Pontifícia Universidade Católica
do Rio de Janeiro

Fábio Markus Nunes Miranda

Volume rendering of unstructured hexahedral meshes

DISSERTAÇÃO DE MESTRADO

Dissertation presented to the Postgraduate Program in Informatics, of the Departamento de Informática, PUC–Rio as partial fulfillment of the requirements for the degree of Mestre em Informática.

Advisor: Prof. Waldemar Celes

Rio de Janeiro
September 2011

Pontifícia Universidade Católica
do Rio de Janeiro

Fábio Markus Nunes Miranda

Volume rendering of unstructured hexahedral meshes

Dissertation presented to the Postgraduate Program in Informatics, of the Departamento de Informática do Centro Técnico Científico da PUC–Rio, as partial fulfillment of the requirements for the degree of Mestre.

Prof. Waldemar Celes
Advisor
Departamento de Informática — PUC–Rio

Prof. Marcelo Gattass
Department of Computer Science — PUC-Rio

Prof. Thomas Lewiner
Department of Mathematic — PUC-Rio

Prof. Luiz Henrique de Figueiredo
Instituto Nacional de Matemática Pura e Aplicada (IMPA)

Prof. José Eugenio Leal
Coordinator of the Centro Técnico Científico — PUC–Rio

Rio de Janeiro, September 5, 2011

Fábio Markus Nunes Miranda

Fabio Markus Miranda graduated from Universidade Federal de Minas Gerais (UFMG) with a B.Sc. in Computer Science in 2009. While at UFMG, he received scholarships from CNPq in the bioinformatics area and later Finep. In 2009, he started the graduate program in Computer Science at PUC–Rio. In 2010 he joined the Computer Graphics Technology Group (Tecgraf), contributing to the group's work for Petrobras.

Bibliographic data

Miranda, Fábio Markus

Volume rendering of unstructured hexahedral meshes / Fábio Markus Nunes Miranda ; advisor: Waldemar Celes. — 2011.
47 f. : il. ; 30 cm

Dissertação (Mestrado em Informática)-Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2011.
Inclui bibliografia

1. Informática – Dissertação. 2. Renderização volumétrica. 3. Malha de hexaedros. 4. Malha não estruturada. 5. Integral do raio. I. Celes, Waldemar. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Acknowledgments

To my family, my adviser, my friends and collegues.
To CAPES and Tecgraf for the financial support.

Resumo

Miranda, Fábio Markus; Celes, Waldemar. **Renderização volumétrica de malha não estruturada de hexaedros**. Rio de Janeiro, 2011. 47p. Dissertação de Mestrado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Importantes aplicações de engenharia usam malhas não estruturadas de hexaedros para simulações numéricas. Células hexaédricas, comparadas com tetraedros, tendem a ser mais numericamente estáveis e requerem um menor refinamento da malha. Entretando, visualização volumétrica de malhas não estruturadas é um desafio devido a variação trilinear do campo escalar dentro da célula. A solução convencional consiste em subdividir cada hexaedro em cinco ou seis tetraedros, aproximando uma variação trilinear por uma inadequada série de funções lineares. Isso resulta em imagens inadequadas e aumenta o consumo de memória. Nesta tese, apresentamos um algoritmo preciso de visualização volumétrica utilizando *ray-casting* para malhas não estruturadas de hexaedros. Para capturar a variação trilinear ao longo do raio, nós propomos usar uma integração de quadratura. Nós também propomos uma alternativa rápida que melhor aproxima a variação trilinear, considerando os pontos de mínimo e máximo da função escalar ao longo do raio. Uma série de experimentos computacionais demonstram que nossa proposta produz resultados exatos, com um menor gasto de memória. Todo algoritmo é implementado em placas gráficas, garantindo uma performance competitiva.

Palavras–chave

Renderização volumétrica; Malha de hexaedros; Malha não estruturada; Integral do raio;

Abstract

Miranda, Fábio Markus; Waldemar Celes (Advisor). **Volume rendering of unstructured hexahedral meshes**. Rio de Janeiro, 2011. 47p. MSc Dissertation — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Important engineering applications use unstructured hexahedral meshes for numerical simulations. Hexahedral cells, when compared to tetrahedral ones, tend to be more numerically stable and to require less mesh refinement. However, volume visualization of unstructured hexahedral meshes is challenging due to the trilinear variation of scalar fields inside the cells. The conventional solution consists in subdividing each hexahedral cell into five or six tetrahedra, approximating a trilinear variation by an inadequate piecewise linear function. This results in inaccurate images and increases the memory consumption. In this thesis, we present an accurate ray-casting volume rendering algorithm for unstructured hexahedral meshes. In order to capture the trilinear variation along the ray, we propose the use of quadrature integration. We also propose a fast approach that better approximates the trilinear variation to a series of linear ones, considering the points of minimum and maximum of the scalar function along the ray. A set of computational experiments demonstrates that our proposal produces accurate results, with reduced memory footprint. The entire algorithm is implemented on graphics cards, ensuring competitive performance.

Keywords

Volume Rendering; Hexahedral Mesh; Unstructured Mesh; Ray Integral;

Contents

List of Figures

List of Tables

I seldom end up where I wanted to go, but almost always end up where I need to be.

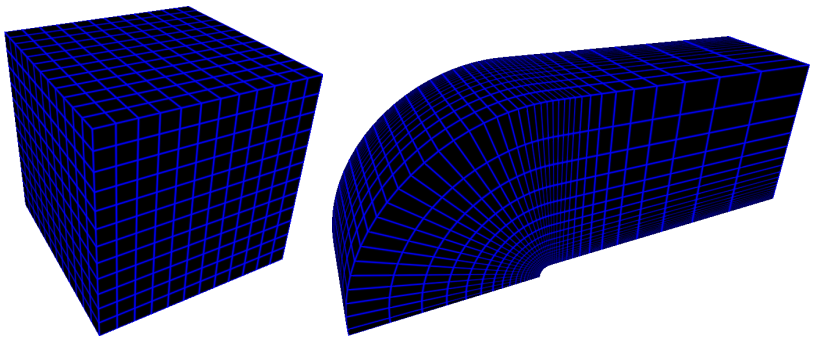
Douglas Adams, *The Hitchhiker's Guide to the Galaxy.*

1
Introduction

Volume rendering is a popular way to visualize scalar fields from a number of different data, from medical MRI to results of scientific simulations. Unlike regular surface rendering, volume rendering allows the user to visualize the interior of the dataset. By computing the light intensity along rays as they traverse the volume, it is possible to calculate the color and opacity for the pixels on the screen. How to accurately calculate the interaction between light and volume, while maintaining an acceptable rendering time, is one of the main difficulties of volume visualization algorithms.

Important engineering applications use unstructured hexahedral meshes for numerical simulations. Hexahedral cells, when compared to tetrahedral ones, tend to be more numerically stable and to require less mesh refinement. However, volume visualization of unstructured hexahedral meshes is challenging due to the trilinear variation of scalar fields inside the cells. The conventional solution consists in dividing each hexahedron cell into five or six tetrahedra, approximating a trilinear variation by a piecewise linear function. This results in a less smooth variation, inaccurate images and increases the memory consumption.

Figures 1.1(a) and 1.1(b) present two typical types of meshes. The first one is an example of structured data. Such data is represented by a set of uniform size cells and, as such, its adjacency information can be implicitly retrieved (e.g. cell (i, j, k) is adjacent to cell $(i+1, j, k)$). Figure 1.1(b), on the other hand, is an unstructured data; its cells do not necessarily have an uniform size and its adjacency information can no longer be retrieved implicitly; we must store such information in a data structure.



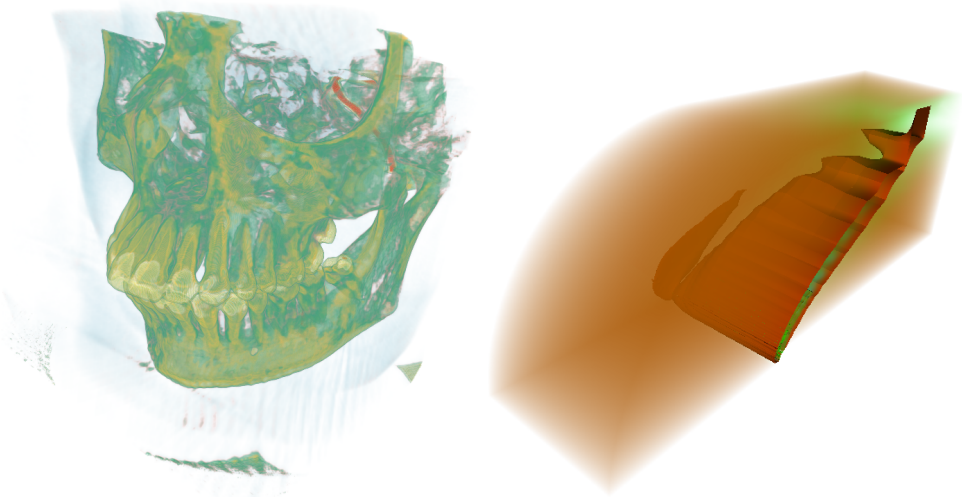
1.1(a): Structured regular mesh

1.1(b): Unstructured mesh

Figure 1.1: Meshes types.

Figures 1.2(a) and 1.2(b) present two typical applications of volume

visualization. The first one, an x-ray image, is a structured regular data, while Figure 1.2(b) illustrates an unstructured data of a computational fluid dynamics simulation.



1.2(a): X-Ray 1.2(b): Computational fluid dynamics

Figure 1.2: Volume rendering examples.[1]

One popular approach to render volumetric data, both structured and unstructured, is ray-casting (4). By tracing a ray for each pixel on the screen, we can traverse the volume and calculate the contribution of a set of discrete volume cells to the final pixel color, using an emission-absorption optical model (17). Because volumetric data are nothing more than a series of scalar values arranged in the 3D space, we must map those scalar values to color and opacity using a transfer function, as seen in Figure 1.3. Considering a normalized scalar field, with its values ranging from 0 to 1, the associated color and opacity of a scalar value will be fetched from a RGBA texture, using the scalar as the texture coordinate.

