Eurographics/ IEEE-VGTC Symposium on Visualization 2010
G. Melançon, T. Munzner, and D. Weiskopf
(Guest Editors)

Volume 29 (2010), Number 3

Hardware-Assisted Projected Tetrahedra

A. Maximo[1] and R. Marroquim[1] and R. Farias[1]

[1]COPPE, Federal University of Rio de Janeiro, Brazil

Abstract

We present a flexible and highly efficient hardware-assisted volume renderer grounded on the original Projected Tetrahedra (PT) algorithm. Unlike recent similar approaches, our method is exclusively based on the rasterization of simple geometric primitives and takes full advantage of graphics hardware. Both vertex and geometry shaders are used to compute the tetrahedral projection, while the volume ray integral is evaluated in a fragment shader; hence, volume rendering is performed entirely on the GPU within a single pass through the pipeline. We apply a CUDA-based visibility ordering achieving rendering and sorting performance of over 6 M Tet/s for unstructured datasets. Furthermore, as each tetrahedron is processed independently, we employ a data-parallel solution which is neither bound by GPU memory size nor does it rely on auxiliary volume information. In addition, iso-surfaces can be readily extracted during the rendering process, and time-varying data are handled without extra burden.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Viewing algorithms; I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures and data types.

1. Introduction

Volume rendering has been widely used in a variety of fields, such as visualization of geological data, fluid simulation and inspection of medical images. The main objective is to obtain a better insight of the volume data either by rendering iso-surfaces, i.e. *indirect volume rendering*, or by rendering the volume as a semi-transparent material, i.e. *direct volume rendering*. In this work we are interested in direct and indirect volume rendering of datasets defined over tetrahedral meshes (see an example of both renderings of a fluid simulation dataset in Figure 1).

The main data sources for volume rendering applications are numerical simulations of natural phenomena, e.g. Computational Fluid Dynamics (CFD), and measurement devices such as Magnetic Resonance Imaging (MRI) and Seismic Tomography. Generally, measurement devices produce regular volume data, while simulations yield both regular and irregular data.

Volume rendering of irregular structures has mainly focused on iso-surface rendering [TPG99, RKE00, KSE04, Pas04], e.g. numerical simulations for fluid dynamics and

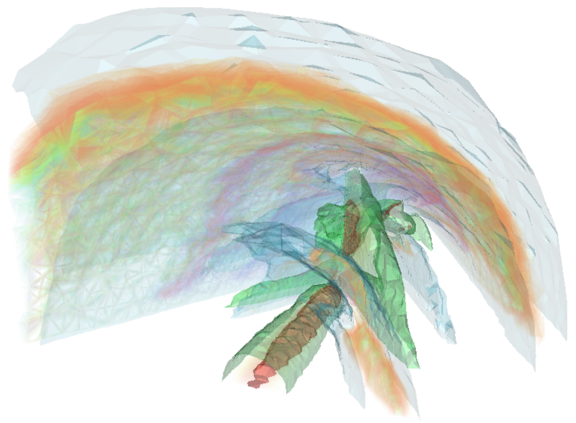

Figure 1: *The Fighter dataset rendered with our algorithm – Hardware-Assisted Projected Tetrahedra (HAPT) – using direct and indirect volume rendering.*

tensor fields in geosciences [Zeh06]. Even so, direct volume rendering is also employed in important applications such as the visualization of geodynamics phenomena of mantle convection [CZY*08].

In this paper we present *HAPT – Hardware-Assisted Projected Tetrahedra* – an irregular volume rendering method based on the original PT algorithm [ST90] and completely developed to exploit programmable graphics hardware. We address the visibility ordering problem using a fast quicksort algorithm [CT08] adapted to our scenario and implemented on nVidia's Compute Unified Device Architecture (CUDA) [NVI07a]. Once the cells are sorted, our approach uses vertex, geometry and fragment shaders to perform the original PT algorithm entirely in a single GPU pass, while taking full advantage of processors dedicated to triangle rasterization. In this way, HAPT has four main features: first, it performs volume rendering of irregular structures in a single rendering pass after sorting; second, it consumes almost no GPU memory since the tetrahedra are streamed to the graphics card; third, in addition to direct volume rendering, isosurfaces can be extracted and rendered on-the-fly, and time-varying data are also easily handled; and finally, HAPT's framework allows for easy exchange of its modules, such as sorting, outside the rendering pipeline, or volume integration methods, inside the pipeline, providing greater implementation flexibility.

The remainder of this paper is organized as follows. In Section 2, we discuss related work on different volume rendering techniques. We present our method in Section 3 and its results in Section 4. In Section 5, a comparative discussion between our method and other approaches is laid out. Finally, in Section 6, we conclude our work.

2. Related Work

Cell projection is a simple and efficient method widely used for direct volume rendering of irregular meshes [WMFC02, KQE04, SCT06, MMFE06, MMFE07, MMFE08]. One of the most popular cell-projection technique is the Projected Tetrahedra (PT) algorithm introduced by Shirley and Tuchman [ST90]. The PT method subdivides a tetrahedron projection into four classes, where for each the tetrahedron cell is decomposed into a fixed number of triangles in order to benefit from the graphic card's rasterization pipeline. However, due to early hardware limitations, subsequent efforts to map it to the GPU failed to achieve this premise and had to resort to auxiliary data structures or multi-pass strategies.

Briefly, PT consists of breaking up each tetrahedron cell into a set of triangles, and sending them for rendering in a back-to-front order. The tetrahedron is classified depending on the shape of the projection, and the classification decomposes the projection in one to four triangles. Figure 2 shows an example of a class 1 projection; in this case, vertex v'_3, i.e. vertex v_3 projected, falls inside the projected face $\{v'_0 v'_1 v'_2\}$ generating three triangles for rasterization.

