GPU-Based Cell Projection for Interactive Volume Rendering

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

Abstract

We present a practical approach for implementing the Projected Tetrahedra (PT) algorithm for interactive volume rendering of unstructured data using programmable graphics cards. Unlike similar works reported earlier, our method employs two fragment shaders, one for computing the tetrahedra projections and another for rendering the elements. We achieve interactive rates by storing the model in texture memory and avoiding redundant projections of implementations using vertex shaders. Our algorithm is capable of rendering over 2.0 M Tet/s on current graphics hardware, making it competitive with recent ray-casting approaches, while occupying a substantially smaller memory footprint.

1. Introduction and Related Work

Volume rendering is a technique used for inspecting medical images, geological visualization, fluid simulation, among other applications. The input is a three-dimensional scalar field representing a particular attribute for each point of a given volume, such as density, heat, velocity, etc. The scalar field is usually mapped to color and opacity components by means of a so-called *transfer function*. In this work we are interested in scalar fields presented as tetrahedral decompositions of the volume so that field samples correspond to the mesh vertices.

Many algorithms for direct volume rendering have been proposed in the past, e.g. cell projection, ray casting and sweep-plane. Our approach is based on the Projected Tetrahedra (PT) technique introduced by Shirley and Tuchman [17]. An overview of their method is presented on section 2. Almost at the same time, Max et al. [12] also implemented an early version of the cell projection algorithm. Since then, much work has been applied to improve the quality and performance of the method.

Williams and Max [24] describe an exact optical model for volume rendering. However, due to computational limits, it is impractical for interactive use.

Stein et al. [18] developed a new volume rendering approximation using 2D texture mapping. This method attenuated the artifacts produced by linearly approximating the non-linear opacity effects. These artifacts, called *Mach bands*, were reported first by Max et al. [11].

Röttger et al. [16] improved Stein's results by using 3D textures and generalized it for non-linear transfer function. The textures map the ray-integration of a cell as a function of the ray's entry and exit values, and the distance traveled through a tetrahedron. In a later work, motivated by the relative scarcity of hardware support for 3D textures, Röttger describes an alternative implementation using 2D textures [15]. This idea was further generalized by Engel et al. [3] by proposing a pre-classification technique where the non-linear transfer function is integrated in a pre-processing step.

Lum et al. [10] proposed a faster method to compute the pre-integration. This work also introduced lighting effects that produced more accurate rendering while producing fewer artifacts.

Moreland and Angel [13] introduced the partial pre-integration technique, where the pre-computation is not dependent of the transfer function and thus can be stored for future uses. This not only allows much faster pre-computation, but also allows dynamic changes in the transfer function.

Farias et al. [5] proposed a software memory efficient sweeping algorithm. Even though it is not hardware-based, it is very suitable for parallelization.

The PT algorithm has grown in importance in recent years due to the advance of programmable graphics hardware. Wylie et al. [25] adapted the Shirley and Tuchman algorithm to programmable graphics hardware using the vertex shader. By creating a *basis graph*, he was able to redefine the different projections classes in a manner that one vertex shader could treat them all in the same way. However, the algorithm is very redundant, as each cell projec-

tion computation is repeated five times. Also, an approximate color is used due to limited number of operations supported for vertex programs at that time.

The use of programmable graphics card was further explored by Weiler et al. [22], as an extension of their earlier View Independent Cell Projection algorithm (VICP) [20]. They eliminate Wylie's redundancy problem by scan converting the tetrahedra in hardware. They described both a vertex and a fragment implementation, however, at that time, fragment processors were not sufficiently advanced to fully support their proposal.

Kraus et al. [8] have investigated the removal of rendering artifacts in perspective mode, and improved the accuracy of the tetrahedral coloring by applying a logarithmic scale for the pre-integration table and floating-point color buffers. Much of the artifacts were reduced by using textures with 16 bits per component.

Espinha and Celes [4] propose a new hardware-based ray-casting (HARC) algorithm, introduced by Weiler et al. [21]. They implement the HARC algorithm using partial pre-integration. Their algorithm achieves high quality, fast volume rendering and allows interactive modification of the transfer function. However, as it is based on the ray-casting technique, the memory consumption is high since it needs auxiliary data structures to traverse the tetrahedra.

On the other hand, one drawback of the PT algorithm is the need of a visibility ordering of the cells. This problem has been specifically addressed on other references, e.g. [1, 2, 18], and is not focused in this paper. Our main contribution resides on a high performance realization of the PT algorithm, which is done 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.

