High-Quality Hardware-Based Ray-Casting Volume Rendering Using Partial Pre-Integration

Rodrigo Espinha, Waldemar Celes
Tecgraf/PUC-Rio - Computer Science Department
Pontifical Catholic University of Rio de Janeiro
Rua Marquês de São Vicente 225, Rio de Janeiro, RJ, 22450-900, Brazil
{*rodesp, celes*}*@tecgraf.puc-rio.br*

Abstract

In this paper, we address the problem of the interactive volume rendering of unstructured meshes and propose a new hardware-based ray-casting algorithm using partial pre-integration. The proposed algorithm makes use of modern programmable graphics card and achieves rendering rates competitive with full pre-integration approaches (up to 2M tet/sec). This algorithm allows the interactive modification of the transfer function and results in high-quality images, since no artifact due to under-sampling the full numerical pre-integration exists. We also compare our approach with implementations of cell-projection algorithm and demonstrate that ray-casting can perform better than cell projection, because it eliminates the high costs involved in ordering and transferring data.

1. Introduction

The current power of modern programmable graphics hardware has made it possible to perform volume visualization of unstructured meshes at interactive rates using commodity computer platforms. In the last few years, different proposals have been presented by the volume visualization community for achieving fast and accurate volume visualization of tetrahedral meshes. As graphics hardware evolves, new algorithms, exploiting new graphics hardware capabilities, have been presented.

The ultimate goal is to provide an effective visualization tool for actual scientific applications, which impose challenges that have to be considered while designing volume-rendering algorithms:

- It is important to achieve interactive rates in order to allow the user to directly manipulate the model.
- It is important to achieve artifact-free images to avoid misinterpreting the model.
- It is important to allow the transfer functions to be interactively modified, which makes the inspection of the model easier.
- It is desirable to support dynamic meshes, thus allowing, for instance, the use of multi-resolution techniques to render large models.

Two different approaches seem to be the most promising for achieving interactive rates in the direct volume rendering of unstructured meshes. The first approach is based on cell projection [7, 19, 22, 27], as in the view-independent cell-projection (VICP) algorithm [22]. The second is based on volume ray-casting, as in the hardware-assisted ray-casting (HARC) algorithm [21, 23]. Both approaches present advantages and drawbacks. The two main drawbacks of cell-projection algorithms are the requirement of a visibility ordering of the cells (assuming, for instance, an emission-abortion optical model [12, 25]) and the high cost of transferring large amounts of data through the graphics bus. The ray-casting algorithm, on the other hand, demands more per-fragment processing and tends to be limited by the rasterization stage. The ray-casting algorithm also imposes restrictions on the size of the models, since the entire model has to be stored in the texture memory.

Aiming at the achievement of an interactive volume renderer for scientific applications, we have opted for investigating the ray-casting approach for direct volume rendering of linear tetrahedral meshes. Our choice was based on the growth of graphics processing power observed in recent years [11]. As ray-casting algorithms store the complete model in texture memory and perform all computations using the graphics processing unit (GPU), this approach tends to take better advantage of such fast growth. In fact, we have run a set of computational experiments demonstrating that the ray-casting approach can present better performance than cell-projection.

In this paper, we propose a modified hardware-based ray-casting algorithm that makes use of partial pre-integration [13] instead of using full pre-integration (whose results are stored in a 3D texture) [5, 18]. This new algorithm allows the interactive modification of the transfer function and results in high-quality images, since no artifact due to undersampling the full numerical pre-integration exists. We also present an alternative data structure to store the model, in an attempt to balance the trade-off between storage space and simplicity (ease of implementation). Moreover, for performance comparison, we implement a variant of the VICP algorithm storing the model in texture memory, thus eliminating the cost of transferring large amounts of data through the graphics bus.

The remaining of this paper is organized as follows: Section 2 describes previous work on optical models and volume-rendering integral. Section 3 reviews ray-casting algorithms and Section 4 describes, in detail, the new proposed algorithm using partial pre-integration, together with the data structure we have used to store the model in textures. Section 5 presents the experimental results we have achieved and, in Section 6, some concluding remarks are drawn.

2. Volume rendering integral

In order to be visualized using direct volume rendering, the scalar properties associated to the mesh's vertices are mapped to color and opacity by a transfer function. This function can be user-defined or computer-generated [8], and is often tabulated. Using an emission-absorption optical model [12, 25], we can integrate the total color and opacity contributions along a parametrized ray traversing the volume. Given a ray entering the volume at $\lambda = 0$ and exiting at $\lambda = l$, this results in the following equation:

$$I(l) = I_0 e^{-\int_0^l \tau(s(\lambda'))d\lambda'} + \int_0^l C(s(\lambda))\tau(s(\lambda))e^{-\int_\lambda^l \tau(s(\lambda'))d\lambda'} d\lambda \quad (1)$$

where λ is the ray parameter, s is the value of the scalar field, l is the total length of the ray segment inside the volume, and I_0 is the light's intensity where the ray enters the volume (at $\lambda = 0$). $C(s(\lambda))$ and $\tau(s(\lambda))$ are, respectively, the emitted light's intensity and the light attenuation factor associated to the scalar field by the transfer function. For the general case, there is no known analytical solution for Equation 1, thus requiring numerical integration.