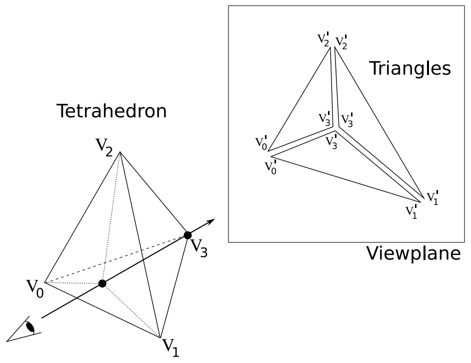

Figure 2: *One example of class 1 projection of the PT algorithm, where three triangles are generated.*

Before rendering the triangles, scalar values for the ray's entry and exit points, as well as its traversed distance inside the tetrahedron, are computed for each vertex. These values are interpolated during rasterization, making it possible to compute an approximation of the volume ray integral using regular triangle rasterization. For further details we refer the reader to the original paper [ST90].

Wylie *et al.* [WMFC02] presented the first GPU adaptation of the PT algorithm using a vertex shader, dubbed *GPU Accelerated Tetrahedra Renderer* (GATOR). In their approach, a basis graph is employed to perform triangle decomposition on the GPU in such a way that all cases are treated by the same vertex shader. The main advantage of this approach is the creation of a graphics primitive to handle tetrahedra. However, each tetrahedron projection has to be computed five times, since the projections are performed inside the vertex shader and the basis graph has five vertices.

Both GATOR and the original PT algorithm share the issue of poorly approximating the final pixel color by averaging the entry and exit scalar values. Quality can be greatly improved by evaluating the physical interaction of light rays with the volume through integration [WM92]. One way to avoid an expensive integral computation during rendering is to pre-compute and store the integration values in a *pre-integration* table. Kraus *et al.* [KQE04] presented an artifact-free PT rendering in perspective mode by applying a logarithmic scale for the pre-integration table. Another work aiming to improve this method is the *partial pre-integration* technique introduced by Moreland and Angel [MA04], where the pre-computation of the integral equation does not depend on the transfer function.

Marroquim *et al.* [MMFE06, MMFE08] further improved the GATOR method by removing the redundancy of their basis graph and adding a better approximation of the final color. Their algorithm, called *Projected Tetrahedra with Partial Pre-Integration* (PTINT), relies on a two-pass GPU approach. In the first step the tetrahedra projections are computed in the GPU and brought back to the CPU for sorting, and in the second step the triangles resulting from the projections are rasterized. The PT classification scheme is done in the first step using a truth table, as in GATOR. However, the PTINT method extended this classification table, naming it the *ternary truth table*, by considering all possible projection permutations, and thus treating also the degenerated cases. The final color computation is improved by employing the partial pre-integration technique, in contrast to the average scalar method of PT and GATOR. The main problem of the PTINT approach is a high cost to read data back from the GPU, imposed by the two-pass technique. In this paper we introduce a single-pass GPU method, removing both the need of reading back from the GPU of the PTINT approach, and the auxiliary basis graph structure of GATOR.

Another common approach for volume rendering is ray casting. In this image-order strategy, for each pixel, the volume ray integral is computed by casting a ray through the volume, usually in a front-to-back order. Efficient ray-casting algorithms can also be achieved using GPUs, such as the *Hardware-Based Ray-Casting* algorithm (HARC) [WKME03]. Here, a multi-pass technique is employed, where for each pass a fragment shader computes the ray integral for a single traversal. Intermediate results are written to texture targets and read in subsequent rendering passes to update the ray information. The write-to-buffer operations avoid moving information back to the CPU eliminating most data transfer overhead. Unfortunately, it consumes large amounts of GPU memory to store adjacency information, constraining the maximum allowed size of working data. Espinha and Celes [EC05] further improved the original HARC algorithm reducing memory consumption and employing the partial pre-integration technique. Nonetheless, memory requirements can still be prohibitive for large volumes and time-varying datasets.

Since direct volume rendering algorithms involve compositing, they depend on a correct traversal order for a given viewpoint. On the one hand, ray-casting algorithms employ an adjacency graph in such way that when a ray leaves a cell it has enough information to find the next one; on the other hand, cell-projection algorithms usually compute an approximate ordering in object space. Even though there are methods for exact ordering of cells [SMW98, KE01, CMSW04], they are quite complex and time consuming. A first approach aiming to combine the better of cell-projection and ray-casting techniques is the *View-Independent Cell Projection* (VICP) of Weiler *et al.* [WKE02]. By performing ray casting only inside each projected cell, the VICP achieves high quality image while consuming less memory than HARC. Callahan *et al.* [CICS05] present an approximate sorting approach named *Hardware-Assisted Visibility Sorting* (HAVS), which uses a hybrid volume rendering strategy such as the VICP method. First, volume faces are sorted in object-space by their centroids in the CPU and rendered using regular triangle rasterization. Next, volume ray integration is evaluated in image space, while a refined sorting method is carried out using their *k-buffer* technique in the GPU, where k determines the sorting precision, balancing between performance and quality. One disadvantage is that by embedding sorting with rendering it limits HAVS to a fixed framework, prohibiting the algorithm to use an exact ordering technique.

3. Hardware-Assisted Projected Tetrahedra

HAPT's framework is presented in Figure 3. First the visibility order is computed using any CPU or GPU method. The ordered tetrahedra are streamed to the graphics pipeline through the vertex shader (VS), and decomposed into triangles in the geometry shader (GS). The triangle primitives are sent down the pipeline with scalar values and traversal length as color attributes for the direct volume rendering (DVR) technique, and face normals for the iso-surface rendering (ISO) technique. This process brings a great benefit highly desirable for a hardware-based approach: fitting a volumetric primitive that is well supported by graphics hardware. Finally, we use a fragment shader (FS) to evaluate the volume ray integral. Note that, after sorting, the whole rendering algorithm is performed in a single pass, performing the original PT technique entirely on the GPU, while taking full advantage of triangle rasterization dedicated processors.