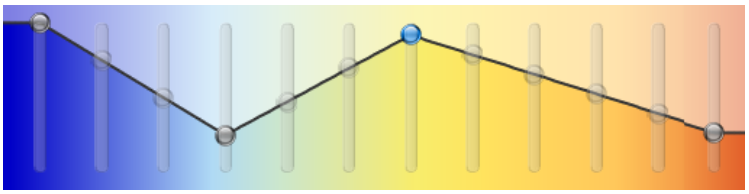
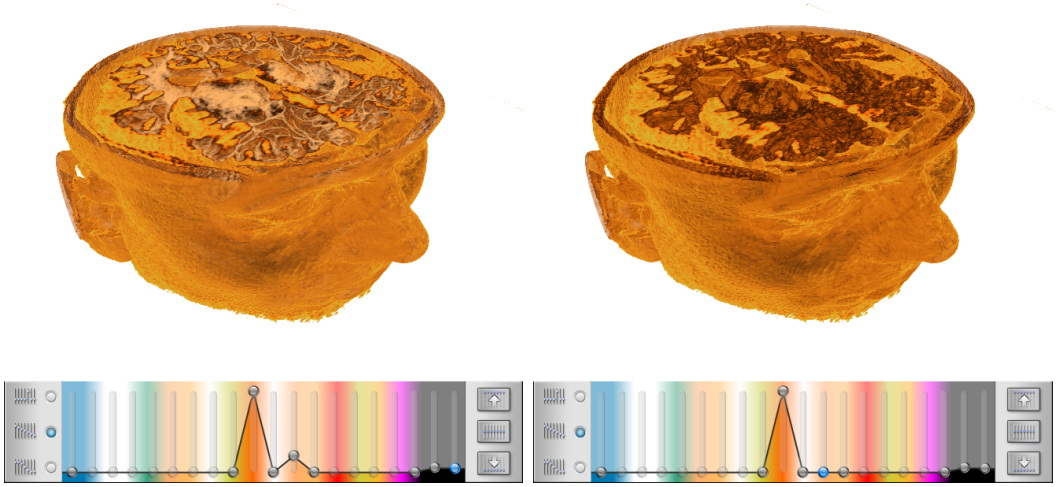


Figure 1.3: An example of a transfer function

Figure 1.4 shows an example of a magnetic resonance imaging dataset modificated through the use of a transfer function.

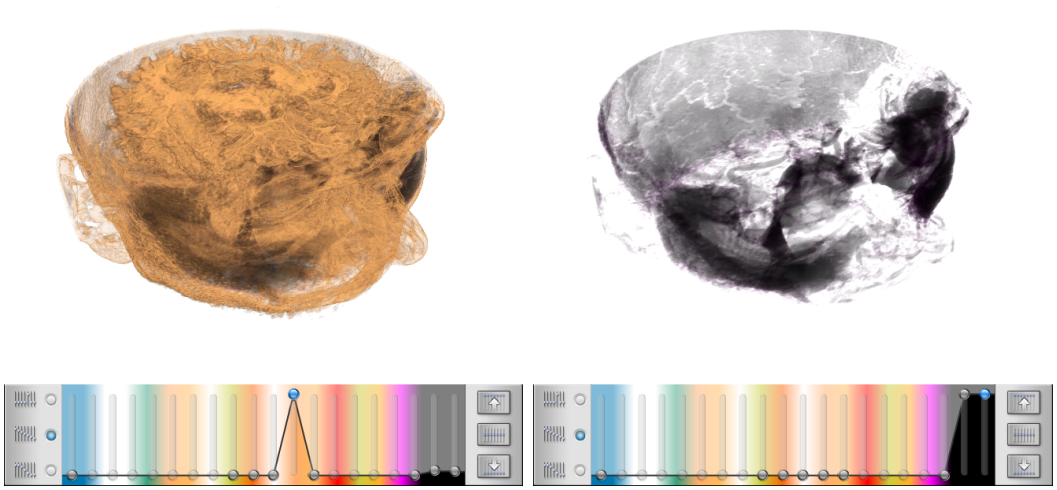
For regular data, as in Figure 1.2(a), it is common to store the volume into a 3D texture and traverse the volume using regular steps, sampling the texture

[1]All volume images in this thesis were taken from our volume renderer.



1.4(a): Transfer function 1

1.4(b): Transfer function 2



1.4(c): Transfer function 3

1.4(d): Transfer function 4

Figure 1.4: Example of four transfer functions on a MRI dataset.

as the ray progresses through the volume. Volume rendering of unstructured data is not so simple; we cannot store the volume into a 3D texture and just sample it, because of the non-uniform cell size. We must then traverse the volume cell-by-cell. As the ray traverses the volume, its color and opacity are calculated taking into consideration the variation of the scalar field inside each cell. As the number of nodes per cell increases, so does the order of the scalar function inside the cell.

For obvious reasons, a linear variation of the scalar field translates into a much simpler interaction between the ray light and the volume. That is why the common approach to render hexahedral meshes is to split each cell into five or six tetrahedra, approximating the trilinear variation of the scalar function by a piecewise linear function. This results in inaccurate image and increases the memory consumption. In this thesis we present a ray-casting

volume rendering algorithm for unstructured hexahedral meshes that considers the trilinear variation of the scalar field inside the cells, and uses a quadrature integration scheme to calculate the interaction between light and volume. We evaluate our solution considering rendering time, image quality, and memory efficiency, and compare it against a hexahedral division solution.

We also present another proposal that approximates the trilinear scalar function by a linear one, but considering certain integration intervals that makes it a better approximation than a hexahedral division approach.

The main contributions of this thesis are:

- A proposal for accurate volume rendering of unstructured hexahedral meshes, considering a trilinear variation of the scalar field.
- A proposal for approximated volume rendering of unstructured hexahedral meshes, considering a linear variation of the scalar field.

This thesis is divided as follow: Chapter 2 presents relevant related work. Chapter 3 presents our proposal for **Accurate Volume Rendering of Unstructured Hexahedral Meshes**, and Chapter 4 our proposal for **Fast Volume Rendering of Unstructured Hexahedral Meshes**. Chapter 5 presents a discussion about the results and Chapter 6 presents our conclusion and future work.

2
Related Work

2.1
Ray integration

The emission-absorption optical model, proposed by Williams and Max (17), computes the interaction between the light and the volume, within each cell, using the following equation:

$$\begin{aligned} I(t_b) &= I(t_f)e^{-(\int_{t_f}^{t_b}\rho(f(t))dt)} \\ &+ \int_{t_f}^{t_b} e^{-(\int_{t}^{t_b}\rho(f(u))du)}\kappa(f(t))\rho(f(t))dt \end{aligned} \tag{2-1}$$

where t_f and t_b are the ray length from the eye to the entry and exit points of a cell, respectively; $f(t)$ is the scalar function inside the cell, along the ray; $\rho(t)$ is the light attenuation factor, and $\kappa(t)$ is the light intensity, both given by a transfer function.

Evaluating such integral accurately and efficiently is one of the main difficulties faced by volume rendering algorithms. Williams et al. (18) first proposed to simplify the transfer function as a piecewise linear function. They introduced the concept of *control points*, which represent points where the transfer function (TF) is non-linear (the TF in Figure 3.1(b) presents, for example, 3 control points inside the interval). A later work by Röttger et al. (13) proposed to use pre-integration for tetrahedral meshes, storing the parameterized result in a texture, accessed by the entry scalar value, exit scalar value, and ray length. Their proposal works for any transfer function, but any change on the transfer function requires the pre-integration to be recomputed. Röttger (12) later proposed to utilize the GPU to accelerate the precomputation of the 3D table. Moreland et al. (10) re-parameterized the pre-integration result, turning it independent of the transfer function, but under the assumption that the transfer function was piecewise linear. The pre-integration result was then stored in a 2D texture, accessed via the normalized values of s_f and s_b (the scalar value at the entry and exit points of the cell). However, the pre-integration results are computed by assuming a linear scalar field variation inside the cell.

A ray-casting algorithm for unstructured meshes was first presented by Garrity et al. (4). Weiler et al. (15) later proposed a GPU solution using shaders. The main idea is to cast a ray for each screen pixel, and

then to traverse the intersecting cells of the mesh until the ray exits the volume. In order to properly traverse the mesh, one has to store an adjacency data structure in textures; the algorithm will then fetch these textures and determine to which cell it has to step to. At each traversal step, the contribution of the cell to the final pixel color is given by evaluating the ray integral (Equation (2-1)).

2.2 Hexahedral meshes

Volume rendering of unstructured hexahedral meshes was explored by Shirley et al. (14) and Max et al. (8), where they proposed to subdivide each hexahedron into five or six tetrahedra in order to properly render the volumetric data, approximating the trilinear scalar variation by a piecewise linear function. This not only increases the memory consumption but also decreases the rendering quality. Carr et al. (2) focused on regular grids and discuss schemes for subdividing a hexahedral mesh into a tetrahedral one, comparing rendering quality for isosurface and volume rendering.

One of the first proposals to consider something more elaborated than a simple subdivision scheme was made by Williams et al. (18); the authors, however, focused on cell projection of tetrahedral meshes, only making small notes about how the algorithm could handle hexahedral cells, but did not discuss the results. Recently, Marmitt et al. (7) proposed an hexahedral mesh ray-casting, focusing on the traversal between the elements, but neglecting to mention how they integrated the ray considering the trilinear scalar function of a hexahedron.