However, Roettger et al. [18] have observed that the contribution of a single linear tetrahedron can be expressed by a function of three variables: the scalar value at the entering face (front), s_f; the scalar value at the exiting face (back), s_b; and the length of the ray segment inside the tetrahedron, l; as illustrated in Figure 1. This formulation allows coding the results of numerical pre-integration in a 3D texture. An advantage of the use of pre-integration is that it allows the transfer function to vary arbitrarily along the ray. Although good results can be achieved, pre-integration is still subject to some sampling issues [13], especially for functions containing high-frequency components. Of course, when using pre-integration, the 3D texture has to be updated whenever the transfer function is modified, which can be very costly. For this reason, approximate methods were devised [17, 21]. The pre-integration approach has also the shortcoming of being limited to one-dimensional transfer functions, and recent studies have demonstrated the usefulness of higher dimensional transfer functions [9].

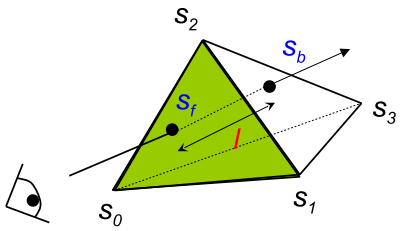

Figure 1. Ray-tetrahedron intersection.

As an alternative for avoiding numerical integration, Williams et al. [26] proposed the use of piecewise linear transfer functions (a general transfer function has to be decomposed into linear segments before its use). The transfer function is then described by a sequence of *control points* (as denominated by Moreland and Angel [13]), each defining a planar iso-surface of the scalar field inside a linear tetrahedron. The tetrahedron is then divided into several slices limited by two iso-surfaces (Figure 2). In each slice, the transfer function (both color and opacity) varies linearly and an analytic solution is used to compute the slice's contribution [13, 26], which can be then composed in back-to-front or front-to-back order.

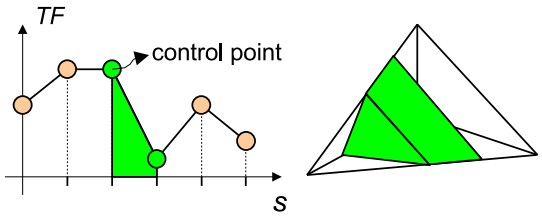

Figure 2. Piecewise linear transfer function (left) and the tetrahedron slice defined by the region between two control points (right).

However, analytically solving Equation 1 can be too expensive to be implemented in GPU and subject to numerical inaccuracies. In order to overcome this problem, Moreland and Angel [13] devised a new formulation for a fast integration of linear segments, which can be easily implemented in a fragment program. They called it *partial pre-integration*. Given $C(\lambda) = C_b(1 - \lambda/l) + C_f(\lambda/l)$ and $\tau(\lambda) = \tau_b(1 - \lambda/l) + \tau_f(\lambda/l)$, Equation 1 becomes:

$$I(l) = I_0\zeta(l, \tau(\lambda)) + C_b\left(\psi(l, \tau(\lambda)) - \zeta(l, \tau(\lambda))\right) + C_f\left(1 - \psi(l, \tau(\lambda))\right) \tag{2}$$

with

$$\zeta(l, \tau_f, \tau_b) = e^{-\frac{l}{2}(\tau_f + \tau_b)} \tag{3}$$

and

$$\psi(\gamma_f, \gamma_b) = \int_0^1 e^{-\int_\lambda^1 \left(\frac{\gamma_b}{1-\gamma_b}(1-\lambda') + \frac{\gamma_f}{1-\gamma_f}\lambda'\right) d\lambda'} d\lambda \tag{4}$$

where

$$\gamma \equiv \frac{\tau l}{\tau l + 1};\ \gamma \in [0, 1)$$

As Equation 4 is independent of the transfer function, it can be computed in a pre-processing step and stored in a conventional 2D texture map for later lookup. Equation 3 and, then, Equation 2 can be efficiently computed in the fragment program, with only a few instructions. The advantage of this technique is that it produces very accurate images and allows the use of multidimensional transfer functions.

A 2D texture is also used by Roettger et al. [18] for visualizing opaque iso-surfaces with direct volume rendering. Given the values s_f and s_b, the color of the first iso-surface hit by the ray inside a tetrahedron is stored in the texture, as illustrated in Figure 3. Consequently, the contribution of a tetrahedron along a ray is determined by a single texture lookup.