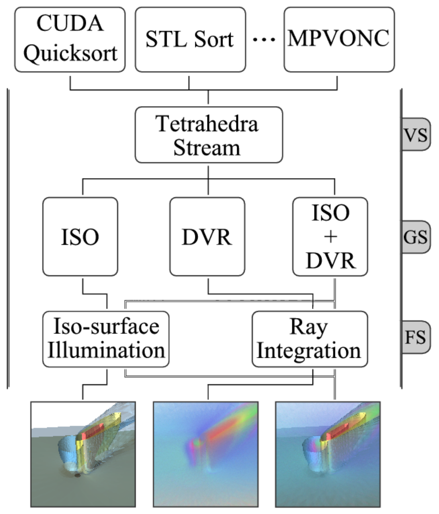

Figure 3: *HAPT's Framework divided into vertex (VS), geometry (GS) and fragment (FS) shaders. Any sorting method can be used prior to rendering, for example: quicksort in CUDA; STL-based introsort on the CPU; or the MPVONC algorithm on the CPU (see Section 3.1 for details).*

The rendering pipeline is depicted in Figure 4. An important point about the data flow is that, since each tetrahedron is processed independently and in a parallel fashion, it fits perfectly within the streaming paradigm. In addition, no auxiliary volume data structure needs to be accessed during

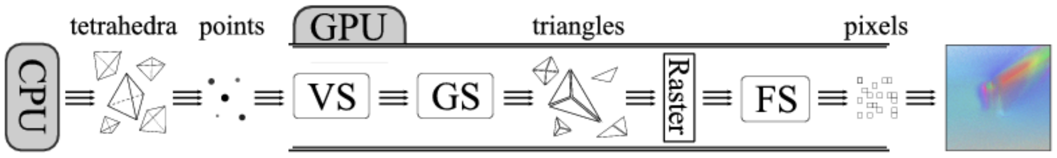

Figure 4: *HAPT's pipeline: sorted tetrahedra are streamed as point primitives to the GPU; decomposed into triangles in the geometry shader (GS); and finally, during rasterization, the ray integral is computed per fragment to compose the final image.*

rendering as each primitive contains all information required for processing. This is specially important because it reduces GPU data storage in such a way that there is no restriction to render a volume due to GPU memory requirements. The graphics card memory is limited when compared to the CPU memory, for instance a volume with several million tetrahedra can be rendered by a ray-casting or cell-projection approach on the CPU, but fails to be stored on the GPU when using the HARC or PTINT methods.

The streaming feature of HAPT allows it to handle time-varying data trivially as a sequence of static volumes per frame (see Section 3.5 for details). Moreover, it minimizes the data fetching latency, which usually imposes a high transfer overhead and, consequently, significantly decreases the algorithm's performance. In the next sections each feature and step of the algorithm is presented.

3.1. Sorting

We have tested our algorithm with four different sorting methods for comparison: the STL-based introsort [PLMS00] on the CPU; the MPVONC [Wil92], *Meshed Polyhedra Visibility Ordering for Non-Convex* meshes, on the CPU; the bitonic sort on the GPU using CUDA [NVI07b]; and the quicksort on the GPU using CUDA [CT08]. Except for the MPVONC, the other three strategies perform approximate sort using the tetrahedra centroids.

For the CUDA-based algorithms, the ordered tetrahedron indices are written directly into a GPU *data buffer* or *vertex buffer object* (VBO), avoiding the transfer back to the CPU. In case the data is too large to fit in a VBO, any of the methods can read back to the CPU and stream the tetrahedra down the pipeline. Also, since the bitonic CUDA sorting works only with power-of-two arrays, we enlarge the indices buffer to fit the nearest corresponding size, making it slower than the CUDA quicksort counterpart.

For the exact ordering algorithm on the CPU, i.e. MPVONC, it is important to note that it needs two auxiliary data structures, the adjacency list and the precomputed face normals; this extra information can increase the memory consumption on the CPU by around four times the volume size.

There are some degenerated cases where the MPVONC method does not perform a correct visibility ordering, notwithstanding, it works for most meshes [KQE04]. On the other hand, the centroid sorting usually introduces an error, where in some cases the ordering of adjacent tetrahedra can be inverted. Fortunately, this error is still very low. As a matter of fact, we noticed no visual difference from the MPVONC, let alone any artifacts.

To support this last statement, we have run a series of tests to estimate the error of centroid sorting in regards to the MPVONC. Table 1 presents the average and maximum error for various datasets, described in Section 4. The errors are computed per color channel separately, therefore the second column (Max. Error) is the maximum difference between all channels of all corresponding pixels. The last column (Diff. Pixels) gives the percentage of different pixels, that is, pixels between images that are not an exact match in all three RGB channels. For all statistics, only pixels with error above zero are taken into account, thus correct and background pixels are not included. An error of approximately 1.2%, is equivalent to a difference of 3 units in the RGB domain [0, 255] for one specific color channel of a given pixel. Usually the average error is approximately of one single unit, becoming imperceptible for visualization purposes.

The statistics for Table 1 were gathered by rendering with direct volume rendering using both methods (centroid sorting and MPVONC) from at least 100 different viewpoints sampled over a sphere. It is also important to notice that the numbers may vary with different transfer functions, even so, we have noticed no discrepancies from the presented values. In this manner, the centroid method is still a good alternative for accelerating rendering speed and cutting down on memory space while introducing a very low error. Since for static data the error is already visually imperceptible even when there are many different pixels, this method is even more adequate for interactively visualization of time-varying data.