Marchesin et al. (6) and El Hajjar et al. (5) proposed solutions to structured hexahedral meshes, focusing on how the ray can be integrated over the trilinear scalar function of a regular hexahedron. Marchesin et al. (6) proposed to approximate the trilinear function by a bilinear one. They then stored a pre-integration table in a 3D texture, where each value was accessed by the scalar values at the ray enter, middle, and exit points. To avoid the use of a 4D texture, they consider a constant ray step size. El Hajjar et al. (5) approximated the trilinear scalar function by a linear one and used the same pre-integration table proposed by (13), accessed via the scalar values at enter and exit points, and the ray length. Differently from our proposal, they don't consider the minimum and maximum values of the scalar function to choose adequate the integration intervals.

These papers made the assumption that the scalar function is either linear or bilinear to calculate an integral that could be stored in a texture with feasible

dimensions. Also, their proposals do not support interactive modifications of the transfer function, because the pre-integration table must be recalculated for each TF change.

In this thesis, we avoid the use of pre-integration and propose the use of a quadrature approach to integrate the ray, supporting interactive modifications of the TF. We consider the actual trilinear scalar function and thus achieve accurate results. We also propose another method that approximates the trilinear scalar function by a series of linear ones, but considering the minimum and maximum values of the scalar function along the ray (in a cell), sacrificing accuracy but increasing performance.

3
Accurate Volume Rendering of Hexahedral Meshes

Our hexahedral ray-casting was also presented in (9) and takes into consideration the trilinear variation of the scalar field. Traditional hexahedral ray-casting algorithms divide an hexahedron into five or six tetrahedra, thus increasing the memory footprint and losing rendering quality. We detail our proposal in the next sections, focusing on how we handle the ray integration (Sections 3.1 and 3.2), data structure (Section 3.3), ray traversal (Section 3.4) and isosurface rendering (Section 3.5). We expose our final algorithm in Section 3.6.

3.1
Ray integration

The trilinear scalar function inside an hexahedron cell can be described with the following equation:

$$\begin{aligned} f(x,y,z) &= c_0 + c_1x + c_2y + c_3z \\ &+ c_4xy + c_5yz + c_6xz + c_7xyz \end{aligned} \tag{3-1}$$

where $c_0, c_1, c_2, c_3, c_4, c_5, c_6, c_7$ are the cell coeficients. They are calculated solving the following linear system:

$$\begin{pmatrix} 1 & x_0 & y_0 & \cdots & x_0y_0z_0 \\ 1 & x_1 & y_1 & \cdots & x_1y_1z_1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_7 & y_7 & \cdots & x_7y_7z_7 \end{pmatrix} \begin{pmatrix} c_0 \\ \vdots \\ c7 \end{pmatrix} = \begin{pmatrix} s_0 \\ \vdots \\ s_7 \end{pmatrix}$$

where $\{x_i, y_i, z_i\}$, with $i = \{0, \ldots, 7\}$, are the hexahedron cell vertex positions, and s_i are the scalar values at each one of the vertices. The system can be solved using singular value decomposition (SVD) (11).

The position of a point inside the cell, along the ray, can be described as:

$$\mathbf{p} = \mathbf{e} + t\vec{d} \tag{3-2}$$

where $\mathbf{e}$ is the eye position, and $\vec{d}$ is the ray direction.

We then parameterize the hexahedral scalar function (Equation (3-1)) by the ray length inside the cell, denoted by t:

$$f(t) = w_3t^3 + w_2t^2 + w_1t + w_0, t \in [t_{back}, t_{front}] \tag{3-3}$$

with:

$$\begin{aligned}
w_0 &= c_0 + c_1 e_x + c_2 e_y + c_4 e_x e_y + c_3 e_z \\
&+ c_6 e_x e_z + c_5 e_y e_z + c_7 e_x e_y e_z \\
&+ c_7 d_x d_y e_z \\
w_1 &= c_1 d_x + c_2 d_y + c_3 d_z + c_4 d_y e_x + c_6 d_z e_x \\
&+ c_4 d_x e_y + c_5 d_z e_y + c_7 d_z e_x e_y + c_6 d_x e_z + c_5 d_y e_z \\
&+ c_7 d_y e_x e_z + c_7 d_x e_y e_z \\
w_2 &= c_4 d_x d_y + c_6 d_x d_z + c_5 d_y d_z + c_7 d_y d_z e_x + c_7 d_x d_z e_y \\
w_3 &= c_7 d_x d_y d_z
\end{aligned} \tag{3-4}$$

Considering now the ray integral from Equation (2-1) and considering the transfer function as a piecewise linear function, we can express:

$$\kappa(t) = \frac{(\kappa_{back} - \kappa_{front}) * (f(t) - f(t_{front}))}{f(t_{back}) - f(t_{front})} + \kappa_{front} \tag{3-5}$$

$$\rho(t) = \frac{(\rho_{back} - \rho_{front}) * (f(t) - f(t_{front}))}{f(t_{back}) - f(t_{front})} + \rho_{front} \tag{3-6}$$

we consider t_{back}, and t_{front} as the interval inside the cell with a linear variation of the transfer function.

Getting back to Equation (2-1), we use a Gauss-Legendre Quadrature method to integrate the color and opacity along the ray:

$$\begin{aligned}
I(t_{back}) &= I(t_{front}) e^{-z_{t_{front}, t_{back}}} \\
&+ \int_{t_{front}}^{t_{back}} e^{-z_{t, t_{back}}} \kappa(t) \rho(t) dt
\end{aligned} \tag{3-7}$$

where

$$\begin{aligned}
z_{a,b} &= \int_a^b \rho(r) dr \\
z_{a,b} &= \rho_{front}(b-a) + \frac{(\rho_{back} - \rho_{front})}{12(s_{back} - s_{front})} \\
&* [12(a s_{front} - b s_{front}) + 12 w_0 (b-a) + 6 w_1 (b^2 - a^2) \\
&+ 4 w_2 (b^3 - a^3) + 3 w_3 (b^4 - a^4)]
\end{aligned} \tag{3-8}$$

going back to Equation 3-7, we have:

$$I(t_b) = I(t_{front})e^{-z_{t_{front},t_{back}}} + \sum_{i=0}^{3} D * GaussWeight_i * e^{-z_{t_g,tb}}\kappa(t_g)\rho(t_g) \quad (3\text{-}9)$$

where $D = (t_{back} - t_{front})$ and $t_g = t_{front} + GaussPoint_i * D$. $GaussPoint_i$ and $GaussWeight_i$ are the pre-computed points and weights for the Gauss-Legendre Quadrature.

The Gaussian quadrature integration method gives exact solutions for functions that are well approximated by polynomials up to degree 5, considering a 3 point quadrature. Since we split our integration into several intervals, as we shall explain in Section 3.2, we make sure that our ray integral is accurately evaluated.

3.2 Integration intervals

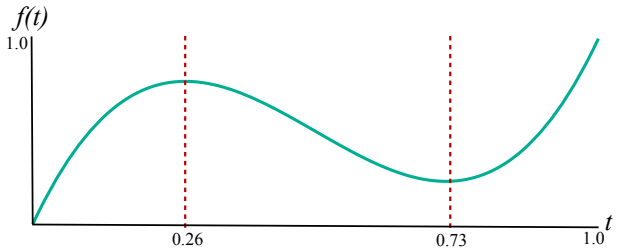
The scalar field variation along a ray in a hexahedron cell can be illustratively represented by the function in Figure 3.1(a). As a cubic polynomial function (according to Equation (3-3)), $f(t)$ has at most two extrema, which can be calculated from its derivative $f'(t)$, a quadratic polynomial. Considering t_{min} and t_{max} the values of minimum and maximum, we calculate t_{near} and t_{far}, such that $t_{near} = min(t_{min}, t_{max})$ and $t_{far} = max(t_{min}, t_{max})$. If t_{front} denotes the point the ray enters the cell and t_{back} the point it exits the cell, the function in the intervals $[t_{front}, t_{near}]$, $[t_{near}, t_{far}]$, and $[t_{far}, t_{back}]$ is monotonic, so each one of these intervals has, at most, one root of Equation (3-3).

To find if there is an isovalue (control point) inside an interval, we use a 2D texture first proposed by Röttger et al. (13) for his tetrahedral cell-projection algorithm. Given s_0 and s_1 (scalars at the interval limit points) as parameters, this texture returns the value of the first control point s_{cp}, if one exists, such that $s_0 < s_{cp} < s_1$ or $s_1 < s_{cp} < s_0$. We discuss how the texture is built in Appendix A. Instead of a 2D texture, we implement it as a 1D texture.

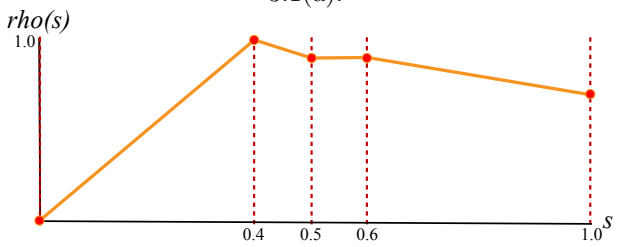
With s_{cp}, we can find the value of t_{cp}, which is the ray length from the eye to the control point that crosses the hexahedron. Considering a trilinear variation of the scalar field, we find it by solving Equation (3-3) for $f(t_{cp}) = s_{cp}$ using the Newton-Raphson method. One of the main problems with such root finding method is its dependency of an initial guess, but we can use the average ray length between the interval limits as our initial guess.

To exemplify this procedure, let us consider the scalar variation inside an hexahedron given by the function in Figure 3.1(a) and the transfer function

illustrated in Figure 3.1(b). We have $t_{max} = 0.26$ and $t_{min} = 0.73$; we then calculate $t_{near} = min(t_{min}, t_{max}) = 0.26$ and $t_{far} = max(t_{min}, t_{max}) = 0.73$. In this case, there is no control point in $[t_{front}, t_{near}]$, and so the first integration interval is $[0, 0.26]$. In $[t_{near}, t_{far}]$, there are three control points: 0.4, 0.5, and 0.6. These give us four integration intervals: $[0.26, 0.4]$, $[0.4, 0.5]$, $[0.5, 0.6]$, and $[0.6, 0.73]$. Beyond t_{far}, there is no other control point, giving us only an additional interval to complete this illustrative integration: $[0.73, 1]$.



