Vis Comput (2012) 28:1005–1014
DOI 10.1007/s00371-012-0742-8

ORIGINAL ARTICLE

Volume rendering of unstructured hexahedral meshes

Fabio Markus Miranda · Waldemar Celes

Published online: 4 July 2012

Abstract 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 a nonadaptive piecewise linear function. This results in inaccurate images and increases the memory consumption. In this paper, 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. 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. We also propose a faster approach that, as the tetrahedron subdivision scheme, also approximates the trilinear variation by a piecewise linear function, but in an adaptive and more accurate way, considering the points of minimum and maximum of the scalar function along the ray.

Keywords Volume rendering · Hexahedral mesh · Unstructured mesh · Ray integral

F.M. Miranda (✉) · W. Celes
Tecgraf/PUC-Rio–Computer Science Department, Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, Brazil
e-mail: fmiranda@tecgraf.puc-rio.br

W. Celes
e-mail: celes@tecgraf.puc-rio.br

1 Introduction

Volume rendering is a popular way to visualize scalar fields from a number of different data, from medical MRI to results of numerical simulations. 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 compute the interaction between light and volume, while maintaining an acceptable rendering time, is one of the main challenges for implementing volume visualization algorithms.

One approach to visualize volumetric data is ray-casting [4]. By tracing a ray for each pixel on the screen, one 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]. The volume dataset is usually decomposed into tetrahedral and hexahedral cells. As the ray traverses the volume, the resulting color and opacity are computed 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 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 nonadaptive piecewise linear function. This results in inaccurate image and increases the memory consumption. In this paper, we present a high-accuracy 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 compute the interaction between light and volume.

The main contribution of this paper is to propose and evaluate an integration scheme that considers the trilinear scalar function of a hexahedral cell, implemented on graphics hardware. To the best of our knowledge, no other work has proposed direct volume rendering of unstructured hexahedral meshes considering such trilinear variation of the scalar field.

We also extend our work first presented in [9] by considering an adaptive linear approximation of the scalar field inside the hexahedron cell. Unlike previous proposals, we find our integration interval considering the minimum and maximum values of the scalar function along each ray. As a result, we end up with a linear approximation that better captures the trilinear variation, as compared to the approximation imposed by a fixed tetrahedron subdivision scheme. We have run computational experiments comparing our two proposals in terms of rendering time and image quality.

This paper is organized as follows: Section 2 reviews previous works on volume rendering related to our proposal. Section 3 details both of our proposals. In Sect. 4, we evaluate the achieved results, and Sect. 5 concludes the paper.

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:

$$I(t_b) = I(t_f)e^{-\left(\int_{t_f}^{t_b}\rho(f(t))\,dt\right)} + \int_{t_f}^{t_b} e^{-\left(\int_{t}^{t_b}\rho(f(u))\,du\right)}\kappa\big(f(t)\big)\rho\big(f(t)\big)\,dt \qquad (1)$$

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

Evaluating such integral accurately and efficiently is one of the main challenges 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 nonlinear (the TF in Fig. 2 presents, for example, 1 control point inside the interval). A later work by Röttger et al. [13] proposed to use preintegration for tetrahedral meshes, storing the parameterized result in a texture, accessed by the front scalar value, back scalar value, and ray length. Their proposal works for any transfer function, but any change on the transfer function requires the preintegration to be recomputed. Röttger [12] later proposed to utilize the GPU to accelerate the precomputation of the 3D table. Moreland et al. [10] reparameterized the preintegration result, making it independent of the transfer function, but under the assumption that the transfer function was piecewise linear. The preintegration result was then stored in a 2D texture, accessed via the normalized values of s_f and s_b (the scalar values at the front and back points of the cell). However, the preintegration results were 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 (Eq. (1)).

2.2 Hexahedral meshes

Volume rendering of unstructured hexahedral meshes was explored by Shirley et al. [14] and Max et al. [8]. 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 discussed 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 a more elaborate approach than a simple fixed 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 a hexahedral mesh ray-casting, focusing on the traversal between the elements, but they did not 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 stored a preintegration table in a 3D texture, where each value was accessed by the scalar values at the ray front, middle, and back 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 preintegration table proposed by Rottger et al. [13], accessed via the scalar values at front and back points, and used a constant step. Unlike our proposal, they do not consider the minimum and maximum values of the scalar function to choose adequate integration intervals.