Dataset	Max. Error	Avg. Error	Diff. Pixels
blunt	1.961%	0.4069%	6.04%
post	2.353%	0.4245%	33.13%
spx2	1.569%	0.3985%	8.13%
delta	5.098%	0.5895%	14.25%
torso	1.176%	0.3933%	1.51%
fighter	1.569%	0.3943%	2.02%

Table 1: *Error between centroid sorting and MPVONC.*

3.2. Projection

To simplify the data throughput to the GPU, a tetrahedron is sent as a vertex with three attributes, that is the other three vertices are passed as texture coordinates. For each tetrahedron's vertex the attributed scalar value is also passed as the w coordinate. Note that different strategies, such as other geometric primitives, can also be used to send the tetrahedra through the pipeline; however, we found point primitives with texture coordinates to be the most efficient approach.

In the geometry shader, the tetrahedron is decomposed into triangles and the three values associated with each vertex, the traversal length l and scalar front s_f and back s_b, are computed using the same scheme as the original PT. To compute the correct tetrahedron projection, we rely on the ternary truth table from PTINT's classification strategy, determining the correct projection case with four cross products and a table look up. For further details we refer the reader to the original paper [MMFE08].

These three values (s_f, s_b and l) are computed depending on the projected vertex. There are two types of projected vertices: the *thick* and *thin* vertices. The thick vertex is defined as the entry point of the tetrahedron where the ray traverses the maximum distance l. Depending on the classification case, the thick vertex may not be one of the four projected tetrahedron's vertices, hence its scalar value has to be interpolated. Analogously, it might be necessary to compute the distance l in cases where l is not one of the edge lengths. Excluding the thick vertex, all others are the projected tetrahedron's vertices, named thin vertices, and have $s_f = s_b$, which are the original scalar value, and $l = 0$, since the ray traverses no distance in these extremities. In the example given in Figure 2, v'_0, v'_1 and v'_2 are the thin vertices while v'_3 is the thick vertex. The classification table stores not only the number of generated triangles, but the cases where the traversal distance l has to be computed and scalar values have to be interpolated for the thick vertex.

The scalar values (s_f and s_b) and traversal length (l) are stored as RGB colors of each corresponding vertex in order to pass the information from the triangles generated by the geometry shader along to the fragment shader.

3.3. Ray Integration

When a triangle is rasterized (one tetrahedron cell can be decomposed in one to four triangles), the scalar values and traversal length are interpolated per fragment. This interpolation is an approximation of the precise integration values for each cell used in the integral equation. The interpolated values (s_f, s_b and l) are employed per fragment to compute the ray integration through the partial pre-integration technique, in contrast to using a simple averaging scheme as done by the original PT and GATOR. In this technique the pre-computation of the ray integral equation does not depend on the transfer function, and thus it is pre-compiled within our application. Inside the fragment shader a table is accessed by two indices computed using the traversal length l and the front and back colors. In turn, these colors can be promptly extracted from the transfer function using the s_f and s_b values. Note that the partial pre-integration in fragment shader can be replaced by a better or faster integration method.

Even though this method is slower than computing the average value, it better approximates the ray integral improving the rendering results. Figure 5 depicts a model rendered with our algorithm – HAPT.

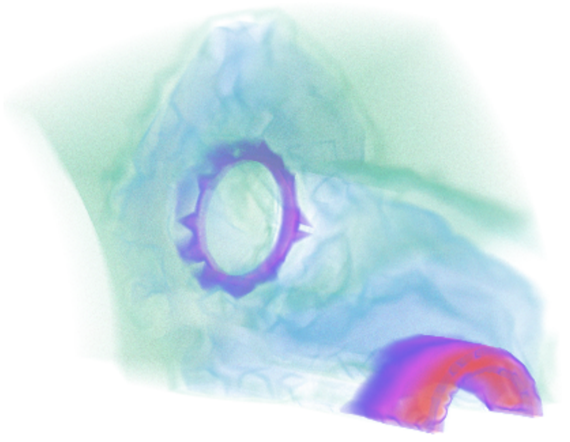

Figure 5: *Direct volume rendering of the spx2 dataset.*

3.4. Iso-Surface Rendering

The simple and flexible data flow of HAPT permits additional effects that are not trivially implementable with other methods. One example is interactive iso-surface rendering. While most approaches, such as PTINT, HAVS and HARC, are able to render iso-surfaces on a per-pixel basis (for each fragment determines if the iso-surface is in range), HAPT allows for not only this strategy, but also for an on-the-fly marching tetrahedra approach. Using the latter, normals can also be extracted to provide illumination effects. Since the geometry shader processes on a per tetrahedron basis, the iso-surface normals are restricted to faces, resulting in a flat shading. To achieve a smoother Phong-like effect it would be necessary to carry also adjacency information and compute normals per vertex from the incident faces, greatly impacting the memory consumption and rendering performance.

Within the geometry shader the iso-surface can be easily extracted from the tetrahedron and sent for rendering as one or two triangles. This method has two advantages: first, the iso-surface can be interactively defined; and second, it is possible to blend indirect and direct volume rendering generating a hybrid visualization, as illustrated in Figure 6.