3.1(a):



3.1(b):

Figure 3.1: Example of scalar field variation inside a hexahedral cell: (a) Maximum and minimum values of a trilinear function along the ray inside an hexahedron; (b) Transfer function represented by a piecewise linear variation.

3.3 Data structure

In order to access information such as normals and adjacency, we use a set of 1D textures, presented in Table 3.1.

Table 3.1: Data structure for one hexahedral cell.

Texture	Data			
$Coef_i, i = \{0, ..., 7\}$	c_0	c_1	$\cdots$	c_7
$Adj_i, i = 0, ...5$	adj_0	adj_1	$\ldots$	adj_5
$\vec{p}_{i,j}, i = \{0, ..., 5\}, j = \{0, 1\}$	$vecn_{0,0}$	$vecn_{0,1}$	$\ldots$	$vecn_{5,1}$

As mentioned, we need to store 8 coefficients per cell. For adjacency information, we need more 6 values, each associated to a face of the cell. The third line in the table represents plane equations defined by the cell faces. To compute the intersection of the ray with a hexahedron cell, we use a simple ray-plane intersection test. We then need to split each quadrilateral face of a

cell into two triangles and store the corresponding plane equations, totaling 12 planes per cell (48 coefficient values).

We then store a total of 62 values associated to each cell. Considering 4 bytes per value, we store 248 bytes per cell. Even optimized data structures, such as the one described by Weiler et al. (16), requires at least 380 or 456 bytes per cell (considering a hexahedron subdivided into five or six tetrahedra, respectively). In fact, one great advantage of ray-casting hexahedron cells is its small memory consumption when compared to the subdivision scheme.

3.4 Ray traversal

To begin the ray traversal through the mesh, we follow the work by Weiler et al. (16) and Bernardon et al. (1). They proposed a ray-casting approach based on depth-peeling that handles models with holes and gaps. The initial step consists in rendering to a texture the external volume boundary, storing the corresponding cell ID for each pixel on the screen. The second step will then fetch the texture and initiate the mesh traversal starting at the stored cell. The algorithm then proceeds by traversing the mesh until the ray exits the volume. In models with holes or gaps, the ray can re-enter the volume. At each peel, the ray re-enters at the volume boundary, and we accumulate the color and opacity from previous peels. The ray-casting algorithm is finished when the last external boundary of the model is reached.

3.5 Isosurfaces

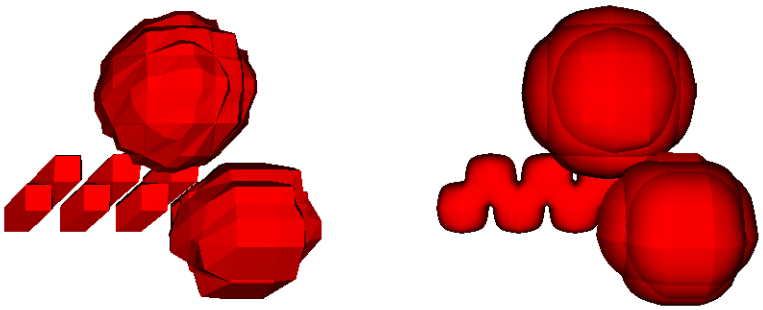
We can extend our volume rendering algorithm to also handle isosurface rendering. This is, in fact, very simple, because we already compute all control points along the ray. The surface normal is given by the gradient of the scalar field:

$$\begin{aligned} \vec{n} &= \nabla f(x,y,z) = < \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} > \\ \vec{n} &= \begin{bmatrix} c_1 + c_4 y + c_6 z + c_7 yz \\ c_2 + c_4 x + c_5 z + c_7 xz \\ c_3 + c_5 x + c_6 y + c_7 xy \end{bmatrix} \end{aligned} \tag{3-10}$$

where (x, y, z) is the intersection between the ray and the iso-surface, and is given by Equation (3-2), considering t_{cp}.

Figures 3.2 presents an isosurface rendering of the Atom9 Dataset, from (2). We chose to use the same isovalue (0.12) as the one used by the authors of

the paper. As can be noted, if compared to a simple subdivision scheme, our proposal depicts the isosurface shape with significant improved accuracy.



3.2(a): Tetrahedral approach 3.2(b): Our proposal

Figure 3.2: Isosurface rendering of the Atom9 dataset.

3.6 Algorithm Overview

The algorithm in Table 3.2 summarizes our approach. We traverse the mesh accumulating each cell contribution to the pixel color. We first calculate the values of minima and maxima of the ray as it goes through each cell, clamping values outside the cell boundary. t_{near} and t_{far} represent the closest and farthest min/max value to the eye position. We then iterate through t_{front}, t_{near}, t_{far}, t_b, fetching the texture described in Section 3.2 to find if there are control points in each interval. If there is, the algorithm integrates from the current position t_i to t_{cp}; we then update the value of t_i.

In order to accurately render the volume, we used double precision. We tried to minimize the amount of double operations, and restricted them to the Newton Root finding method, Quadrature Integration and Hexahedron Coeficients calculation, because of its high cost.

Table 3.2: Ray-casting Algorithm

1: $color \leftarrow (0, 0, 0, 0)$
2: $cell.id \leftarrow VolumeBoundary()$
3: **while** $color.a < 1$ and ray inside volume **do**
4: $t_{back}, cell.nid \leftarrow IntersectRayFaces(t, cell)$
5: $t_{min}, t_{max} = Solve(cell.f'(t) = 0)$
6: $t_{min} = clamp(t_{min}, t_{front}, t_{back})$
7: $t_{max} = clamp(t_{max}, t_{front}, t_{back})$
8: $t_{near} = min(t_{min}, t_{max})$
9: $t_{far} = max(t_{min}, t_{max})$
10: $t_i = [t_{front}, t_{near}, t_{far}, t_{back}]$
11: $i = 0$
12: **while** $i < 3$ **do**
13: {Find control points}
14: $s_{cp} = fetch(s_{cp}, s_{i+1})$
15: **if** $s_i < s_{cp} < s_{i+1}$ *or* $s_i > s_{cp} > s_{i+1}$ **then**
16: $t_{cp} = Newton(cell.f(t) = s_{cp}, \frac{t_i + t_{i+1}}{2})$
17: **else**
18: $t_{cp} = t_{i+1}$
19: **end if**
20: $color \leftarrow Integrate(t_i, s_i, t_{cp}, s_{cp})$
21: $t_i = t_{cp}$
22: **if** $t_i >= t_{i+1}$ **then**
23: $i++$
24: **end if**
25: **end while**
26: $cell.id = cell.nid$
27: $t = t_{back}$
28: **end while**

4
Fast Volume Rendering of Hexahedral Meshes

In order to improve performance and sacrifice a certain rendering accuracy, we also propose a linear approximation of the scalar field, considering certain integration intervals. In this section, we will focus only in the difference between the two aproaches. More specifically, we will detail our linear ray integration and the differences between the integration intervals.

4.1
Ray integration

Like our previous approach, we also consider the trilinear variation of the scalar field. The difference, however, lies in how we calculate the Ray Equation (2-1). We use a 2D pre-integrated table, first proposed by Moreland et al. (10) to render unstructured tetrahedral meshes. We discuss such table in Appendix B.

In order to use the 2D table, we make the assumption that the trilinear scalar function can be approximated by a piecewise linear function. The piecewise linear functions are composed by the intervals delimited by the maximum and minimum scalar of the hexahedron scalar function, as can be seen in Figure 4.1.

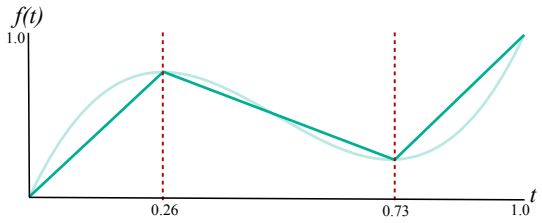


Figure 4.1: Piecewise linear functions, considering the maximum and minimum values of a trilinear scalar function.

It may sound counter intuitive to go back to a piecewise linear scalar function, as the basis of this thesis is to propose something better than the hexahedral division into tetrahedra, also a piecewise linear scalar function. But this is a different approach, as we first consider the trilinear scalar function to find the maximum and minimum values, and only linearizes the scalar function to calculate the color and opacity values. As we will show later, such approximation results in better quality and performance than the hexahedral division approach.

4.2 Integration intervals

The integration intervals are exactly the same as discussed in Section 3.2. The only minor difference is how we find t_{cp}, given s_{cp}. Because we are considering a linear variation inside the integration interval, it is enough to calculate t_{cp} according to Equation (4-1).

$$t_{cp} = t_0 + (t_1 - t_0) * (\frac{|s_{cp} - s_0|}{s_1 - s_0}) \tag{4-1}$$

where t_0 and t_1 are the ray length at the front and back of the integration interval, s_0 and s_1 are the scalar at the front and back of the integration interval.

5
Results

For the evaluation of our proposal, we implemented a ray-casting using CUDA that handles both tetrahedral and hexahedral meshes. We measured the rendering performance with four known volumetric datasets: Blunt-fin, Fuel, Neghip, Oxygen. We tested the following algorithms:

1. **Hexa**$_{CONST}$: Ray-casting for hexahedral meshes that uses a fixed number of 100 steps in each cell to accurately compute the illumination assuming a constant scalar field at each step.

2. **Hexa**$_{ACC}$: Our proposal for accurate volume rendering of hexahedral meshes.

3. **Hexa**$_{FAST}$: Our proposal for fast volume rendering of hexahedral meshes.