3. Hardware-based ray-casting algorithms

Weiler et al. [21] have proposed a GPU implementation of ray-casting, based on the algorithm presented by Bunyk at al. [3]. The idea of this algorithm is illustrated in Figure 4. The ray is propagated along the tetrahedral mesh using a simple adjacency-based topological data structure, which is entirely stored in texture memory. The propagation of the ray is started by rasterizing the front faces on the mesh's boundary. At each step, the contribution of a tetrahedron is computed by accessing the 3D texture with

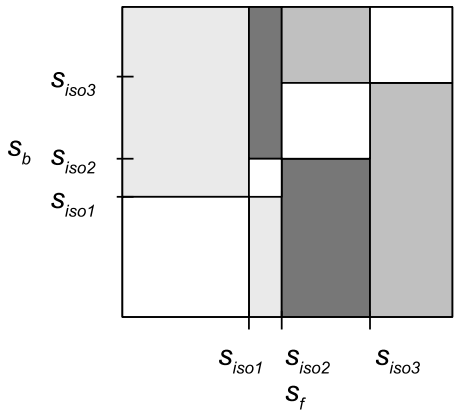

Figure 3. Texture map used to determine the color of the ray segment between s_f and s_b, for three iso-surfaces: s_{iso1}, s_{iso2} and s_{iso3}. The regions in white represent the values for which no iso-surface is hit.

the pre-integration results (after computing the values of s_b and l, as described in the next paragraph). The exiting point of a tetrahedron is the entering point to the next one. This process continues until the ray leaves the mesh. The intermediate results are stored in textures to be accessed by the next step. With the current capabilities of modern graphics cards, all contributions along the ray can be computed using a loop construction in the fragment program, thus avoiding the need for storing a few intermediate results.

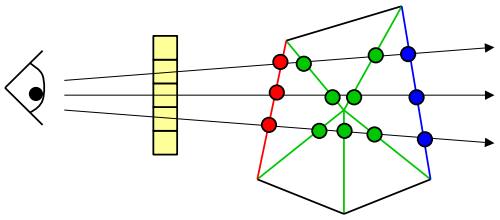

Figure 4. Ray propagation from cell to cell.

For computing the contribution at each step, we need access to the current tetrahedron index, t, the ray parameter at the entry point, λ, and the accumulated associated color and opacity. As front-to-back composition is used, it is possible to optimize the algorithm for high opacities by terminating the ray propagation early. Considering a parametrized ray starting at the eye position $\mathbf{e}$, the ray parameter for the intersection with the plane of the i^{th} face of the tetrahedron can be computed as:

$$\lambda_i = -\frac{(\mathbf{e} \cdot \mathbf{n_{t,i}} + o_{t,i})}{\mathbf{d} \cdot \mathbf{n_{t,i}}} \quad (5)$$

where $(\mathbf{n_{t,i}}, o_{t,i})$ is the plane equation of the face $f_{t,i}$, and $\mathbf{d}$ is the normalized ray direction. The exiting point is computed as $\mathbf{x_b} = \mathbf{e} + \lambda_i \mathbf{d}$, and the length of the ray segment inside the tetrahedron is then given by $l = \min(\max(\lambda_i, 0)) - \lambda$. The scalar value at the exiting point, $s_b = s(\mathbf{x_b})$, can be computed with the following equation:

$$s(\mathbf{x_b}) = \mathbf{g_t} \cdot \mathbf{x_b} + (s(\mathbf{x_0}) - \mathbf{g_t} \cdot \mathbf{x_0}) = \mathbf{g_t} \cdot \mathbf{x_b} + g_t \quad (6)$$

where $\mathbf{x_0}$ is an arbitrary position inside the tetrahedron (e.g, one of its vertices) and $\mathbf{g_t}$ is the gradient of the scalar field inside the tetrahedron. For linear tetrahedral meshes, the scalar term, $g_t = s(\mathbf{x_0}) - \mathbf{g_t} \cdot \mathbf{x_0}$, can be computed in a pre-processing step.The original algorithm was limited to convex meshes. Later, Bernardon et al. [2] and Weiler et al. [23] have proposed the use of the *depth-peeling* technique [6] to deal with non-convex meshes.

Due to graphics card limitations at the time the original algorithm was developed, Weiler et al. [21] have used a data structure that requires at least 144 bytes/tet, thus imposing significant restrictions on the size of the model to be handled. In their following up work, Weiler et al. [23] proposed to represent the meshes by strips of tetrahedra, thus requiring 76 bytes/tet. In both cases, the authors have suggested variants of the data structure: deriving information in the fragment program, using reduced representations to store information, etc. In doing that, the memory space for storing an ideal strip representing the model can be drastically reduced to 15 bytes/tet. However, this reduced representation comes along with a significant performance cost [23]. While encoding tetrahedral strips is an interesting approach to reduce memory consumption, it introduces an additional complexity to the algorithm.