Our algorithm is split into two main steps: update and rendering. The first step computes the view-dependent data by projecting the tetrahedra in screen coordinates. The rendering step simply draws the projected tetrahedra by interpolating the scalar values computed for each vertex using a texture-based transfer function as support. Details of the two-step implementation are given in Section 3. Section 4 presents experimental comparison between our implementation and recent algorithms. Finally, in Section 5, conclusions and future work directions are provided.

2. Projected Tetrahedra Algorithm

In a nutshell, the PT algorithm consists in projecting the tetrahedra to screen space and composing them in visibility order. Each projected tetrahedron is decomposed into triangles, and an approximation of the ray integral is used to compute the vertices' color and opacity values. When rasterizing the primitives, an absorption illumination model is used to compute the pixels' color by summing the contributions of every semi-transparent triangle in back-to-front order.

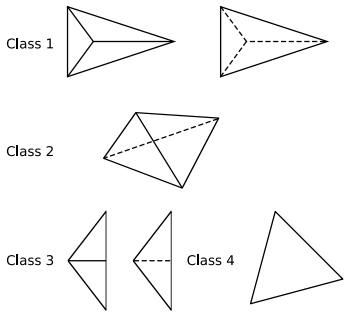

Figure 1. The different classifications of the projected tetrahedra.

The tetrahedra's projected shape is classified into one of four possible cases as depicted in Figure 1. It should be noted that classes 3 and 4 are degenerated cases of classes 1 and 2. Each projected class requires a different decomposition into triangles. For instance, class 1 produces three different triangles while class 2 produces four.

For each projection, the *thick* vertex is defined as the entry or exit point of the ray segment that traverses the maximum distance through the tetrahedron. All other projected vertices are called *thin* vertices. For class 2 projections, the thick vertex is 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 (s_f and s_b) are determined by interpolating the scalars of the thin vertices. The distance traversed by the ray segment is the thickness l of the cell.

Referring to the volume density optical model proposed by Williams and Max [24], the color and opacity at each fragment are computed by interpolating the values s_f, s_b and l of the triangle's vertices. For each fragment, the color is computed by evaluating the chromaticity and extinction coefficient (τ) of the transfer function at the average interpolated value $\frac{s_f+s_b}{2}$. The opacity, on the other hand, is estimated by

$$\alpha = 1 - e^{-\tau l}. \tag{1}$$

The opacity at the thin vertices, for example, is zero since the thickness at these points is also zero (the ray does not traverse any distance).

Finally, the pixels are composited in back-to-front order, and, for each new color added to the frame buffer, the new final color is computed by:

$$I_{new} = Color + (1 - Alpha) \times I, \tag{2}$$

where I is the previous color stored in the frame buffer, and $Color$ and $Alpha$ are the interpolated scalar and thickness values.

3. A Two-GPU Pass Approach

Our algorithm is divided in two main parts and, consequently, in two different shaders. In the first step, all data is processed per tetrahedron, that is, the projection class, the thick vertex properties, and the z coordinate of the tetrahedron's centroid are computed. The second step interpolates the vertices' scalar values to compute color and opacity values for each fragment.

To speed up the rasterization process, we make use of vertex arrays and the primitives are drawn as triangle fans. To draw each fan correctly the order and number of vertices must be determined in the first step and passed on to the second, as described later.

3.1. Projecting the Tetrahedra

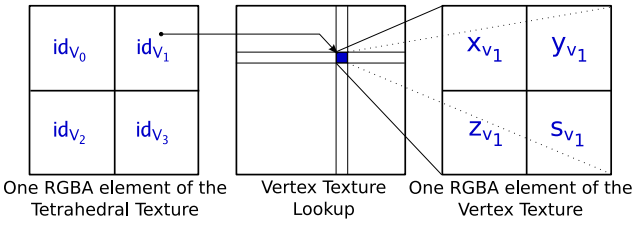

Figure 2. Vertex data retrieval in the first fragment shader. Each texel of the Tetrahedral Texture contains the indices of its four vertices in the Vertex Texture.

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 2. These two texture lookups eliminate the need for vertex attributes and reduce the data transfer overhead from CPU to GPU. Even though we add 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 [6].