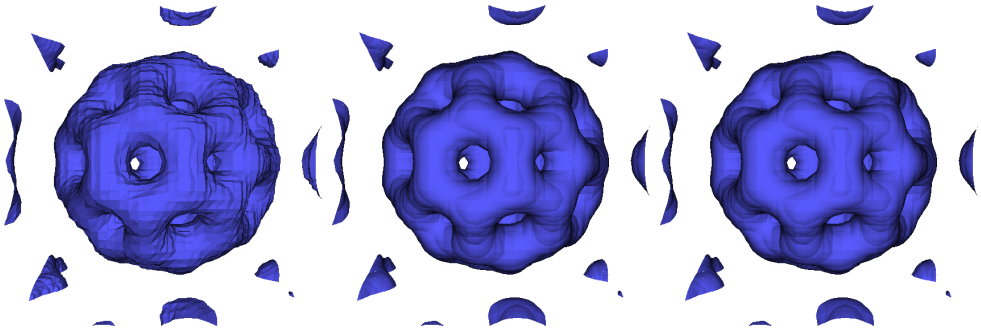
4. **Tetra**$_{HARC}$: Ray-casting using a pre-integrated table (3), with each hexahedral cell subdivided in six tetrahedra.

Although the $Hexa_{Const}$ algorithm is far from being an efficient solution, it produces accurate results, and we use it as our rendering quality reference. We ran the experiments on Windows 7 with an Intel Core 2 Duo 2.8 GHz, 4 GB RAM and a GeForce 460 GTX 1 GB RAM. The screen size was set to 800 x 800 pixels in all experiments.

In Section 5.1.1 we compare a tetrahedral and our hexahedral proposal for isosurface rendering regarding quality. Then, in Section 5.1.2, we compare the volume rendering quality. Section 5.2 presents a time comparison among the algorithms.

5.1
Rendering quality

In this section, we will discuss how our proposals compares with the hexahedron subdivision approach, both in isosurface rendering and volume rendering.



5.1(a): $Tetra_{HARC}$ 5.1(b): $Hexa_{ACC}$ 5.1(c): $Hexa_{FAST}$

Figure 5.1: Bucky dataset isosurfaces.

5.1.1 Isosurface rendering

Figure 5.1 compares the isosurface of the Bucky dataset.

As can be noted, our proposal presents smoother isosurfaces, as expected, since we consider a trilinear scalar function. Both algorithms ($Hexa_{ACC}$ and $Hexa_{FAST}$) presents similar results.

5.1.2 Volume Rendering Quality

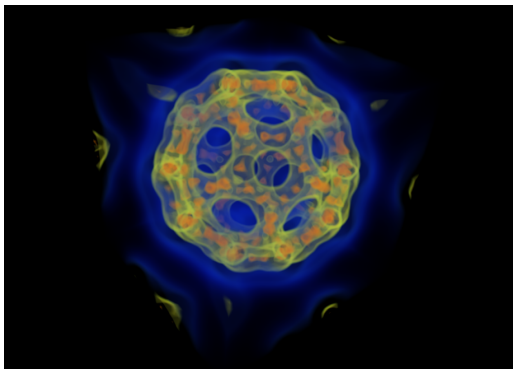
We evaluate our ray-casting algorithms regarding rendering quality. We first devised an experiment using synthetic volume data composed by only one hexahedral cell. Figure 5.2 shows the scalar values set to each vertex and presents the four images achieved by the four algorithms. For these images, we used a transfer function with six thin spikes, isolating six different slabs. Even though such a scalar field variation is rare in actual datasets, this synthetic test aims to show how the subdivision of an hexahedron can lead to unrecognizable results. The test also demonstrates that both of our proposals are capable of rendering scalar fields with complex variations.

Figure 5.3 shows and compares the results achieved by the four algorithms for rendering the Bucky model. Figures 5.3(e), 5.3(f) and 5.3(g) shows the difference between the images achieved with our two proposals and the subdivision scheme when compared to our quality reference. It's possible to notice a clear difference between the rendering images, with our proposals more similar to our quality reference. The $\mathbf{Hexa}_{FAST}$ proposal presents a subtle difference, expected because of our approximations of the ray integral.

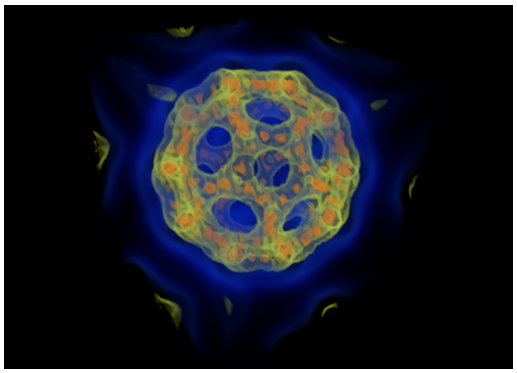
Figure 5.4 shows the result of a similar experiment but now considering the Bluntfin model. As can be noted, the results are equivalent: our algorithm produces images with a better quality. In this example, our two proposal present similar results.

5.2(a): Scalar values at vertices | 5.2(b): $\mathbf{Hexa}_{CONST}$ | 5.2(c): $\mathbf{Tetra}_{HARC}$ | 5.2(d): $\mathbf{Hexa}_{FAST}$ | 5.2(e): $\mathbf{Hexa}_{ACC}$

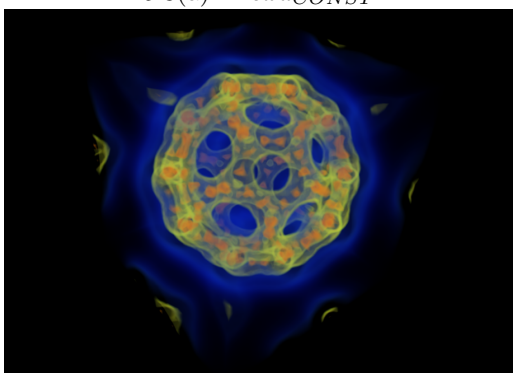
Figure 5.2: Rendering on a synthetic model composed by on hexahedron.



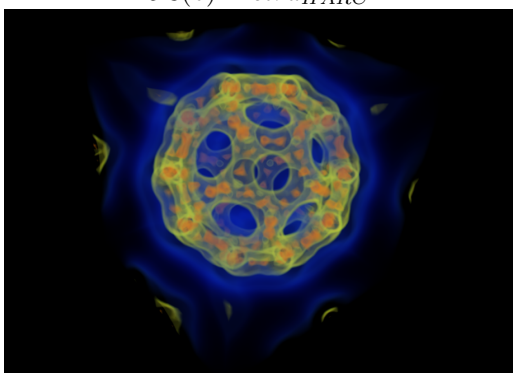
5.3(a): $Hexa_{CONST}$



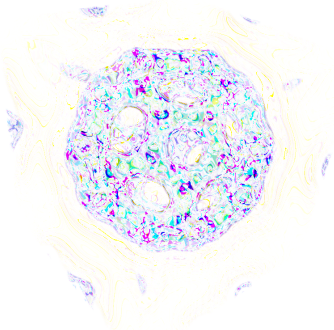
5.3(b): $Tetra_{HARC}$



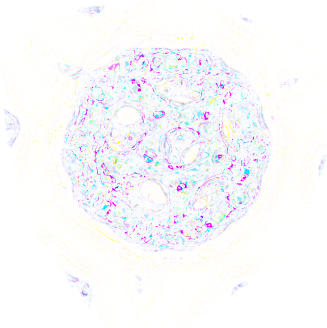
5.3(c): $Hexa_{FAST}$



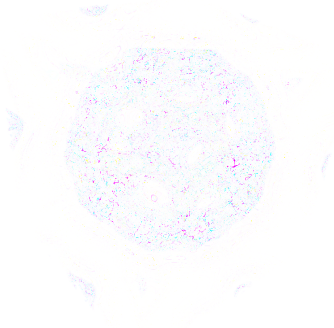
5.3(d): $Hexa_{ACC}$



5.3(e): Difference $Tetra_{HARC}$

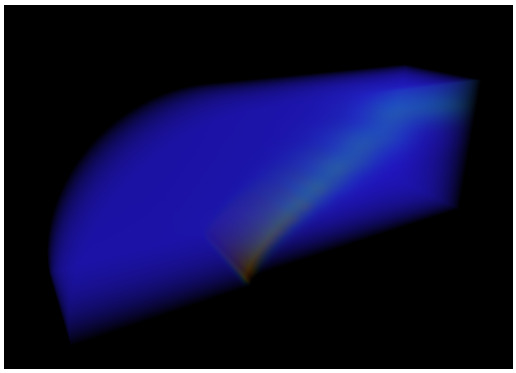


5.3(f): Difference $Hexa_{FAST}$

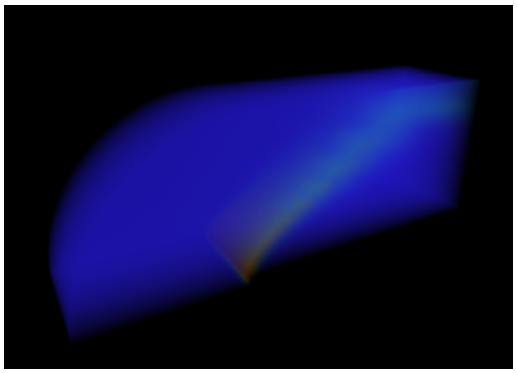


5.3(g): Difference $Hexa_{ACC}$

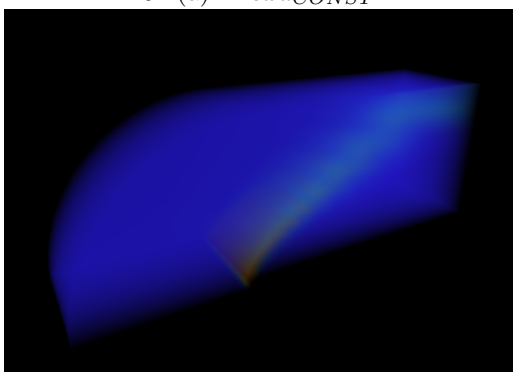
Figure 5.3: Images of the Bucky model.