These papers make the assumption that the scalar function is either linear or bilinear to compute an integral that can 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 rebuilt for each TF change. In this paper, we first avoid the use of preintegration 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 in an adaptive approach, considering the minimum and maximum values of the scalar function along the ray (in a cell), thus reducing accuracy, compared to our first scheme, but increasing performance.

3 Hexahedron ray-casting

In this section, we describe our ray-casting proposals to handle hexahedral meshes. Section 3.1 describes our accurate proposal, Sect. 3.2 discusses how we determine our integration intervals, and Sect. 3.3 describes our fast proposal. Section 3.4 presents our data structure, and Sect. 3.5 describes how we traverse the hexahedral mesh. Section 3.6 details how we also render isosurfaces and finally, Sect. 3.7 presents the overall ray-casting algorithm.

3.1 Accurate ray integration

The trilinear scalar function inside a hexahedral cell can be described with the following equation:

$$f(x,y,z) = c_0 + c_1x + c_2y + c_3z + c_4xy + c_5yz + c_6xz + c_7xyz \tag{2}$$

where $c_0, c_1, c_2, c_3, c_4, c_5, c_6, c_7$ are the cell coefficients. 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].

To find the integration equation considering the trilinear function, we consider the position of a point inside the cell. We parameterize the hexahedral scalar function (Eq. (2)) along the ray by the ray length inside the cell, denoted by t:

$$f(t) = s = w_3t^3 + w_2t^2 + w_1t + w_0, t \in [t_b, t_f] \tag{3}$$

where t_f and t_b are the ray lengths in the front and back faces of the cell, and with:

$$\begin{aligned} w_0 &= c_0 + c_1e_x + c_2e_y + c_4e_xe_y + c_3e_z + c_6e_xe_z \\ &\quad + c_5e_ye_z + c_7e_xe_ye_z + c_7d_xd_ye_z \\ w_1 &= c_1d_x + c_2d_y + c_3d_z + c_4d_ye_x + c_6d_ze_x \\ &\quad + c_4d_xe_y + c_5d_ze_y + c_7d_ze_xe_y + c_6d_xe_z \\ &\quad + c_5d_ye_z + c_7d_ye_xe_z + c_7d_xe_ye_z \\ w_2 &= c_4d_xd_y + c_6d_xd_z + c_5d_yd_z + c_7d_yd_ze_x + c_7d_xd_ze_y \\ w_3 &= c_7d_xd_yd_z \end{aligned} \tag{4}$$

where $\mathbf{e}=\{e_x, e_y, e_z\}$ is the eye position, and $\mathbf{d}=\{d_x, d_y, d_z\}$ is the ray direction.

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

$$\kappa(s) = \frac{(\kappa_b - \kappa_f)(s - s_f)}{s_b - s_f} + \kappa_f \tag{5}$$

$$\rho(s) = \frac{(\rho_b - \rho_f)(s - s_f)}{s_b - s_f} + \rho_f \tag{6}$$

Getting back to Eq. (1), we use a Gauss–Legendre quadrature method to integrate the color and opacity along the ray:

$$I(t_b) = I(t_f)e^{-z_{t_f,t_b}} + \int_{t_f}^{t_b} e^{-z_{t,t_b}}\kappa(t)\rho(t)\,dt \tag{7}$$

where $z_{a,b}$ is given by

$$\begin{aligned} z_{a,b} &= \int_a^b \rho(r)\,dr \\ z_{a,b} &= \rho_f(b-a) + \frac{(\rho_b - \rho_f)}{12(s_b - s_f)} \\ &\quad * \left[12(as_f - bs_f) + 12w_0(b-a) + 6w_1\left(b^2 - a^2\right)\right. \\ &\quad \left. + 4w_2\left(b^3 - a^3\right) + 3w_3\left(b^4 - a^4\right)\right] \end{aligned} \tag{8}$$

Calculating the integral from Eq. (7) using quadrature, we have

$$I(t_b) = I(t_f)e^{-z_{t_f,t_b}} + \sum_{i=0}^{3} D * GW_i * e^{-z_{t_g,t_b}} \kappa(t_g)\rho(t_g) \quad (9)$$

where

$$D = t_b - t_f \quad (10)$$

$$t_g = t_f + GP_i * D \quad (11)$$

GP_i and GW_i are the precomputed points and weights for the Gauss–Legendre quadrature.