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 [25] 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 is an evaluation of a cross product computed with the projected vertices v_0, v_1, v_2, v_3:

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

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

Figure 3. 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 test results are used in a texture lookup operation to determine the class of the projection. The 1-dimensional Classification Texture is loaded in the fragment shader and contains a ternary truth table with the different test result permutations. On top of Wylie's 14 cases, we have 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. Since it is a ternary truth table, there are three different possible results for each test:

$$test_i = \begin{cases} 0 & \text{if cross.z } \leq 0 \\ 1 & \text{if cross.z } = 0 \\ 2 & \text{if cross.z } \geq 0 \end{cases} \quad (3)$$

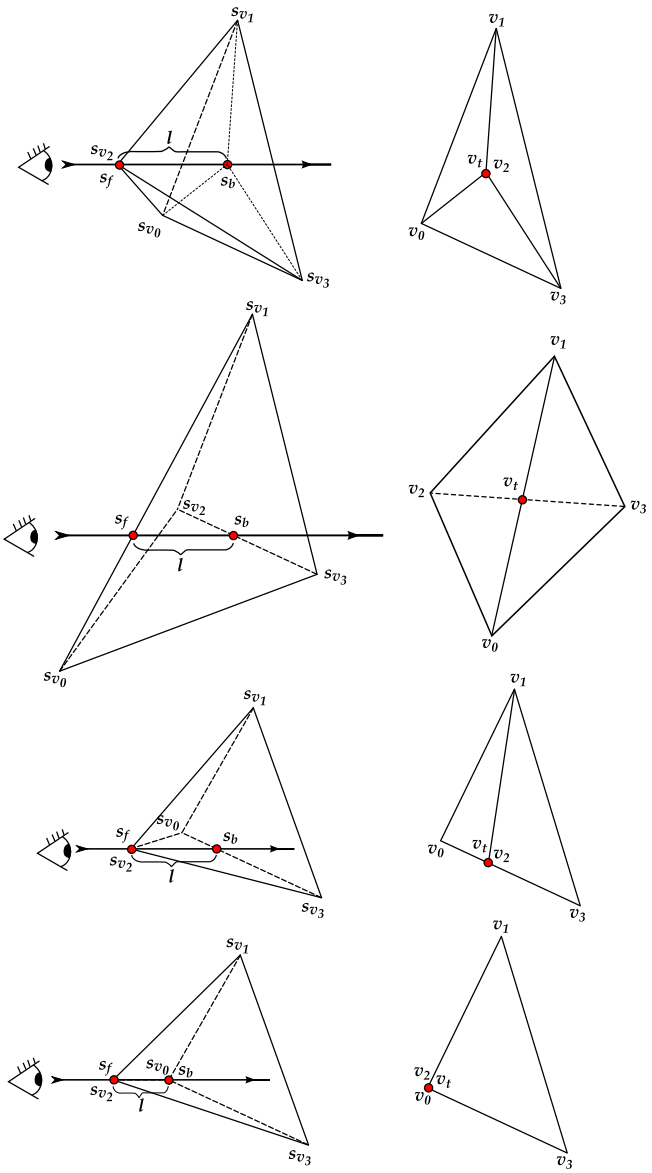

Figure 4. One example for each projection class. From top to bottom: classes 1, 2, 3, and 4. The left column illustrates the tetrahedron in 3D space and the viewing ray intersecting it at the thick vertex v_t. The scalar value is defined as s_{v_i} for each projected vertex v_i, while s_f and s_b are the values at the ray's entry and exit points, respectively. On the right, the projected tetrahedron.

Figure 4 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.

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 5.

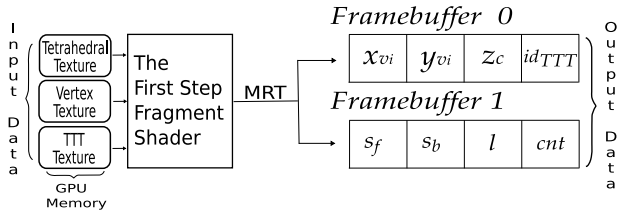

Figure 5. Fragment shader input/output scheme, where TTT is the Ternary Truth Table.

3.2. Sorting

One of the bottlenecks of the Cell Projection algorithm is the need to sort the cells in visibility order before rendering. As mentioned in Section 3.1, 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 our implementation: a quicker but less precise bucket sort is used whenever

the viewing transformation is changing, whereas a standard $O(n\log n)$ 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. Experimental results indicate that this procedure introduces distortions in the animation which are not perceived by the naked eye.

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 [23] or the k-buffer of Callahan et al. [1].