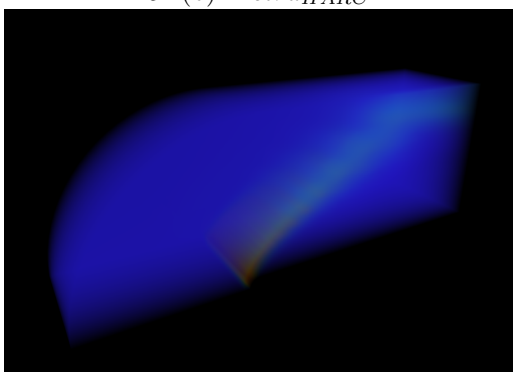
5.4(a): $Hexa_{CONST}$



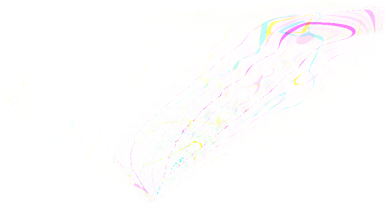
5.4(b): $Tetra_{HARC}$



5.4(c): $Hexa_{FAST}$

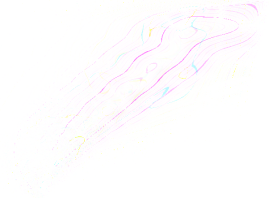


5.4(d): $Hexa_{ACC}$



5.4(e): Difference $Tetra_{HARC}$

5.4(f): Difference $Hexa_{FAST}$



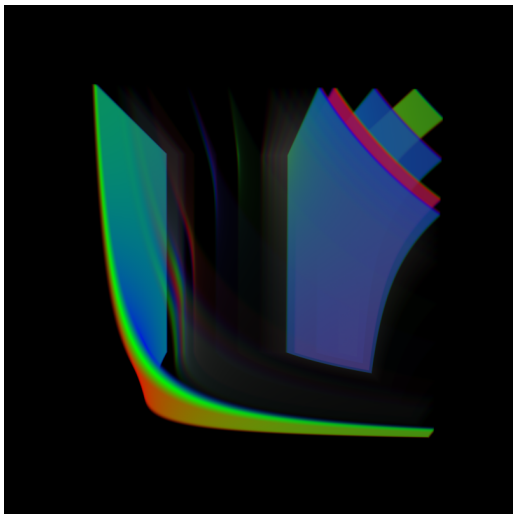
5.4(g): Difference $Hexa_{ACC}$

Figure 5.4: Images of the Bluntfin model.

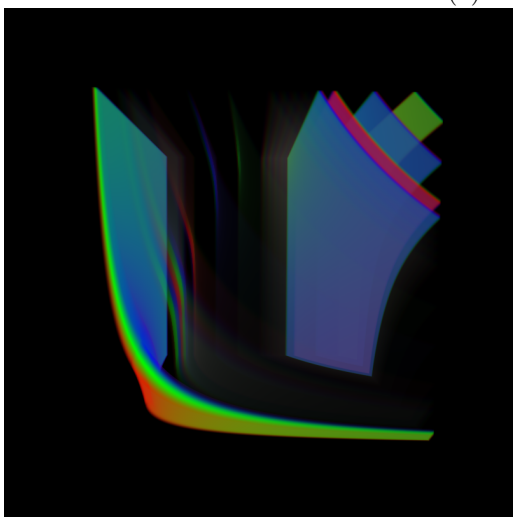
5.1.3
Comparassion with regular data rendering

We also compared our algorithm with a regular data ray-casting, detailed in Appendix D. Figure 5.5 shows the difference between the different algorithms. To highlight the differences, we used a transfer function with 100 control points.

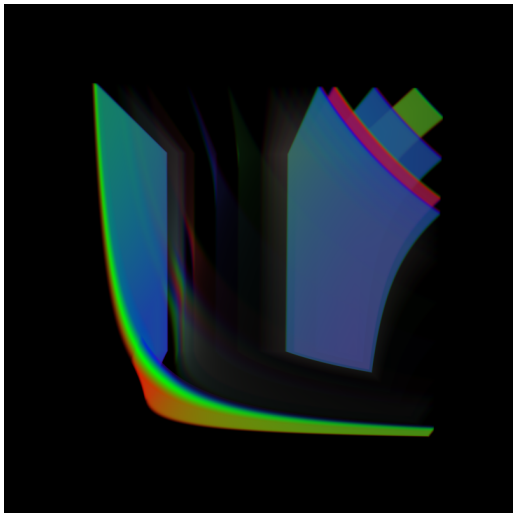
As can be seen, our proposal (both using quadrature and linear approximation) presents a better representation of the volume. Due to the trilinear interpolation during the 3D texture fetch, the rendering of the regular ray-casting algorithm does not appear as smooth as our proposal. Another problem is that, even with a 100 steps pre-integrated table, the regular ray-casting algorithm misses some of the transfer function control points, unlike our proposal, that stops at every control point.



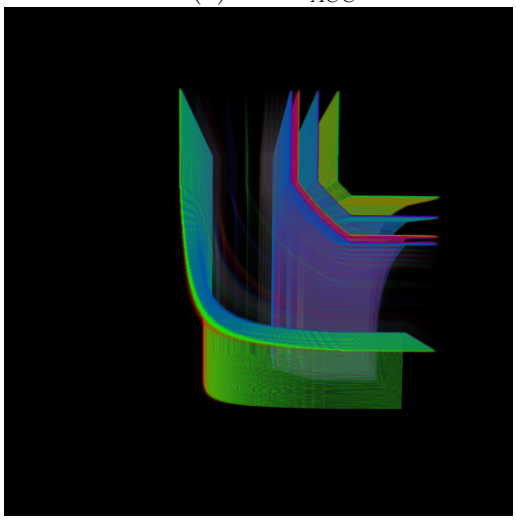
5.5(a): $Hexa_{Const}$



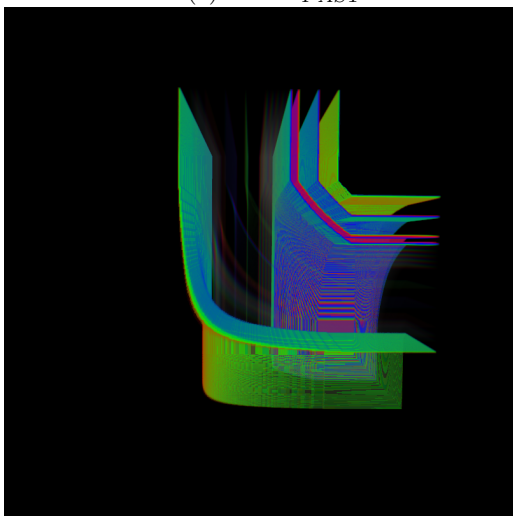
5.5(b): $Hexa_{ACC}$



5.5(c): $Hexa_{FAST}$



5.5(d): Regular ray-casting, using pre-integrated table



5.5(e): Regular ray-casting

Figure 5.5: Achieved images using a synthetic model.

5.2 Time results

Table 5.2 shows a comparison of memory consumption and rendering time. The time reported in the table represents the rendering time for one frame. For each entry, we repeated the experiment 5 times, and, for each run, we changed the camera to 64 different positions, averaging the measured time.

As can be noted, when compared to the subdivision scheme of 6 tetrahedra per hexahedron cell, our proposal reduces memory consumption by a factor of 2.2. Our algorithm also presents competitive performance. The rendering time of our Hex_{ACC} algorithm is in average 12% worse than the algorithm based on the subdivision scheme. To achieve accurate results, we need to perform part of the integration computation in double precision. That is the main reason for losing performance. If we had used single precision, our algorithm would be 15 to 20% faster than the subdivision scheme, but this would bring numerical inaccuracy. Figure 5.6 presents a comparison of the Bluntfin model rendered with both single and double precision. As can be noted, the use of single precision does result in inaccurate results.

Our $Hexa_{FAST}$ algorithm, however, presents a performance around 12% better than the $Tetra_{HARC}$ algorithm.

Model	Algorithm	# cells	Mem (MB)	Time (ms)
Fuel	$Tetra_{HARC}$	1,572,864	144	76.08
	$Hexa_{ACC}$	262,144	62	79.23
	$Hexa_{FAST}$	262,144	62	59.74
Neghip	$Tetra_{HARC}$	1,572,864	144	89.82
	$Hexa_{ACC}$	262,144	62	109.23
	$Hexa_{FAST}$	262,144	62	80.47
Oxygen	$Tetra_{HARC}$	658,464	60.2	32.46
	$Hexa_{ACC}$	109,744	25.9	35.73
	$Hexa_{FAST}$	109,744	25.9	29.62
Bluntfin	$Tetra_{HARC}$	245,760	22.5	27.64
	$Hexa_{ACC}$	40,960	9.6	31.60
	$Hexa_{FAST}$	40,960	9.6	25.89

Table 5.1: Rendering times and memory footprint of the subdivision scheme and our proposal.

5.6(a): Single precision 5.6(b): Double precision

Figure 5.6: Volume rendering of the Bluntfin Dataset.

6
Conclusion

We have presented an accurate hexahedral volume rendering suitable for unstructured meshes. Our proposal integrates the trilinear scalar function using a quadrature approach on the GPU. Although our performance was in average 12% worse than a tetrahedral algorithm, our proposal produces images with better quality. By the use of a proper integration of the trilinear function inside an hexahedron, our proposal ensures that no isosurface value is missed during integration. We also presented a fast and high-quality algorithm for hexahedral meshes that, although does not achieve images as accurate as the ones using quadrature integration, achieves good approximations and is also faster than a tetrahedral volume rendering.

Because of its parallel nature, we believe that ray-casting is better suited for massively parallel environments, such as the GPU. Its main drawback, however, is its memory consumption; our algorithm presents a smaller memory footprint than regular hexahedral subdivision schemes.

In the future, we plan to study the impact of a ray-bilinear patch intersection test, and to analyze how it could improve the image quality and its impact on the performance. We also plan to extend the algorithm to higher-order cells and to investigate its use for rendering models that support adaptive level-of-detail.

