairly limited board with 256 MB.

The visibility ordering remains the true drawback of the algorithm. By using an approximate sorting method during the interaction, the volume can be viewed with no significant visual artifact, as discussed in Section 4.2. In the future, a more precise sorting algorithm can be implemented using shaders, such as the k-buffer of Callahan et al. [3].

1 Supplemental material can be downloaded from `http://www.lcg.ufrj.br/Projetos/volume_render`

It should be noted that current graphics architectures do not allow for manipulating data structures in GPU memory directly. Such facilities are being planned for next generation hardware. This might permit the use of Vertex Buffer Object extensions to render the tetrahedra's data directly to the vertex array, thus eliminating the only major computation currently done in CPU, i.e., the thick vertex data update (computed in the first shader) of the vertex and color arrays.

7. Acknowledgments

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

References

[1] 3Dlabs. OpenGL Shading Language, July 2002. http://developer.3dlabs.com/openGL2/index.htm/.

[2] J. F. Blinn. Light reflection functions for simulation of clouds and dusty surfaces. In *SIGGRAPH '82: Proceedings of the 9th annual conference on Computer graphics and interactive techniques*, pages 21–29, New York, NY, USA, 1982. ACM Press.

[3] S. P. Callahan and J. L. D. Comba. Hardware-assisted visibility sorting for unstructured volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):285–295, 2005. Student Member-Milan Ikits and Member-Claudio T. Silva.

[4] W. Chen, L. Ren, M. Zwicker, and H. Pfister. Hardware-accelerated adaptive ewa volume splatting. In *VIS '04: Proceedings of the conference on Visualization '04*, pages 67–74, Washington, DC, USA, 2004. IEEE Computer Society.

[5] EG. Computer Grafics Forum. http://www.eg.org/EG/Publications/CGF.

[6] EG. International Conference on Computer Graphics Theory and Application, 2007. http://www.grapp.org/grapp2007/index.htm.

[7] K. Engel, M. Kraus, and T. Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 9–16, New York, NY, USA, 2001. ACM Press.

[8] R. Espinha. Visualização interativa de malhas não-estruturadas utilizando placas gráficas programáveis. Master's thesis, Pontifícia Universidade Católica do Rio de Janeiro, Março 2005.

[9] R. Espinha and W. Celes. 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*, page 273. IEEE Computer Society, 2005.

[10] R. Farias, J. S. B. Mitchell, and C. T. Silva. Zsweep: an efficient and exact projection algorithm for unstructured volume rendering. In *VVS '00: Proceedings of the 2000 IEEE Symposium on Volume visualization*, pages 91–99, New York, NY, USA, 2000. ACM Press.

[11] G. Frieder, D. Gordon, and A. Reynolds. Back-to-front display of voxel-based objects. In *IEEE Computer Graphics and Applications*, pages 52–60. IEEE Press, 1985.

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

[13] M. Harris. General-Purpose computation using Graphics Hardware, 2004. http://www.gpgpu.org/.

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

[15] A. Kaufman and K. Mueller. Overview of volume rendering. *Chapter for The Visualization Handbook*, 2005.

[16] Kitware. The Visualization Toolkit VTK, 2006. http://public.kitware.com/VTK/.

[17] J. Kruger and R. Westermann. Acceleration techniques for gpu-based volume rendering. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 38, Washington, DC, USA, 2003. IEEE Computer Society.

[18] LLNL. Lawrence Livermore National Laboratory, 2006. http://www.llnl.gov/.

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

[20] K. Moreland and E. Angel. A fast high accuracy volume renderer for unstructured data. In *VVS '04: Proceedings of the 2004 IEEE Symposium on Volume visualization and graphics*, pages 13–22, Piscataway, NJ, USA, 2004. IEEE Press.

[21] K. D. Moreland. *Fast High Accuracy Volume Rendering*. PhD thesis, The University of New Mexico, Albuquerque, New Mexico, July 2004.

[22] NASA. NASA Advanced Supercomputing (NAS) Division, 2006. http://www.nas.nasa.gov/.

[23] S. Roettger, M. Kraus, and T. Ertl. Hardware-accelerated volume and isosurface rendering based on cell-projection. In *VIS '00: Proceedings of the conference on Visualization '00*, pages 109–116, Los Alamitos, CA, USA, 2000. IEEE Computer Society Press.

[24] R. J. Rost. *OpenGL Shading Language*. Addison Wesley, second edition, 2006.

[25] SGI. OpenGL, 1992. http://www.opengl.org/.

[26] P. Shirley and A. A. Tuchman. Polygonal approximation to direct scalar volume rendering. In *Proceedings San Diego Workshop on Volume Visualization, Computer Graphics*, volume 24(5), pages 63–70, 1990.

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

[28] C. Stein, B. Becker, and N. Max. Sorting and hardware assisted rendering for volume visualization. In A. Kaufman and W. Krueger, editors, *1994 Symposium on Volume Visualization*, pages 83–90, 1994.