3.3. Rendering the Primitives

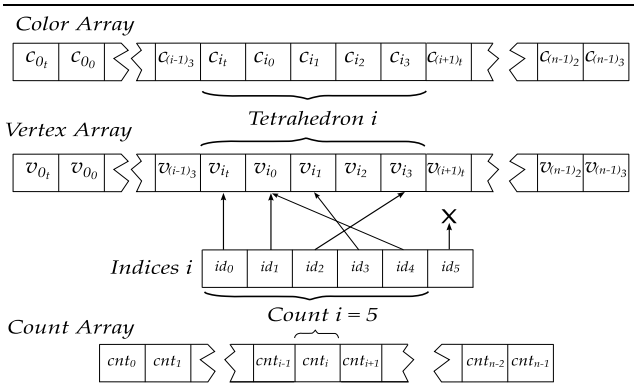

Figure 6. Array data structure. The indices illustrate a class 1 case, where the correct order to draw tetrahedron i is $v_{i_t} - v_{i_0} - v_{i_3} - v_{i_1} - v_{i_0}$. Note that v_{i_2} is the thick vertex and its coordinates are copied to v_{i_t}.

The second fragment shader renders the triangle fans with the optimized OpenGL function *glMultiDrawElements*. This function has an index and a count array as arguments which reference global arrays storing vertex coordinates and a color values (see Figure 6).

The vertex and color arrays are grouped in five elements per tetrahedron: the thick vertex plus the four original vertices. These two arrays are mostly constants, for each change in the view direction only the position and color of the thick vertices are updated.

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 6 elements which store the correct vertex order of the fan used to render a tetrahedron. The *cnt* array contains the number of vertices in each fan. Recall that the maximum number of vertices in a 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 $indices_i$ is used only up to position cnt_i. See Figure 6 for further details.

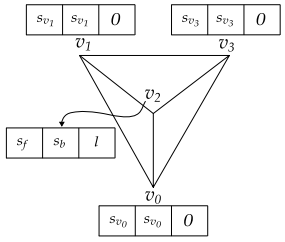

Figure 7. The interpolation values for the class 1 case shown in Figure 6. Note that, except for the thick vertex, all others are rendered with the original values of the color array.

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 2 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 7.

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.

To compute the final opacity value we make use of another 1D texture. This texture contains sample values obtained with Equation 1, that is, 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.

Our experiments show that using this 1D texture is slightly faster than computing the exponential function in the fragment shader.

3.4. Partial Pre-Integration

In order to improve the ray integration approximation, and thus the image quality, the average scalar method is replaced by the partial pre-integration approach [13].

In the pre-integration method, the ray integral is computed for different values of $\{s_f, s_b, l\}$ and stored in a table. These values are used to determine the color of the fragment by performing a single lookup operation. However, since the pre-integration table is computed using the transfer function, every time it changes the table must be recomputed. Depending on the resolution of the table this can be a expensive operation.

The partial pre-integration approach overcomes this disadvantage by computing a table which is independent of the transfer function. This 2D ψ table was computed by Moreland et al. [13] and made available for downloading. Since it does not depend on any attribute of the visualization, it is pre-compiled within our implementation.

In the second fragment shader, the RGBA colors associated with s_f and s_b are retrieved from the transfer function, and, together with the thickness value, are used to compute the indices of the ψ table. The values retrieved from this table are then used to compute the final fragment color.

The partial method is understandably slower than merely using the average value of s as per the original PT method of Shirley and Tuchman. This decreased performance is acceptable, however, since it is less prone to under-sampling errors [4]. Moreover, with partial pre-integration, instead of full pre-integration, we are able to change the transfer function interactively. Every time the function is modified by the user, the lookup texture needs to be reloaded, but this can be done with very little overhead.

4. Results

In the previous sections, we have described an implementation of the PT algorithm using fragment shaders. 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: the Blunt Fin (blunt) and Liquid Oxygen Post (post) from NASA's NAS website [14] and converted to tetrahedra; the Combustion Chamber (comb) from the Visualization Toolkit (Vtk) [7]; the Spx from Lawrence Livermore National Lab [9]; and the Fuel Injection and Brain tomography from [19]. Table 1 further specifies the number of vertices and tetrahedra for each data set, and the average fps and tet/sec. Six models rendered with our algorithm are shown in Figure 9.

$Dataset$	$\#\,Vertices$	$\#\,Tet$	fps	$K\ tet/s$
Spx	2.9 K	13 K	95.32	1233.2
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
Spx2	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 1. Data set sizes and average times. Two different Spx data sets were used. Note that models with few elements have lower tet/s due to the overhead of changing between shaders many times in one second.