7
Bibliography

[1] BERNADON, F. F.; PAGOT, C. A.; COMBA, J. L. D. ; SILVA, C. T. **Journal of Graphics, GPU, and Game Tools**. Gpu-based tiled ray casting using depth peeling, journal, v.11, n.4, p. 1–16, 2006.

[2] CARR, H.; MOLLER, T. ; SNOEYINK, J. **IEEE Transactions on Visualization and Computer Graphics**. Artifacts caused by simplicial subdivision, journal, v.12, p. 231–242, March 2006.

[3] ESPINHA, R.; CELES, W. **High-quality hardware-based ray-casting volume rendering using partial pre-integration**. In: PROCEEDINGS OF THE XVIII BRAZILIAN SYMPOSIUM ON COMPUTER GRAPHICS AND IMAGE PROCESSING, p. 273–, Washington, DC, USA, 2005. IEEE Computer Society.

[4] GARRITY, M. P. **Raytracing irregular volume data**. In: PROCEEDINGS OF THE 1990 WORKSHOP ON VOLUME VISUALIZATION, VVS '90, p. 35–40, New York, NY, USA, 1990. ACM.

[5] HAJJAR, J. E.; MARCHESIN, S.; DISCHLER, J. ; MONGENET, C. **Second order pre-integrated volume rendering**. In: IEEE PACIFIC VISUALIZATION SYMPOSIUM, March 2008.

[6] MARCHESIN, S.; DE VERDIERE, G. **Visualization and Computer Graphics, IEEE Transactions on**. High-quality, semi-analytical volume rendering for amr data, journal, v.15, n.6, p. 1611 –1618, nov.-dec. 2009.

[7] MARMITT, G.; SLUSALLEK, P. **Fast Ray Traversal of Tetrahedral and Hexahedral Meshes for Direct Volume Rendering**. In: PROCEEDINGS OF EUROGRAPHICS/IEEE-VGTC SYMPOSIUM ON VISUALIZATION (EUROVIS), Lisbon, Portugal, May 2006.

[8] MAX, N. L.; WILLIAMS, P. L. ; SILVA, C. T. **Cell projection of meshes with non-planar faces**. In: DATA VISUALIZATION: THE STATE OF THE ART, p. 157–168, 2003.

[9] MIRANDA, F. M.; CELES, W. **Accurate volume rendering of unstructured hexahedral meshes**. In: Lewiner, T.; Torres, R., editors, SIBGRAPI 2011 (24TH CONFERENCE ON GRAPHICS, PATTERNS AND IMAGES), p. 93–100, Maceió, AL, august 2011. IEEE.

[10] MORELAND, K.; ANGEL, E. **A fast high accuracy volume renderer for unstructured data**. In: PROCEEDINGS OF THE 2004 IEEE SYMPOSIUM ON VOLUME VISUALIZATION AND GRAPHICS, VV '04, p. 9–16, Washington, DC, USA, 2004. IEEE Computer Society.

[11] PRESS, W. H.; TEUKOLSKY, S. A.; VETTERLING, W. T. ; FLANNERY, B. P. **Numerical recipes in C (2nd ed.): the art of scientific computing**. New York, NY, USA: Cambridge University Press, 1992.

[12] RÖTTGER, S.; ERTL, T. **A two-step approach for interactive pre-integrated volume rendering of unstructured grids**. In: PROCEEDINGS OF THE 2002 IEEE SYMPOSIUM ON VOLUME VISUALIZATION AND GRAPHICS, VVS '02, p. 23–28, Piscataway, NJ, USA, 2002. IEEE Press.

[13] RÖTTGER, S.; KRAUS, M. ; ERTL, T. **Hardware-accelerated volume and isosurface rendering based on cell-projection**. In: PROCEEDINGS OF THE CONFERENCE ON VISUALIZATION '00, VIS '00, p. 109–116, Los Alamitos, CA, USA, 2000. IEEE Computer Society Press.

[14] SHIRLEY, P.; TUCHMAN, A. **A polygonal approximation to direct scalar volume rendering**. In: PROCEEDINGS OF THE 1990 WORKSHOP ON VOLUME VISUALIZATION, VVS '90, p. 63–70, New York, NY, USA, 1990. ACM.

[15] WEILER, M.; KRAUS, M.; MERZ, M. ; ERTL, T. **Hardware-based ray casting for tetrahedral meshes**. In: PROCEEDINGS OF THE 14TH IEEE VISUALIZATION 2003 (VIS'03), VIS '03, p. 44–, Washington, DC, USA, 2003. IEEE Computer Society.

[16] WEILER, M.; MALLON, P. N.; KRAUS, M. ; ERTL, T. **Texture-encoded tetrahedral strips**. In: PROCEEDINGS OF THE 2004 IEEE SYMPOSIUM ON VOLUME VISUALIZATION AND GRAPHICS, VV '04, p. 71–78, Washington, DC, USA, 2004. IEEE Computer Society.

[17] WILLIAMS, P. L.; MAX, N. **A volume density optical model**. In: PROCEEDINGS OF THE 1992 WORKSHOP ON VOLUME VISUALIZATION, VVS '92, p. 61–68, New York, NY, USA, 1992. ACM.

[18] WILLIAMS, P. L.; MAX, N. L. ; STEIN, C. M. **IEEE Transactions on Visualization and Computer Graphics**. A high accuracy volume renderer for unstructured data, journal, v.4, p. 37–54, 1998.

A
Control point texture

To find control points inside monotonic intervals, we use a 1D texture first proposed by Röttger et al. (13) and built as presented in Table A.1. The algorithm receives a transfer function texture (with size $TFTEXTURESIZE$), and array with the control points values ($cpvalues$) and the number of control points (cp_{num}).

The output of the algorithm is an 1D array ($cptexture$) that, given an scalar s, it returns the next two greater control points ($cptexture.x$ and $cptexture.y$) and the previous two smaller control points ($cptexture.z$ and $cptexture.w$).

Table A.1: Control point texture algorithm

```
1: cp_counter ← 0
2: for i ← 0; i < TFTEXTURESIZE do
3:    i_scalar ← i / TFTEXTURESIZE
4:    cp_scalar ← cpvalues[cp_counter]
5:    if cp_counter < cp_num − 1 then
6:       if i_scalar <= cp_scalar then
7:          cptexture[i].x ← cp_scalar
8:          cptexture[i].y ← cpvalues[cp_counter + 1]
9:          i + +
10:      else
11:         cp_counter + +
12:      end if
13:   else
14:      if i_scalar <= cp_scalar then
15:         cptexture[i].x ← cp_scalar
16:         cptexture[i].y ← 3.0
17:      end ifi + +
18:   end if
19: end for
20: cp_counter ← cp_num − 1
21: for i = TFTEXTURESIZE − 1; i > 0 do
22:   i_scalar ← i / TFTEXTURESIZE
23:   cp_scalar ← cpvalues[cp_counter]
24:   if cp_counter > 0 then
25:      if i_scalar >= cp_scalar then
26:         cptexture[i].z ← cp_scalar
27:         cptexture[i].w ← cpvalues[cp_counter − 1]
28:         i − −
29:      else
30:         cp_counter − −
31:      end if
32:   else
33:      if i_scalar >= cp_scalar then
34:         cptexture[i].z ← cp_scalar
35:         cptexture[i].w ← ∞
36:      end ifi − −
37:   end if
38: end for
```

B
2D Pre-integration table texture

A 2D pre-integration table was first proposed by Moreland et al. (10), to evaluate Equation (2-1). The table is the solution to the integral in Equation (2-1) considering the transfer function as piecewise linear function, just like (3-5) and (3-6). Substituing (3-5) and (3-6) in (2-1) we get:

$$\begin{aligned} I(t_b) &= I(t_f)e^{-\int_0^D \tau(t)dt} \\ &+ \kappa(t_b)(-e^{-\int_{t_f}^{t_b} \tau(t)dt} + \frac{1}{D}\int_{t_f}^{t_b} e^{-\int_{s+t_f}^{t_b} \tau(t)dt} ds) \\ &+ \kappa(t_f)(1 - \frac{1}{(t_b - t_f)}\int_{t_f}^{t_b} e^{-\int_{s+t_f}^{t_b} \tau(t)dt} ds) \end{aligned} \tag{B-1}$$

considering the following two repeting terms:

$$\zeta_{(t_b-t_f),\tau(t)} = e^{-\int_{t_f}^{t_b} \tau(t)dt} = \tag{B-2}$$

$$\psi_{(t_b-t_f),\tau(t)} = \frac{1}{D}\int_{t_f}^{t_b} e^{-\int_{s+t_f}^{t_b} \tau(t)dt} ds \tag{B-3}$$

considering a linear variation of the scalar field (represented by $f(t)$ in Equations (3-5) and (3-6)), and also a parametrization of $\tau(s_b - s_f)$ by $\tau(s_b - s_f) = \frac{\gamma}{1-\gamma}$ so that it can fit in a 2D texture in the $[0, 1)$ domain, they find the following equations:

$$\psi_{\gamma_b,\gamma_f} = \int_0^1 e^{-\int_s^1 (\frac{\gamma_b}{1-\gamma_b}(1-t)+\frac{\gamma_f}{1-\gamma_f}t)dt} ds \tag{B-4}$$

$$\zeta_{(t_b-t_f),\tau_b,\tau_f} = e^{-\frac{t_b-t_f}{2}(\tau_b+\tau_f)} \tag{B-5}$$

C
Unstructured Tetrahedral Meshes

We implemented the proposal by Espinha and Celes (3) to render unstructured tetrahedral meshes. We detail the integration scheme in Section C.1, the data structure in Section C.2, the ray traversal in Section C.3, and, finally, an overview of the algorithm in Section C.4.

C.1
Ray integration

To integrate the ray according to (2-1), Espinha and Celes (3) used a 2D pre-integrated table first proposed by Moreland et al. (10). We detail the table in Section B. It considers a linear variation of the scalar function and also a piecewise linear transfer function, so we must stop at every TF control point. That is done by using the 2D table explained in A.