4. Proposed ray-casting algorithm

In order to achieve better image quality and to allow the interactive modification of the transfer function, we propose a new hardware-assisted ray-casting algorithm using partial pre-integration. Also, aiming to balance the trade-off among storage space, performance, and simplicity, we have opted for using an alternative data structure.

4.1. Texture-based data structure

In our implementation of the hardware-based ray-casting algorithm, we have opted for using a variant of the original data structure. We have focused on a data structure that is more compact than the original (described in [21]), but conceptually less complex than the texture-encoded tetrahedral strips (described in [23]). We have also opted for a performance-optimized data structure, thus avoiding the need of deriving information in the fragment program and the use of reduced representations (all floats are 32-bit values). Table 1 presents the proposed data structured.

Table 1. Data structure used to store the model mesh for the ray-casting algorithm: t is the index of a tetrahedron; ($\mathbf{n_{t,i}}$,$o_{t,i}$) is the plane equation of the i^{th} tetrahedron face; $\mathbf{g_t}$ and g_t are, respectively, the gradient of the scalar field inside the tetrahedron and the scalar term of the Equation 6; and $a_{t,i}$ is the index of the tetrahedron which is adjacent to the i^{th} face.

Texture	Coord.		Data			
	u	v	r	g	b	a
Normal0	t		$\mathbf{n_{t,0}}$			$o_{t,0}$
Normal1	t		$\mathbf{n_{t,1}}$			$o_{t,1}$
Normal2	t		$\mathbf{n_{t,2}}$			$o_{t,2}$
Normal3	t		$\mathbf{n_{t,3}}$			$o_{t,3}$
Gradient	t		$\mathbf{g_t}$			g_t
Adjacency	t		$a_{t,0}$	$a_{t,1}$	$a_{t,2}$	$a_{t,3}$

This data structure requires 96 bytes/tet, which is less than the 144 bytes/tet of the original implementation, but more than the 76 bytes/tet of the performance-optimized version of texture-encoded tetrahedral strips. We believe its simplicity compensates for the additional memory space.

4.2. Partial pre-integration

We propose the use of partial pre-integration [13] to implement a new hardware-based ray-casting algorithm. Our implementation is inspired in the technique used by Roettger et al. [18] to render opaque iso-surfaces.

The actual transfer function is represented by a piecewise linear approximation, described by a sequence of control points (Figure 2). It suffices to have a good piecewise linear approximation of the transfer function for ensuring high accuracy. Each control point represents an iso-surface of the scalar field. In a linear tetrahedron, the iso-surfaces are planar and the transfer function is linear between two consecutive iso-surfaces. As a consequence, the volume rendering integral can be accurately evaluated using partial pre-integration.

At each step of the ray-casting algorithm, we are given the front scalar (s_f) computed in the previous step, and are able to compute the back scalar (s_b) using Equation 6. In order to integrate the linear segments along the viewing ray

inside a tetrahedron, we detect the value of the first iso-surface associated to a control point (s_{iso}) between s_f and s_b. We then integrate the segment from s_f to s_{iso}, using the formulation by Moreland and Angel [13]. The integration continues by setting s_{iso} as the new s_f value and repeating the process, trying to find the next crossed iso-surface. If there is no iso-surface between s_f and s_b, we integrate the ray from s_f to s_b, reaching the end of the tetrahedron. The process then continues through the adjacent tetrahedron until the ray reaches an external boundary of the mesh. This algorithm is similar to the original ray-casting procedure, but we perform some additional propagation steps in which the ray remains inside the same tetrahedron.

In order to detect the first iso-surface crossed by the ray between s_f and s_b, we make use of a 2D texture, like the one employed by Roettger et al. [18]. However, instead of iso-surface colors, our texture stores normalized scalar values representing iso-surfaces (s_{iso}), or -1 representing the absence of an iso-surface between s_f and s_b. Thus, given s_f and s_b, we access the texture and find the iso-surface between them. In order to correctly deal with sampling problems, the texture also stores the value of the consecutive iso-surface ($next\_s_{iso}$ or -1), whose use is described in the following example.