$Algorithm/Dataset$	$Blunt$	$Post$
GPU CP for Int. VR	11.30 fps	4.49 fps
GATOR	4.07 fps	1.51 fps
HARC	4.47 fps	8.63 fps
HARC PPI	4.94 fps	5.93 fps
VICP	5.20 fps	1.93 fps

Table 2. Time comparison of different volume rendering algorithms.

The timings are given in tetrahedra per second (tet/s) using a 512^2 pixel viewport and considering that the model is constantly rotating. Table 2 compares our algorithm (GPU-Based CP for Int. Vol. Rend.) with the following algorithms: GATOR – GPU Accelerated Tetrahedra Renderer [25], VICP – View-Independent Cell Projection (in GPU) [22], HARC – Hardware-Based Ray-Casting [21] and HARC PPI – HARC using Partial Pre-Integration [4]. Figure 8 illustrates a graph of the average rendering time of the Oxygen Post data set for different screen resolution sizes.

Furthermore, our implementation requires less bytes per tetrahedron than the compared Ray-Casting algorithms, which require storing heavy data structures in GPU memory. Our approach requires 16 bytes per element of the tetrahedral and the vertex textures. Considering an average of 4 tet per vertex, we have 20 bytes/tet. This means that one million cells occupy roughly 20 MB of GPU memory. In contrast, HARC PPI [4] requires 96 bytes/tet, while the original HARC implementation requires 144 bytes/tet.

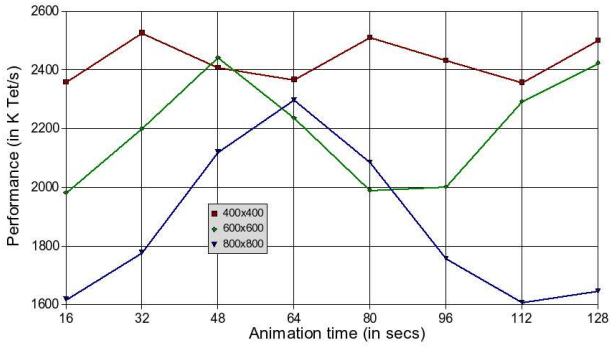

Figure 8. Average performance (in K tet/s) by animation time (in seconds) of the Oxygen Post data set for different resolutions.

5. Conclusions and Future Work

We have presented a Cell Projection Volume Rendering method that takes full advantage of modern graphics hardware. The implementation [1] achieves rates over 2.0 M Tet/s with high quality images, and, at the same time offers interactive control over the transfer function.

Differently from previous PT algorithms, our 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, we were able to visualize a model with 5.5 million tetrahedra using a fairly limited board with 256 MB of graphics memory.

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 perceptually significant visual artifact, as discussed in Section 3.2. In the future, we plan to implement a more precise sorting algorithm using shaders, such as the k-buffer of Callahan et al. [1].

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.

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

6. Acknowledgments

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

References

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

[2] J. Comba, J. T. Klosowski, N. L. Max, J. S. B. Mitchell, C. T. Silva, and P. L. Williams. Fast polyhedral cell sorting for interactive rendering of unstructured grids. *Computer Graphics Forum*, 18(3):369–376, 1999.

[3] 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.

[4] 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.

[5] 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.

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

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

[8] M. Kraus, W. Qiao, and D. S. Ebert. Projecting tetrahedra without rendering artifacts. In *VIS '04: Proceedings of the conference on Visualization '04*, pages 27–34, Washington, DC, USA, 2004. IEEE Computer Society.

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

[10] E. Lum, B. Wilson, and K.-L. Ma. High-quality lighting and efficient pre-integration for volume rendering. The Joint Eurographics-IEEE TVCG Symposium on Visualization 2004, 2004.

[11] N. Max, B. Becker, and R. Crawfis. Flow volumes for interactive vector field visualization. In *VIS '93: Proceedings of the 4th conference on Visualization '93*, pages 19–24, 1993.

[12] N. Max, P. Hanrahan, and R. Crawfis. Area and volume coherence for efficient visualization of 3d scalar functions. In *VVS '90: Proceedings of the 1990 workshop on Volume visualization*, pages 27–33, New York, NY, USA, 1990. ACM Press.