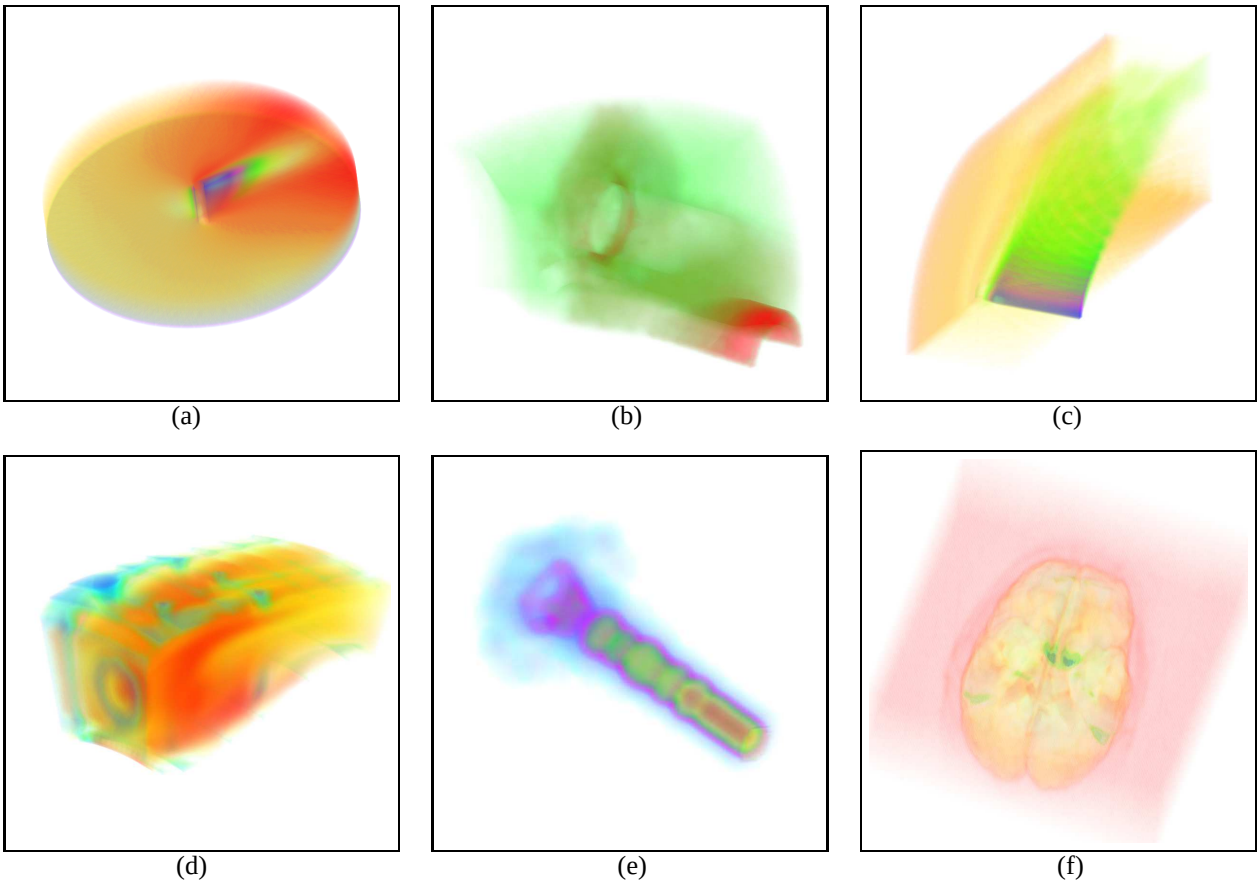

Figure 14. Data sets: Post (a), Spx (b), Blunt (c), Comb (d), Fuel (e), Brain (f).

[29] UFAM. Simpósio Brasileiro de Computação Gráfica, Processamento de Imagens e Visão Computacional, 2006. http://www.sibgrapi.ufam.edu.br/index.php?lang=pt_BR.

[30] UFMG. Simpósio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho, 2006. http://www.sbc.org.br/sbac/2006/index.htm.

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

[32] VolVis. Volume Visualization, 2006. http://www.volvis.org/.

[33] M. Weiler, M. Kraus, M. Merz, and T. Ertl. Hardware-based ray casting for tetrahedral meshes. In *VIS '03: Proceedings of the 14th IEEE conference on Visualization '03*, pages 333–340, 2003.

[34] M. Weiler, M. Kraus, M. Merz, and T. Ertl. Hardware-based view-independent cell projection. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):163–175, 2003.

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

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

[37] P. L. Williams and N. L. Max. A volume density optical model. In *1992 Workshop on Volume Visualization*, pages 61–68, 1992.

[38] O. Wilson, A. VanGelder, and J. Wilhelms. Direct volume rendering via 3d textures. Technical report, University of California at Santa Cruz, Santa Cruz, CA, USA, 1994.

[39] M. Woo, J. Neider, T. Davis, and D. Shreiner. *OpenGL Programming Guide*. Addison Wesley, third edition, 1999.

[40] B. Wylie, K. Moreland, L. A. Fisk, and P. Crossno. Tetrahedral projection using vertex shaders. In *VVS '02: Proceedings of the 2002 IEEE Symposium on Volume visualization and graphics*, pages 7–12, Piscataway, NJ, USA, 2002. IEEE Press.