Let us consider the existence of a transfer function represented by three control points, which define the following iso-surfaces: $s_{iso1} = 0.25$, $s_{iso2} = 0.5$ and $s_{iso3} = 0.75$. The 2D texture is coded as illustrated in Figure 5. If we now consider a ray crossing a tetrahedron with $s_f = 0.2$ and $s_b = 0.7$, the 2D texture lookup provides the first iso-surface value, $s_{iso} = \mathbf{Tex2D}(0.2, 0.7) = 0.25$. The contribution from 0.2 to 0.25 is then computed. For the next iteration, we set $s_f = s_{iso} = 0.25$ and repeat the texture lookup. However, due to sampling problems, if s_f represents the value of an iso-surface, two different positions may be accessed. In this example, for $s_f = 0.25$, the nearest accessed position can report $s_{iso} = 0.5$, which would be correct, or $s_{iso} = 0.25$, which would cause the algorithm to loop forever. In order to disambiguate such occurrences, we use the second value stored in the texture ($next\_s_{iso}$). If s_f is equal to the reported s_{iso}, we must proceed to the next iso-surface ($s_{iso} = next\_s_{iso}$). So, the next integrated segment goes from 0.25 to 0.5, and s_f is set to 0.5 for the next iteration. Now, we have $s_f = 0.5$ and, accessing the texture, get $s_{iso} = 0.75$. As $s_b = 0.7$, s_{iso} is not inside the tetrahedron, and the integration goes from s_f to s_b. The algorithm then proceeds to the next tetrahedron.

The pseudo-code in Figure 6 illustrates one step of the described algorithm. The parameters passed for each propagation step are the current tetrahedron index, t, the entering ray parameter, λ, and the accumulated associated color and opacity, (R, G, B, A). After step execution, the corresponding parameters for the next step are returned. The algorithm proceeds until the ray leaves the mesh (when the tetrahedron index is invalid).

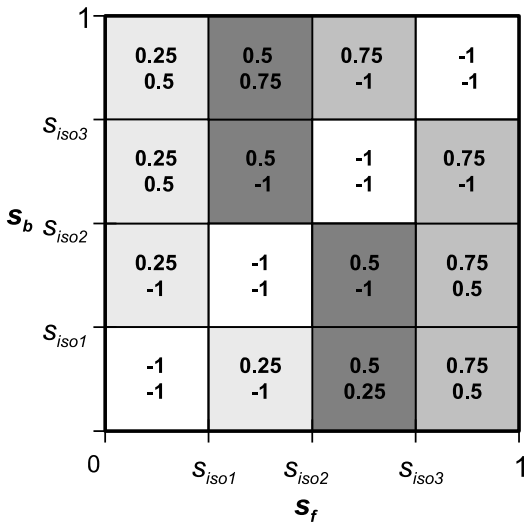

Figure 5. Texture map used to determine the next iso-surface crossed by the viewing ray inside a tetrahedron. The image illustrates a transfer function defined by three control points: $s_{iso1} = 0.25$, $s_{iso2} = 0.5$ and $s_{iso3} = 0.75$.

Whenever the transfer function is modified, only the 2D texture must be updated. This update is quite fast, allowing interactive modifications of the transfer function. For dealing with non-convex meshes, we use depth-peeling, as proposed by Bernardon et al. [2] and Weiler et al. [23].

5. Experimental results

In order to test the proposed algorithm, we have run a set of computational experiments. The tests were run on a NVIDIA GeForce 6800 GT graphics card with 256MB of memory and AGP 8X on a 2.53MHz Intel Pentium 4 machine with 512MB of memory. The vertex and fragment programs were implemented using the Cg language [15, 16], with the *vp40* and *fp40* profiles. The data sets used are presented in Figure 7. The Fixed Bar and the Wheel data sets are finite-element models. The Bluntfin and the Liquid Oxygen Post data sets are results of fluid simulations available at the NASA's NAS website [14]. The reported times were achieved by redrawing the model for several different fixed viewpoints in a 512x512 viewport. For all the implementations, we have disabled early ray termination.

For comparison purposes, we have implemented the view-independent cell projection (VICP) algorithm as de-