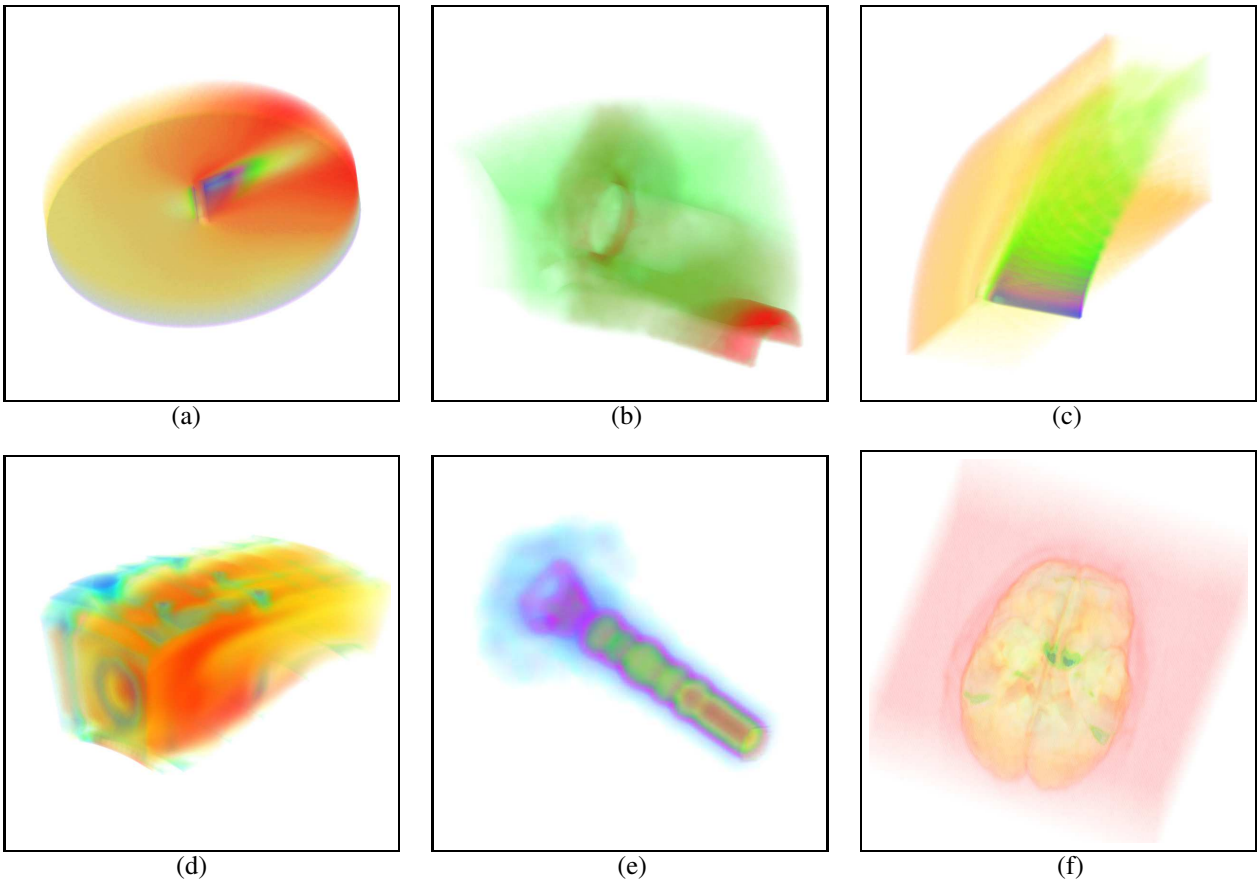

Figure 9. Datasets : Post (a), Spx (b), Blunt (c), Comb (d), Fuel (e), Brain (f).

[13] 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, 2002. IEEE Press.

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

[15] S. Röettger and T. Ertl. A two-step approach for interactive pre-integrated volume rendering of unstructured grids. In *VVS '02: Proceedings of the 2002 IEEE Symposium on Volume visualization and graphics*, pages 23–28, Piscataway, NJ, USA, 2002. IEEE Press.

[16] S. Röttger, 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.

[17] 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.

[18] 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.

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

[20] M. Weiler, M. Kraus, , and T. Ertl. Hardware-based view-independent cell projection. In *VVS '02: Proceedings of the 2002 IEEE Symposium on Volume visualization and graphics*, pages 13–22, Piscataway, NJ, USA, 2002. IEEE Press.

[21] 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 '93*, pages 333–340, 2003.

[22] 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.

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

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

[25] 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.