The Gaussian quadrature integration method gives exact results for polynomials up to degree 5, considering a 3-point quadrature, and good approximations for functions that are approximated by such polynomials [18]. Since we split our integration into several intervals, as we shall explain in Sect. 3.2, we make sure that our ray integral is properly evaluated.

3.2 Integration intervals

The scalar field variation along a ray in a hexahedral cell, $f(t)$, can be illustratively represented by the function in Fig. 1, according to Eq. (3). As a cubic polynomial function, $f(t)$ has two extrema, which can be calculated from its derivative $f'(t)$, a quadratic polynomial. Considering $t_{\min}$ and $t_{\max}$, the values of t where the scalar function is minimum or maximum, respectively, we calculate t_{near} and t_{far}, such that $t_{\text{near}} = \min(t_{\min}, t_{\max})$ and $t_{\text{far}} = \max(t_{\min}, t_{\max})$. If t_f denotes the point the ray enters the cell and t_b the point it exits the cell, the function $f(t)$ is monotonic in the intervals $[t_f, t_{\text{near}}]$, $[t_{\text{near}}, t_{\text{far}}]$, and $[t_{\text{far}}, t_b]$.

Since we guarantee that we only have monotonic intervals, we can use a 2D texture, as first proposed by Röttger et al. [13], to check the existence of control points in the intervals. 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} along the ray, if one exists, such that $s_0 < s_{\text{cp}} < s_1$ or $s_1 < s_{\text{cp}} < s_0$.

With s_{cp}, we can find the value of t_{cp}, which is the ray length from the eye to the corresponding isosurface that crosses the hexahedral, by solving Eq. (12) with the Newton–Raphson method.

$$w_3 t_{\text{cp}}^3 + w_2 t_{\text{cp}}^2 + w_1 t_{\text{cp}} + w_0 = s_{\text{cp}} \quad (12)$$

One of the main problems with the Newton–Raphson root finding method is its dependency of a good initial guess. However, in our case, the mid-point of the interval limits represents a good first choice.

To exemplify this procedure, let us consider the scalar variation inside a hexahedron given by the function depicted in Fig. 1, with $t_f = \alpha_0$ and $t_b = \alpha_3$, and the transfer function illustrated in Fig. 2, assuming that $0 < f(\alpha_0) < f(\alpha_2) < 0.5 < f(\alpha_1) < f(\alpha_3) < 1$. We have $t_{\max} = \alpha_1$ and $t_{\min} = \alpha_2$; we then calculate $t_{\text{near}} = \min(t_{\min}, t_{\max}) = \alpha_1$ and $t_{\text{far}} = \max(t_{\min}, t_{\max}) = \alpha_2$. In this case, there is one control point in the interval $[t_f, t_{\text{near}}]$, one control point in $[t_{\text{near}}, t_{\text{far}}]$, and one control point in $[t_{\text{far}}, t_b]$. These give us the following integration intervals: $[\alpha_0, t_{\text{cp1}}]$, $[t_{\text{cp1}}, \alpha_1]$, $[\alpha_1, t_{\text{cp2}}]$, $[t_{\text{cp2}}, \alpha_2]$, $[\alpha_2, t_{\text{cp3}}]$, and $[t_{\text{cp3}}, \alpha_3]$.

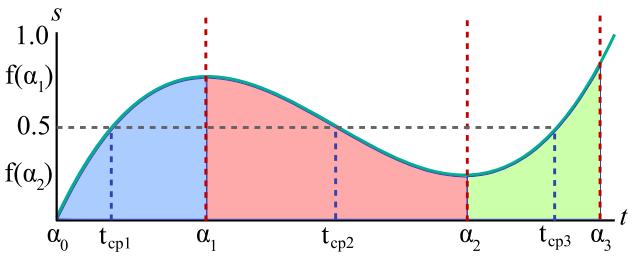


Fig. 1 Minimum and maximum values of a trilinear function along the ray inside a hexahedron