HARC_PartialPreInt_Step$(t,\ \lambda,\ (R,G,B,A))$

1. $\lambda' \leftarrow \infty$
2. $t' \leftarrow 0$
3. **for** $i \leftarrow 0$ **to** 3 **do**
4. **if** $(\mathbf{d} \cdot \mathbf{n_{t,i}}) > 0$ **then**
5. $\lambda_{tmp} \leftarrow -(\mathbf{e} \cdot \mathbf{n_{t,i}} + o_{t,i}) \,/\, (\mathbf{d} \cdot \mathbf{n_{t,i}})$
6. **if** $\lambda_{tmp} < \lambda'$ **then**
7. $\lambda' \leftarrow \lambda_{tmp}$
8. $t' \leftarrow a_{t,i}$
9.
10. $s_f \leftarrow \mathbf{g_t} \cdot (\mathbf{e} + \lambda \mathbf{d}) + g_t$
11. $s_b \leftarrow \mathbf{g_t} \cdot (\mathbf{e} + \lambda' \mathbf{d}) + g_t$
12. $l \leftarrow \lambda' - \lambda$
13. $(s_{iso}, next\_s_{iso}) \leftarrow \mathbf{T}ex2D(s_f, s_b)$
14. **if** $s_{iso} \neq -1$ **then**
15. **if** $s_f = s_{iso}$ **then**
16. $s_{iso} \leftarrow next\_s_{iso}$
17. **if** $\mathbf{abs}(s_b - s_f) > \mathbf{abs}(s_{iso} - s_f)$ **then**
18. $t' \leftarrow t$
19. $l \leftarrow l(s_{iso} - s_f)/(s_b - s_f)$
20. $\lambda' \leftarrow \lambda + l$
21. $s_b \leftarrow s_{iso}$
22.
23. $(r,g,b,a) \leftarrow \mathbf{IntegrateRaySegment}(s_f, s_b, l)$
24. $(R,G,B,A)' \leftarrow (R,G,B,A) + (1-A) \cdot (r,g,b,a)$
25. **return** $(t',\ \lambda',\ (R,G,B,A)')$

Figure 6. Pseudo-code for one step of the propagation of a viewing ray in the proposed algorithm.

scribed by Weiler et al. [22], using a 128^3 8-bit RGBA pre-integrated texture and the MPVONC algorithm [24] for the visibility ordering of the cells. We have also implemented a variant of the VICP algorithm (named VICP-GPU), storing the model in graphics memory and, hence, eliminating the high cost of transferring data through the graphics bus. The original hardware-based ray-casting (HARC) algorithm was implemented using a 128^3 8-bit RGBA and dynamic looping capabilities of the GeForce 6800 GT graphics card. For this implementation, we used the data structure described in Section 4.1.

Table 2 shows the achieved results. As can be noted, by storing the data in graphics memory we were able to significantly improve the VICP performance, but our implementations are still limited by the visibility ordering of the cells. For this reason, our implementation of the original HARC presents better performance with the larger models (these models fit well in the available graphics memory). Recently, Callahan et al. [4] have proposed a hardware-assisted visibility-ordering algorithm for cell projection, reporting a rendering rate from 1.5M to 1.8M tets/sec[1] for different models (ranging from 240K to 1.4M cells).

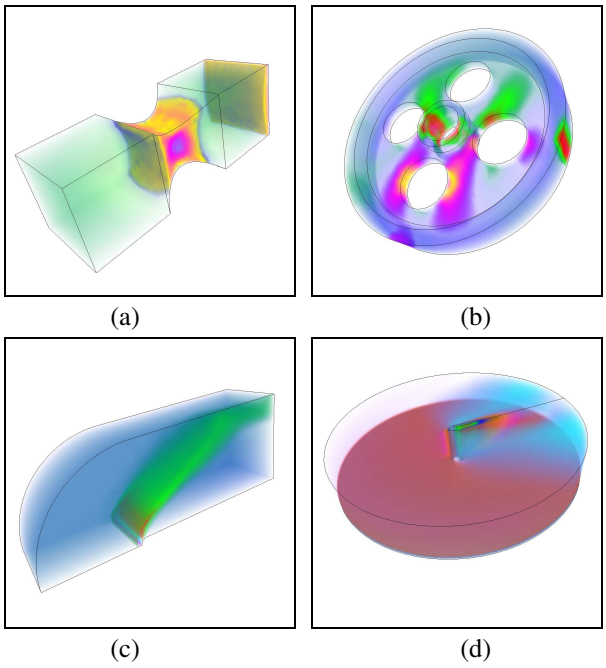

Figure 7. Images of tetrahedral meshes rendered with our proposed ray-casting algorithm using partial pre-integration: (a) Fixed Bar model with 27,691 tets; (b) Wheel model with 31,725 tets; (c) Bluntfin model with 224,874 tets; (d) Liquid Oxygen Post model with 616,050 tets.

For the proposed ray-casting algorithm with partial pre-integration, we have tested approximating the transfer function with 10 and 255 linear segments. The performance of the algorithm decreases as the number of linear segments increases, because more steps are necessary to propagate the ray. Moreland and Angel [13] reported a growth of 3% to 4% in the number of steps for each additional control point in the transfer function. Nevertheless, as shown in Table 3, the performance achieved using 10 segments is competitive with the pre-integration approach. Even with 255 segments, the performance was better than VICP for the larger models. However, it is important to mention that, for highly non-convex meshes, the application of the depth-peeling technique may be expensive, thus decreasing the performance of ray-casting algorithms.

With the proposed hardware-based ray-casting with partial pre-integration, we have been able to generate images

[1] Callaham et al. [4] have used an ATI Radeon 9800 graphics card in their experiments.