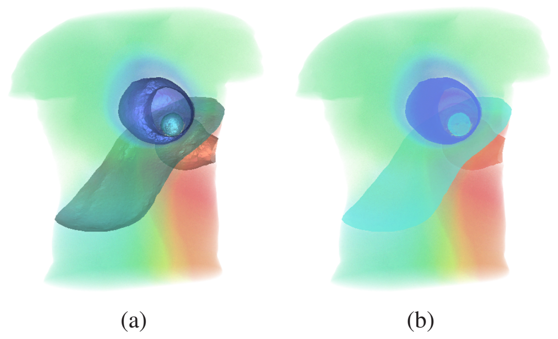

Figure 6: *Torso dataset rendered with direct volume rendering and iso-surfaces with (a) and without (b) illumination.*

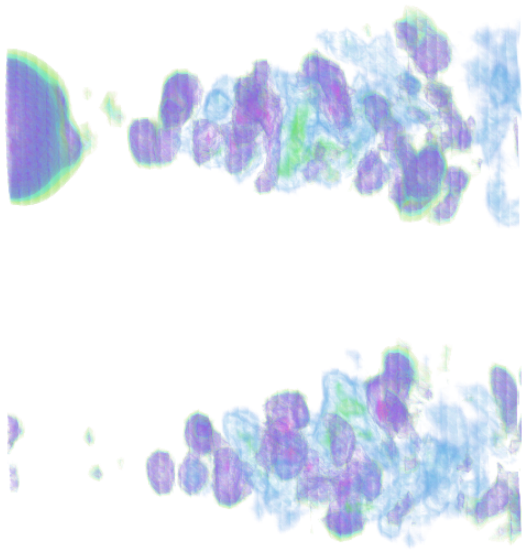

Figure 7: *Two of the 150 frames from the time-varying turbulent jet dataset (1 M tet per frame) rendered with HAPT.*

3.5. Time-Varying Rendering

To further demonstrate the algorithm's flexibility, we describe how it can be applied to time-varying datasets, where the volume is a sequence of animation frames, and each frame is a static volume.

As aforementioned in the beginning of Section 3, the usage of VBOs is optional and not suitable for models that do not fit on the GPU memory; hence, for time-varying datasets, the frames are not stored in memory but streamed as separate static volumes.

Additionally, we apply an early discard test in the vertex shader to remove empty tetrahedra, i.e. cells with all vertices mapped to zero opacity in the transfer function. Note that the volume must have large empty regions to profit from the additional texture fetches to discard the tetrahedra in vertex shader.

Figure 7 depicts two frames of a time-varying dataset, composed of 150 M tetrahedra over the whole animation. Applying the above modifications to HAPT's framework, we are able to interactively render this sequence at 17 fps.

4. Results

We have tested HAPT with the following irregular datasets: Blunt Fin (blunt); Oxygen Post (post); modified Super Phoenix (spx2); Delta Wing (delta); Human Torso (torso); Langley Fighter (fighter); and time-varying Turbulent Jet (turbjet). The dataset sizes and timings are presented in Table 2. The timings are given using a 512^2 pixel viewport and considering that the model is constantly rotating. All timings were performed in an Intel Xeon E5345 CPU with 4 GB of RAM and a GeForce 8800 GTX graphics card with 768 MB of VRAM (GPU memory).

For these results we have used either direct volume rendering, iso-surface rendering, or both (see Figure 6 and 8). For the last two cases, we have fixed a maximum of four iso-surfaces. One issue is that the geometry shader requires the specification of the maximum number of output vertices, which even when not fully used has an expressive impact on performance. This implies in a significant overhead when rendering with both methods (last two columns of Table 2). Even though it is unlikely that all four iso-surfaces will cross one tetrahedron at the same time, we have accounted for this to be consistent with the results, but efficiency can be greatly improved without considerably limiting the algorithm.

Table 3 compares our results with other GPU-based direct volume rendering algorithms (without extracting iso-surfaces nor having any illumination effect) using the spx2 dataset shown in Figure 5, where the sorting and drawing times are further detailed. The compared algorithms are:

- HAPT – Our approach, Hardware-Assisted Projected Tetrahedra, with four different sorting methods;
- HAVS – Hardware-Assisted Visibility Sorting [CICS05] with two k-buffer sizes;
- PTINT – Projected Tetrahedra with Partial Pre-Integration [MMFE08];
- GATOR – GPU Accelerated Tetrahedra Renderer [WMFC02];
- HARC – Hardware-Based Ray Casting with normal pre-integration [WKME03] and partial pre-integration [EC05].

The original PTINT algorithm uses an approximate bucket sorting during rotation. Here we take timings only for the complete CPU sorting; and even though PTINT's sort

Datasets	Size		*Dir. Vol. Rend.*		*Iso-surface Rend.*		*DVR + ISO*	
	# Verts	*# Tet*	*FPS*	*M Tet/s*	*FPS*	*M Tet/s*	*FPS*	*M Tet/s*
blunt	40 K	187 K	19.2	3.59	25.5	4.78	7.7	1.44
post	110 K	513 K	8.1	4.15	11.9	6.10	3.0	1.51
spx2	150 K	828 K	7.4	6.11	8.2	6.76	1.9	1.57
delta	211 K	1 M	4.5	4.52	6.0	6.01	1.5	1.51
torso	168 K	1.08 M	5.6	6.08	7.2	7.78	1.7	1.82
fighter	256 K	1.40 M	4.2	5.83	5.0	7.06	1.1	1.60
turbjet	212 K	1.01 M	17.5	17.67	n/a	n/a	n/a	n/a

Table 2: *Dataset sizes and total timing of our algorithm applying direct volume rendering, iso-surface rendering or both.*