C.2
Data structure

All relevant mesh information is stored in 1D textures, as described in Table C.1. We must store the adjacency information of each cell, its faces normals and also its scalar function. The scalar function inside a tetrahedron is described by the equation presented in (C-3).

$$f(x,y,z) \;=\; c_0 + c_1 x + c_2 y + c_3 z \tag{C-1}$$

Table C.1: Data structure for one tetrahedral cell.

Texture	Data			
$Coef_i, i = \{0,...,3\}$	c_0	c_1	c_2	c_3
$Adj_i, i = 0,...3$	adj_0	adj_1	adj_2	adj_3
$\vec{p}_i, i = \{0,...,3\}$	$vecn_0$	$vecn_1$	$vecn_2$	$vecn_3$

Each tetrahedron occupies 96 bytes of data: 16 bytes for its scalar function, 16 bytes for its adjacency information, and 64 bytes for its normals. Considering that an hexahedron can be divided into either 5 or 6 tetrahedrons, each hexahedron can take from 480 bytes to 576 bytes.

C.3 Traversal

In order to traverse from cell to cell, the algorithm does 4 ray/plane collision tests. Considering the eye position $\mathbf{e}$, the intersection between the ray and the tetrahedron face i is given by the ray parameter t:

$$t = -\frac{(\mathbf{e}.\mathbf{n_i} + o_i)}{\mathbf{t}.\mathbf{n_i}} \tag{C-2}$$

where $(\mathbf{n_{i,j}}, o_{i,j})$ is the plane equation of face i. The exiting point is then given by:

$$\mathbf{p_b} = \mathbf{e} + t\mathbf{d}. \tag{C-3}$$

the scalar at the exit point $\mathbf{p_b}$ is given by Equation (C-3).

C.4 Algorithm Overview

The ray-casting algorithm is summerized in Table C.2.

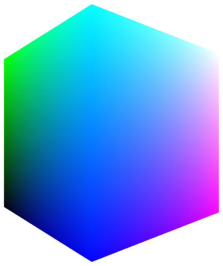
Table C.2: Ray-casting Algorithm

$color \leftarrow (0,0,0,0)$
$cell.id \leftarrow VolumeBoundary()$
while $color.a < 1$ and ray inside volume **do**
  $t_{back}, cell.nid \leftarrow IntersectRayFaces(t, cell)$
  {Find control points}
  $s_{cp} = texture2D(controlpoints, s_{front}, s_{back})$
  if $s_{front} < s_{cp} < s_{back}$ *or* $s_{front} > s_{cp} > s_{back}$ **then**
    $t_{cp} = \frac{s_{cp} - s_{front}}{s_{back} - s_{front}}$
  else
    $t_{cp} = t_{back}$
  end if
  $color \leftarrow Integrate(t_{front}, s_{front}, t_{cp}, s_{cp})$
  $cell.id = cell.nid$
  $t_{front} = t_{back}$
end while

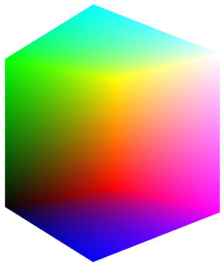
D
Regular data

Regular data can be stored in a 3D texture and sampled along the ray using a constant size step.

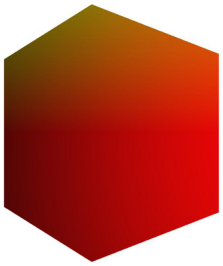
The regular data ray-casting follows the simple steps: Steps 1 and 2 render an unitary cube front and back faces to a texture using a FBO (frame buffer object). These two textures are then passed to a shader in Step 3 that computes the ray direction and ray length, saving the result to another texture. Finally, in Step 4 we trace a ray for each screen pixel, using the ray direction and ray length. The kernel samples the 3D texture using a constant size step until the current ray length reaches the total ray length, as calculated in Step 3.



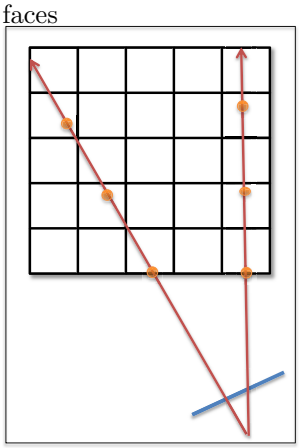
D.1(a): Step 1: cube front faces



D.1(b): Step 2: cube back faces



D.1(c): Step 3: ray direction

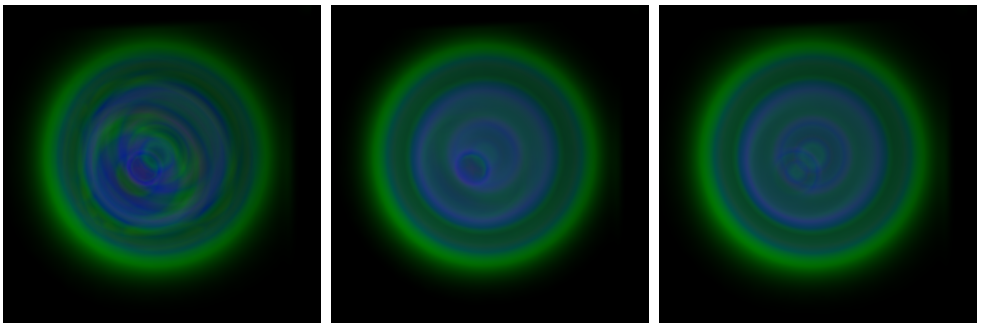


D.1(d): Step 3: ray traversal

Figure D.1: Structured ray-casting steps.

Figure D.2 shows the difference using a pre-integration table and not using one, considering the same step size and number of steps (10). We also show an image using 100 steps(Figure D.2(c)), considered our quality reference.

As can be noted, pre-integration produces the best rendering quality. This is obvious if we consider how the two are obtained; in post-classification,



D.2(a): Without pre-integration, 10 steps

D.2(b): Pre-integration, 10 steps

D.2(c): 100 steps

Figure D.2: Structured ray-casting, using the same number of steps.

we fetch an interpolated scalar value from the volume and then accumulate its color and opacity according to a transfer function, then advance the ray in STEPSIZE length. However, such move can miss an important feature of the volume, like a high spike in the transfer function. Differently, a 3D pre-integrated table can use a small step, because the integration burden is moved to a pre-processing step. It decreases the chance to miss an important feature of the volume.

D.1 3D Pre-integration table texture

Our structured ray-casting uses a 3D pre-integration table to evaluate Equation (2-1), and it is calculated as a pre-processing step on the CPU. The algorithm can be seen in Table D.1. As it is transfer function dependent, it receives the transfer function array as an input.

D.2 Data structure

In order to render the volumetric data, we use the following textures:

- Volume: 3D texture (R) with the scalar values of the dataset.
- Pre-integration table: 3D texture (RGBA) with the pre-integrated ray integral ((2-1)).
- Transfer function: 1D texture (RGBA) with the color scale.

The structured ray-casting fetches the scalar values from a 3D texture, with the dimensions of the datasets. The pre-integration algorithm fetches a 3D pre-integration table, and the post-classification algorithm fetches the 1D transfer function.

Table D.1: 3D pre-integration table algorithm

```
1: for t_counter ← 0; t_counter < SIZE; t_counter++ do
2:   for sf_counter ← 0; sf_counter < SIZE; sf_counter++ do
3:     for sb_counter ← 0; sb_counter < SIZE; sb_counter++ do
4:       c_rgb ← (0, 0, 0)
5:       extinction ← 1.0
6:       dt ← (t_counter * MAXLENGTH / SIZE) / numsteps
7:       for step_counter ← 0; step_counter < numsteps; step_counter++ do
8:         normt ← t / numsteps
9:         scalar ← sf_counter + normt * (sb_counter − sf_counter)
10:        rgba ← transferfunc[s]
11:        c_rgb ← c_rgb rgb * a * extinction * dt
12:        extinction ← extinction * expf(−a * dt)
13:      end for
14:      index ← SIZE * SIZE * t_counter + SIZE * sf_counter + sb_counter
15:      table[index].rgb ← c_rgb
16:      table[index].a ← 1.0 − extinction
17:    end for
18:  end for
19: end for
```

D.3 Algorithm Overview

Tables D.2 and D.3 reviews the regular data ray-casting algorithm, using a post-classification and pre-integration approach.

Table D.2: Structured data post-classification ray-casting algorithm

```
1: color_final ← (0, 0, 0, 0)
2: t = 0
3: while color_final.a < 1 and t < ray_length do
4:   scalar ← texture3D(volume, ray_pos)
5:   color ← texture1D(transferfunction, scalar)
6:   color.w ← color.w * STEPSIZE
7:   color.rgb ← color.rgb * color.w
8:   color_final ← color_final + ((1.0 − color_final.a) * color)
9:   ray_length ← ray_length + STEPSIZE
10:  ray_pos ← ray_pos + STEPSIZE * ray_dir
11: end while
```

Table D.3: Structured data pre-integration ray-casting algorithm

1: $color \leftarrow (0, 0, 0, 0)$
2: $t = 0$
3: $scalar_{front} \leftarrow texture3D(volume, ray_{pos})$
4: **while** $color_{final}.a < 1$ and $t < ray_{length}$ **do**
5: $\quad scalar_{back} \leftarrow texture3D(volume, ray_{pos} + STEPSIZE * ray_{dir})$
6: $\quad color \leftarrow texture3D(preinttable, scalar_{back}, scalar_{front}, STEPSIZE)$
7: $\quad color_{final} \leftarrow color_{final} + ((1.0 - color_{final}.a) * color)$
8: $\quad ray_{length} \leftarrow ray_{length} + STEPSIZE$
9: $\quad ray_{pos} \leftarrow ray_{pos} + STEPSIZE * ray_{dir}$
10: $\quad scalar_{front} \leftarrow scalar_{back}$
11: **end while**