Table 2. Comparison of the achieved results of the VICP, VICP-GPU, and HARC algorithms. The table shows the minumum and maximum achieved values, considering different viewpoints.

Data set	VICP		VICP-GPU		HARC	
	fps	tet/s	fps	tet/s	fps	tet/s
Fixed bar	9.84	272K	24.63	682K	8.76	243K
	10.67	295K	26.67	739K	14.88	412K
Wheel	8.76	278K	22.08	700K	7.80	247K
	9.28	294K	22.88	726K	14.53	461K
Bluntfin	1.26	283K	3.09	695K	6.09	1.37M
	1.36	306K	3.50	787K	8.76	1.97M
Oxygen	0.46	283K	0.98	604K	3.40	2.09M
	0.47	290K	1.21	745K	5.91	3.64M

Table 3. Comparison between the original HARC algorithm and the proposed HARC with partial pre-integration. The table shows the minumum and maximum achieved values, considering different viewpoints.

Data set	HARC-Full		HARC-Partial (10 segments)		HARC-Partial (255 segments)	
	fps	tet/s	fps	tet/s	fps	tet/s
Fixed bar	8.76	243K	6.27	174K	2.41	67K
	14.88	412K	1.22	311K	4.81	133K
Wheel	7.80	247K	7.35	233K	3.86	122K
	14.53	461K	10.67	339K	7.19	228K
Bluntfin	6.09	1.37M	3.50	787K	2.98	670K
	8.76	1.97M	7.11	1.60M	5.04	1.13M
Oxygen	3.40	2.09M	2.32	1.43M	1.22	752K
	5.91	3.64M	4.21	2.59M	2.38	1.47M

with significant superior quality if compared to the results achieved by using full pre-integration, because no artifact due to under-sampling the full numerical pre-integration exists. This is especially important for meshes with cells of different sizes, as illustrated in Figure 8. Recently, Kraus at al. [10] have proposed the use of a logarithmic scale for the pre-integrated color lookup table to remove artifacts from the full pre-integration for cell-projection.

6. Conclusion

The proposed hardware-based ray-casting algorithm using partial pre-integration has shown to be competitive with the full pre-integration approach while ensuring high quality and accurate images, thus being appropriate to be integrated into actual scientific applications. It also allows interactive modifications of the transfer function, which can help the user to inspect the model.

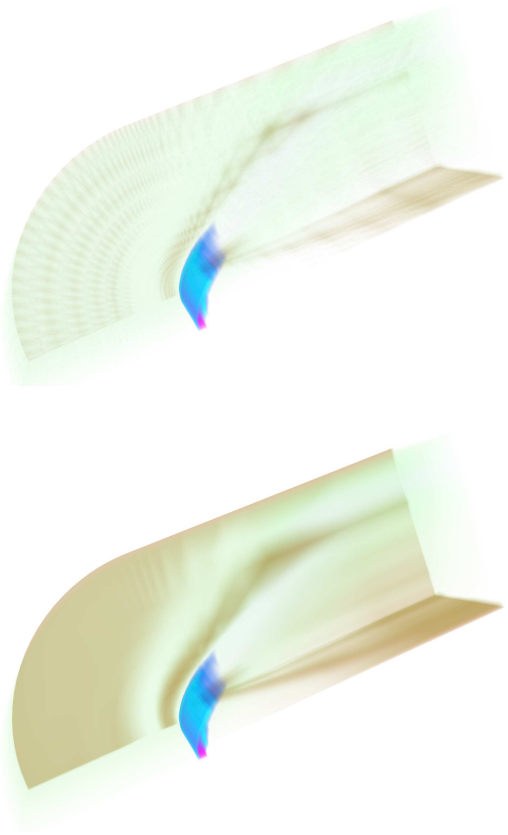

Figure 8. Difference in image quality achieved by the HARC algorithm with full pre-integration (at the top) and by the proposed algorithm with partial pre-integration (at the bottom).

In our implementations, ray-casting performs better than cell projection for the larger models, because it eliminates the high costs involved in ordering and transferring data. However, the graphics memory can be a limiting factor in the use of ray-casting. Nevertheless, the next generation of graphics cards will have up to 512Mb of graphics memory, which is sufficient to represent very large meshes, even using the simple data structure described in Section 4.1 (which requires 96 bytes/tet). We believe that the conception of texture-based compact data structures for handling dynamic meshes is a promising research topic.

One additional advantage of the ray-casting approach is that the rays are processed in parallel, being the appropriate choice for sort-first distributed visualization using clusters

of PCs [1]. We also believe that the ray-casting approaches will greatly benefit from the upcoming graphics card generations.