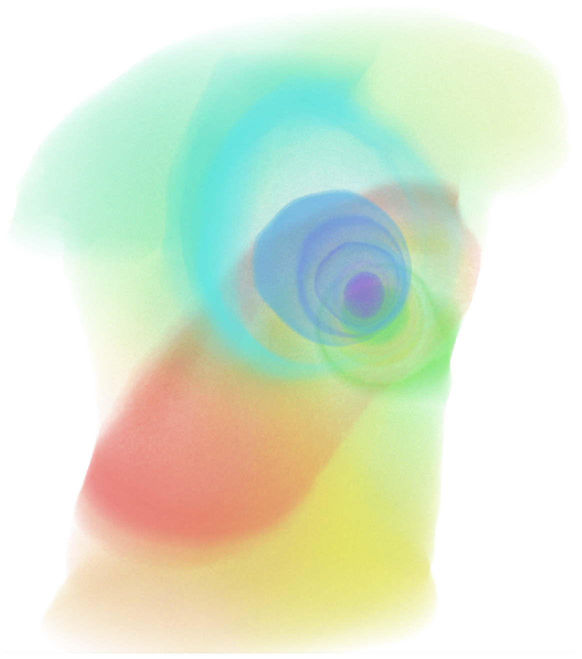

Figure 8: *Direct volume rendering of the torso dataset (without iso-surfaces).*

Algorithm	*Sort*	*Draw*	*FPS*	*M Tet/s*
HAPT^Q	0.03	0.09	7.4	6.11
HAPT^B	0.04	0.09	6.9	5.73
HAPT^S	0.08	0.09	5.4	4.50
HAPT^M	0.13	0.09	4.4	3.61
HAVS^2	0.09	0.11	5.0	4.14
HAVS^6	0.09	0.12	4.7	3.94
PTINT	0.19	0.20	2.4	2.06
GATOR	0.08	0.83	1.1	0.93
HARC^n	n/a	0.22	4.6	3.82
HARC^p	n/a	0.28	3.5	2.90

Table 3: *Timings for direct volume rendering of the spx2 model (828 K tet) with different algorithms and criteria. Sorting and drawing columns are in seconds. Variations for HAPT are **Q**uicksort, **B**itonic-sort, **S**TL-sort, **M**PVONC; for HAVS k-buffer are $k = \mathbf{2}$ and $k = \mathbf{6}$; and for HARC are with **n**ormal and **p**artial pre-integration.*

is equivalent to the STL sort method used by GATOR and HAPT^S, it still transfers data from the GPU for reordering during this step. Moreover, in both cases for HAVS, sorting time accounts only for the CPU pre-ordering, while for the HARC approaches there is no sorting step, since ray-casting algorithms traverse the entire volume using an adjacency data structure.

An important remark is that HAPT has better rendering time than the compared algorithms, including cell-projection, ray-casting or hybrid approaches, and it does not store the volume on the GPU memory as done by PTINT and HARC. On the other hand, HAVS and GATOR stream the volume similar to our approach, but perform a multi-pass or redundant streaming. The main variation in HAPT's performance comes from the shape of the tetrahedra, which influences the probability of the cell falling in each of the projection cases, dictating the number of generated triangles.

Table 4 specifies frames and tetrahedra per second for the torso and fighter datasets using HAPT as well as for the other methods compared. All results were taken for direct volume rendering flushing the graphics pipeline every frame, and thus not profiting from continuous streaming. The volume streaming uses VBOs for HAPT, HAVS and GATOR (PTINT and HARC read the volume from texture and can not benefit from VBOs and continuous streaming). This comparison shows that HAPT is faster even when dealing with datasets with more than one million cells.

One way to test the streaming performance of our algorithm for larger datasets (tens of millions of cells) is to render time-varying data as a sequence of static volumes. We tested HAPT using continuous streaming and an early discard test, explained in Section 3.5, to render the turbjet dataset (Figure 7). This time-varying volume consists of 150 frames with 1 million cells per frame, thus the entire animation is composed of 150 million different cells (see Table 2 for de-

Algorithm	torso 1,082 K Tet		fighter 1,403 K Tet	
	FPS	*M Tet/s*	*FPS*	*M Tet/s*
HAPT^Q	5.6	6.08	4.2	5.83
HAPT^B	4.3	4.68	3.6	5.09
HAPT^S	3.9	4.25	2.9	4.10
HAPT^M	1.6	1.73	1.2	1.62
HAVS^2	3.7	4.01	2.9	4.12
HAVS^6	3.3	3.60	2.7	3.89
PTINT	1.3	1.47	0.9	1.31
GATOR	0.7	0.76	0.4	0.56
HARC^n	4.8	5.19	3.8	5.33
HARC^p	3.9	4.22	3.0	4.21

Table 4: *Time comparison between HAPT and previous algorithms for the torso and fighter datasets.*

tails). For this dataset size, algorithms that rely on storing the whole volume on the GPU memory, such as PTINT and HARC, would require uploading and downloading different chunks of animation frames several times, and consequently would suffer a major performance loss.

Exemplary renderings of our algorithm are shown in Figures 7, 8, 9, and 10.

5. Discussion

In this section we highlight the main differences, advantages and disadvantages of our approach from previous methods.

GATOR and PTINT are the most similar methods to HAPT since they also built upon the original PT algorithm. Nevertheless, both previous algorithms rely on strategies to trick the graphics card in rendering tetrahedra. In contrast, HAPT at the same time avoids the data redundancy when streaming the volume imposed by GATOR, and storing the dataset on the GPU memory as done by PTINT. Our approach also avoids PTINT's two-pass approach by rendering in a single pass through the graphics pipeline.

The usage of the partial pre-integration technique improves the volume rendering quality of GATOR and the original PT, placing our algorithm in the same quality category of PTINT. When compared with ray-casting approaches, such as HARC, and a finer interpolation scheme, such as HAVS, HAPT presents the rendering problems of interpolating scalar values and traversal length at extremities inherited from the PT idea. This problem is more pronounced when rendering regular volumes converted to tetrahedra, as done in the turbulent jet sequence shown in Figure 7.

In terms of memory consumption, HAPT requires a fixed and very small amount of GPU memory to store the transfer function, classification table and the partial pre-integration table. Note that none of them are related to the volume size being rendered, yielding a total memory usage of about 10 KB. When streaming the volume using VBOs, our approach consumes additional GPU memory proportional to the dataset size. The memory-aware option is to send the primitives via streaming without using VBOs, losing 10% of the rendering performance, but avoiding memory limitations. The options of GPU memory usage by rendering primitives or applying VBOs can also be explored by GATOR and HAVS methods, since they also rely on streaming the volume to perform rendering. In contrast with these methods, HARC^n and HARC^p require the significant amount of 144 bytes/tet and 96 bytes/tet, respectively. Moreover, even an algorithm that has low memory footprint, such as PTINT, would require 3 GB of GPU memory to fit the entire turbulent jet sequence.