Acknowlegments

During this research, the first author was financially supported by the Brazilian agency CAPES (*Coordenação de Aperfeiçoamento de Pessoal de Nível Superior*). We thank the support for conducting this research provided by the Tecgraf laboratory at PUC-Rio, which is mainly funded by the Brazilian oil company, Petrobras.

References

[1] F. R. Abraham, W. Celes, R. Cerqueira, and J. L. E. Campos. A load-balancing strategy for sort-first distributed rendering. In *Proceedings of SIBGRAPI '2004*. IEEE Computer Society, 17–20 Oct. 2004.

[2] F. F. Bernardon, C. A. Pagot, J. L. D. Comba, and C. T. Silva. GPU-Tiled Ray Casting using Depth Peeling. Technical report, UUSCI-2004-006, SCI Institute, University of Utah, 2004.

[3] P. Bunyk, A. Kaufman, and C. T. Silva. Simple, Fast, and Robust Ray Casting of Irregular Grids. In *DAGSTUHL '97: Proc. of the Conference on Scientific Visualization*, page 30, Washington, DC, USA, 1997. IEEE Computer Society.

[4] 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.

[5] K. Engel, M. Kraus, and T. Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proc. of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 9–16, New York, NY, USA, 2001. ACM Press.

[6] C. Everitt. Interactive Order-Independent Transparency. Technical report, NVIDIA Corporation, http://developer.nvidia.com, 2001.

[7] R. Farias, J. S. B. Mitchell, and C. T. Silva. ZSWEEP: an efficient and exact projection algorithm for unstructured volume rendering. In *Proc. of the 2000 IEEE symposium on Volume visualization*, pages 91–99, New York, NY, USA, 2000. ACM Press.

[8] G. Kindlmann and J. W. Durkin. Semi-automatic generation of transfer functions for direct volume rendering. In *Proc. of the 1998 IEEE symposium on Volume visualization*, pages 79–86, New York, NY, USA, 1998. ACM Press.

[9] J. Kniss, G. Kindlmann, and C. Hansen. Multidimensional Transfer Functions for Interactive Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):270–285, 2002.

[10] 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.

[11] A. Lefohn, I. Buck, J. D. Owens, and R. Strzodka. GPGPU: General-Purpose Computation on Graphics Processors, Tutorial no. 3. In *Proc. of IEEE Visualization 2004*, 2004.

[12] N. Max. Optical Models for Direct Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, 1995.

[13] K. Moreland and E. Angel. A Fast High Accuracy Volume Renderer for Unstructured Data. In *VV '04: Proceedings of the 2004 IEEE Symposium on Volume Visualization and Graphics (VV'04)*, pages 9–16, Washington, DC, USA, 2004. IEEE Computer Society.

[14] NASA. NAS - NASA Advanced Supercomputing Division, 2005. http://nas.nasa.gov.

[15] NVIDIA. NVIDIA, Cg Toolkit User's Manual: A Developer's Guide to Programmable Graphics, release 1.2, 2004. http://developer.nvidia.com.

[16] NVIDIA. NVIDIA GPU Programming Guide Version 2.2.1, 2004. http://developer.nvidia.com.

[17] S. Roettger 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.

[18] 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.

[19] P. Shirley and A. Tuchman. A polygonal approximation to direct scalar volume rendering. In *VVS '90: Proceedings of the 1990 workshop on Volume visualization*, pages 63–70, New York, NY, USA, 1990. ACM Press.

[20] C. M. Stein, B. G. Becker, and N. L. Max. Sorting and hardware assisted rendering for volume visualization. In *Proc. of the 1994 Symposium on Volume Visualization*, pages 83–89, New York, NY, USA, 1994. ACM Press.

[21] M. Weiler, M. Kraus, M. Merz, and M. Ertl. Hardware-Based Ray Casting for Tetrahedral Meshes. In *Proc. of IEEE Visualization '03*, 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] M. Weiler, P. N. Mallon, M. Kraus, and T. Ertl. Texture-Encoded Tetrahedral Strips. In *Proc. of the 2004 IEEE Symposium on Volume Visualization and Graphics*, pages 71–78, Washington, DC, USA, 2004. IEEE Computer Society.

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

[25] P. L. Williams and N. Max. A volume density optical model. In *Proc. of the 1992 Workshop on Volume Visualization*, pages 61–68, New York, NY, USA, 1992. ACM Press.

[26] P. L. Williams, N. L. Max, and C. M. Stein. A High Accuracy Volume Renderer for Unstructured Data. *IEEE Trans. on Visualization and Computer Graphics*, 4(1):37–54, 1998.

[27] B. Wylie, K. Moreland, L. A. Fisk, and P. Crossno. Tetrahedral projection using vertex shaders. In *Proc. of the 2002 IEEE Symposium on Volume Visualization and Graphics*, pages 7–12, Piscataway, NJ, USA, 2002. IEEE Press.