Although different in nature, HAVS is still one of the most popular irregular volume renderers. One point in favor of HAVS is that it performs a more accurate visibility sorting, when compared with the three centroid sort methods of our approach (HAPT^Q, HAPT^B and HAPT^S). However, our implementation offers a flexible pipeline, has a gain of approximately 50% in frame rates in the case of HAPT^Q *x* HAVS^6, and does not require multiple rendering passes. This latter characteristic may impact the overall rendering performance depending on the graphics card.

In addition, our strategy totally decoupled sorting from rendering, which leads to several benefits over previous algorithms. PTINT relies on bringing the data back to the CPU to be sorted in between passes. HAVS has sorting embedded with rendering and relies on two different sorting passes, one on the CPU and one on the GPU. HARC, as any ray-casting method, does not require to sort the volume cells, but on the other hand, it relies on auxiliary data structures that increases memory fetches and, consequently, decreases rendering performance. Moreover, the volume have to be loaded to the GPU texture memory, further limiting the volume size.

6. Conclusions

We have presented an efficient GPU approach of the PT algorithm called *HAPT – Hardware-Assisted Projected Tetrahedra* – that offers a series of advantages over previous methods. The algorithm is flexible and achieves higher frame rates than other volume rendering approaches by avoiding multiple rendering passes and data transfer from the GPU to the CPU, and not requiring auxiliary data structures. The flexibility and modularity provided by the pipeline allows for easy mixing and matching of other strategies, such as the sorting method and the partial pre-integration technique. Even more, iso-surfaces with normals can be extracted on the fly using the marching tetrahedra method. Finally, the algorithm is not limited by the amount of GPU memory since the vertices and cells are streamed to the GPU instead of keeping the whole volume in texture memory. This

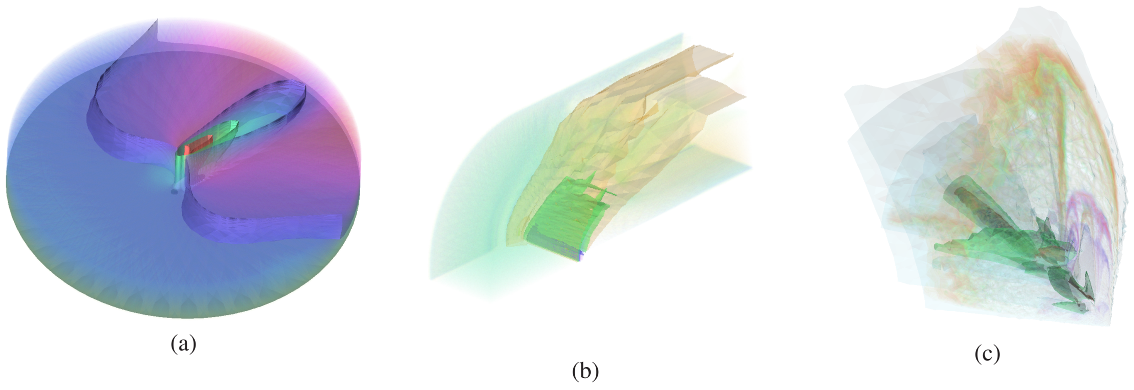

Figure 9: *Direct volume rendering with illuminated iso-surfaces: (a) post; (b) blunt; (c) fighter (front view of Figure 1).*

is not only desirable for large volumes, but even necessary for time-varying datasets.

Most of the efficiency comes from the fact that the algorithm follows closely the rasterization pipeline, profiting from the graphics hardware specialization on triangle rendering. This tight fit with the graphics pipeline makes the method less prone to becoming obsolete with new hardware improvements, considering that new trends will probably allow for extensions of HAPT instead of substantial reformulations. For instance, further enhancements may be possible with new hardware functionalities such as more control over the blending and depth operations.

Moreover, an improved geometry shader would allow normals per vertex to be computed (enabling iso-surface Phong shading), and better performance results when mixing direct and indirect volume rendering.

The algorithm has a few important peculiarities that we find pertinent to share with the reader. First, most of the improvements over previous cell-projection methods is related to the evolution of graphics hardware, so the geometry shader is mandatory. Second, the visibility ordering is still the main disadvantage for cell-projection algorithms; however, even if we have not solved this specific problem, the flexibility and modular pipeline assists on keeping up with new improvements of sorting methods. It is important to note that even with the MPVONC implementation in the CPU, HAPT is able to achieve considerable high frame rates. Although the centroid method introduces a very low error, one obvious pursue is a GPU implementation of the MPVONC, even if known not to be trivial.

In summary, we believe that the combination of flexibility and efficiency, due to the high adaptivity to the graphics card's rendering pipeline, brings the algorithm to a distinguished level of usability. The fact that the data can be streamed is also an important point in making this method suitable for possible out-of-core strategies, since it facilitates the data distribution among processing units.

Acknowledgments

We would like to acknowledge the grant of the first author provided by Brazilian agency CNPq (National Counsel of Technological and Scientific Development). We also thank the anonymous reviewers of this paper for their valuable comments, Dr. Varshney for his help and esteemed insights, and Dr. Kwan-Liu Ma for providing the Turbulent Jet dataset.

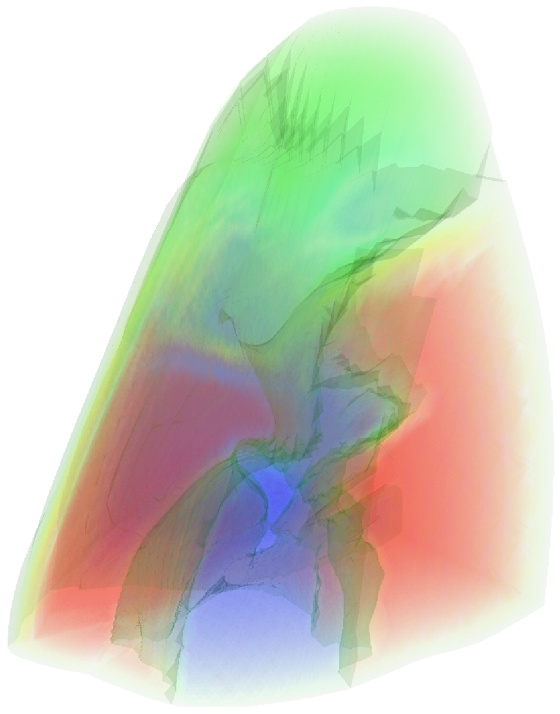

Figure 10: *Direct volume rendering and illuminated iso-surfaces of the delta dataset.*

References

[CICS05] CALLAHAN S., IKITS M., COMBA J., SILVA C.: Hardware-Assisted Visibility Ordering for Unstructured Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics 11*, 3 (2005), 285–295.

[CMSW04] COOK R., MAX N., SILVA C. T., WILLIAMS P. L.: Image-Space Visibility Ordering for Cell Projection Volume Rendering of Unstructured Data. *IEEE Transactions on Visualization and Computer Graphics 10*, 6 (Nov.-Dec. 2004), 695–707.

[CT08] CEDERMAN D., TSIGAS P.: A Practical Quicksort Algorithm for Graphics Processors. In *ESA '08: Proceedings of the 16th annual European symposium on Algorithms* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 246–258.

[CZY*08] CHEN S., ZHANG H., YUEN D., ZHANG S., ZHANG J., SHI Y.: Volume Rendering Visualization of 3D Spherical Mantle Convection with an Unstructured Mesh. *Visual Geosciences 13*, 1 (July 2008), 97–104.

[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.

[KE01] KRAUS M., ERTL T.: Cell-Projection of Cyclic Meshes. In *VIS '01: Proceedings of the conference on Visualization '01* (Washington, DC, USA, 2001), IEEE Computer Society, pp. 215–222.

[KQE04] KRAUS M., QIAO W., EBERT D. S.: Projecting Tetrahedra Without Rendering Artifacts. In *VIS '04: Proceedings of the 15th IEEE conference on Visualization '04* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 27–34.

[KSE04] KLEIN T., STEGMAIER S., ERTL T.: Hardware-Accelerated Reconstruction of Polygonal Isosurface Representations on Unstructured Grids. In *PG '04: Proceedings of the 12th Pacific Conference on Computer Graphics and Applications* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 186–195.

[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.

[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.

[MMFE07] MAXIMO A., MARROQUIM R., FARIAS R., ESPERANÇA C.: GPU-Based Cell Projection for Large Structured Data Sets. In *Proceedings of the II International Conference on Computer Graphics Theory and Applications (GRAPP'07), Barcelona-Spain* (March 2007).

[MMFE08] MARROQUIM R., MAXIMO A., FARIAS R., ESPERANÇA C.: Volume and Isosurface Rendering with Gpu-accelerated Cell Projection. *Computer Graphics Forum 27*, 1 (2008), 24–35.

[NVI07a] NVIDIA™: CUDA Environment – Compute Unified Device Architecture, 2007. http://www.nvidia.com/object/cuda_home.html.

[NVI07b] NVIDIA™: CUDA SDK Bitonic Sort Example, 2007. http://developer.download.nvidia.com/compute/cuda/sdk.

[Pas04] PASCUCCI V.: Isosurface Computation Made Simple: Hardware Acceleration, Adaptive Refinement and Tetrahedral Stripping. In *In Joint Eurographics - IEEE VGTC Symposium on Visualization (VisSym)* (2004), pp. 293–300.

[PLMS00] PLAUGER P., LEE M., MUSSER D., STEPANOV A. A.: *C++ Standard Template Library*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[RKE00] RÖTTGER 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.

[SCT06] SADOWSKY O., COHEN J. D., TAYLOR R. H.: Projected Tetrahedra Revisited: A Barycentric Formulation Applied to Digital Radiograph Reconstruction Using Higher-Order Attenuation Functions. *IEEE Transactions on Visualization and Computer Graphics 12* (2006), 461–473.

[SMW98] SILVA C. T., MITCHELL J. S. B., WILLIAMS P. L.: An Exact Interactive Time Visibility Ordering Algorithm for Polyhedral Cell Complexes. In *VVS '98: Proceedings of the 1998 IEEE symposium on Volume visualization* (New York, NY, USA, 1998), ACM, pp. 87–94.

[ST90] SHIRLEY P., TUCHMAN 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.

[TPG99] TREECE G. M., PRAGER R. W., GEE A. H.: Regularised Marching Tetrahedra: Improved Iso-Surface Extraction. *Computers & Graphics 23*, 4 (1999), 583–598.

[Wil92] WILLIAMS P. L.: Visibility-Ordering Meshed Polyhedra. *ACM Trans. Graph. 11*, 2 (1992), 103–126.

[WKE02] WEILER M., KRAUS M., ERTL T.: Hardware-Based View-Independent Cell Projection. In *VVS '02: Proceedings of the 2002 IEEE symposium on Volume visualization and graphics* (Piscataway, NJ, USA, 2002), IEEE Press, pp. 13–22.

[WKME03] 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* (2003), pp. 333–340.

[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.

[Zeh06] ZEHNER B.: Interactive Exploration of Tensor Fields in Geosciences using Volume Rendering. *Computers & Geosciences 32*, 1 (2006), 73–84.