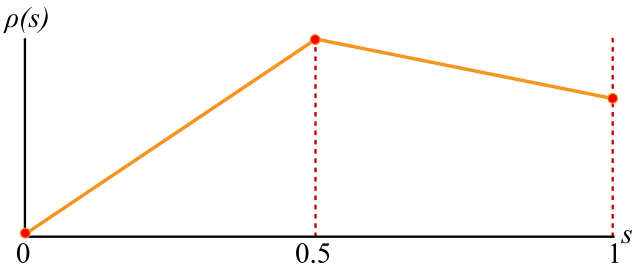


Fig. 2 Transfer function represented by a piecewise linear variation

3.3 Fast ray integration

In order to achieve better performance, without significantly compromising quality, we also propose an alternative approach. Like in our previous approach, we also consider the trilinear variation of the scalar field. The difference, however, lies in how we calculate the ray integral in Eq. (1). We approximate the cubic variation along the ray with an adaptive piecewise linear function, whose intervals are delimited by the minimum and maximum values of the hexahedron scalar function along the ray, as shown in Fig. 3.

To compute the ray integral, we then use the 2D preintegrated table, first proposed by Moreland et al. [10], to render unstructured tetrahedral meshes:

$$I(t_b) = I(t_f)\zeta_{(D,\tau(t))} + \kappa(t_b)(\psi_{D,\tau(t)} - \zeta_{D,\tau(t)}) + \kappa(t_f)(1 - \psi_{D,\tau(t)}) \quad (13)$$

with $D = t_b - t_f$ and:

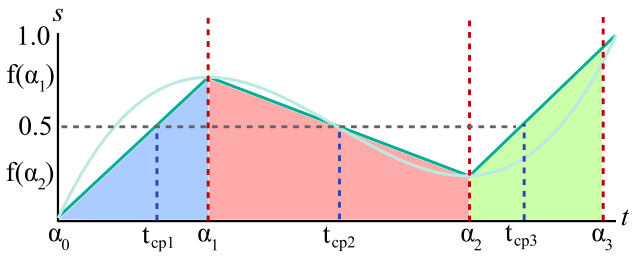


Fig. 3 Piecewise linear functions, considering the minimum and maximum values of a trilinear scalar function

$$\zeta_{D,\tau(t)} = e^{-\int_0^D \tau(t)\,dt} \tag{14}$$

$$\psi_{D,\tau(t)} = \frac{1}{D}\int_0^D e^{-\int_u^D \tau(t)\,dt}\,du \tag{15}$$

Equations (14) and (15) are reparameterized substituting $\tau(t)D$ by $\frac{\gamma}{1-\gamma}$ so that it can fit in a 2D texture in the $[0, 1)$ domain [10]. The final equations are

$$\psi_{\gamma_b,\gamma_f} = \int_0^1 e^{-\int_u^1 (\frac{\gamma_b}{1-\gamma_b}(1-t)+\frac{\gamma_f}{1-\gamma_f}t)\,dt}\,du \tag{16}$$

$$\zeta_{D,\tau_b,\tau_f} = e^{-\frac{D}{2}(\tau_b+\tau_f)} \tag{17}$$

It may sound counterintuitive to go back to a piecewise linear scalar function, since the basis of this paper is to propose something better than the tetrahedron subdivision scheme, which is also a piecewise linear scalar function. But this is a different, adaptive approach, as we first consider the trilinear scalar function to find the minimum and maximum values, resulting in a better linear approximation. With the standard tetrahedralization scheme, the subdivision is non-adaptive, ignoring the minimum and maximum values.

Our fast proposal finds t_{cp} considering the linear variation of the scalar function, given by Eq. (18).

$$t_{cp} = t_0 + (t_1 - t_0)\left(\frac{|s_{cp} - s_0|}{s_1 - s_0}\right) \tag{18}$$

where t_0 and t_1 are the ray length at the extreme points of the integration interval; s_0 and s_1 are the corresponding scalar values.

3.4 Data structure

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

We need to store eight coefficients per cell (Eq. (2)). For adjacency information, we need six more 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 between a ray and a hexahedral cell, we use a simple ray-plane intersection test. We split each quadrilateral face of a cell into two triangles and store the corresponding plane equations, totaling 12 planes per cell (48 coefficient values).

Table 1 Data structure for one hexahedral cell

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

We then store a total of 62 values associated to each cell. Considering 4 bytes per value, we need 248 bytes per cell. Even optimized data structures, such as the one described by Weiler et al. [16], require 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 hexahedral cells is its small memory consumption when compared to the tetrahedron subdivision scheme.

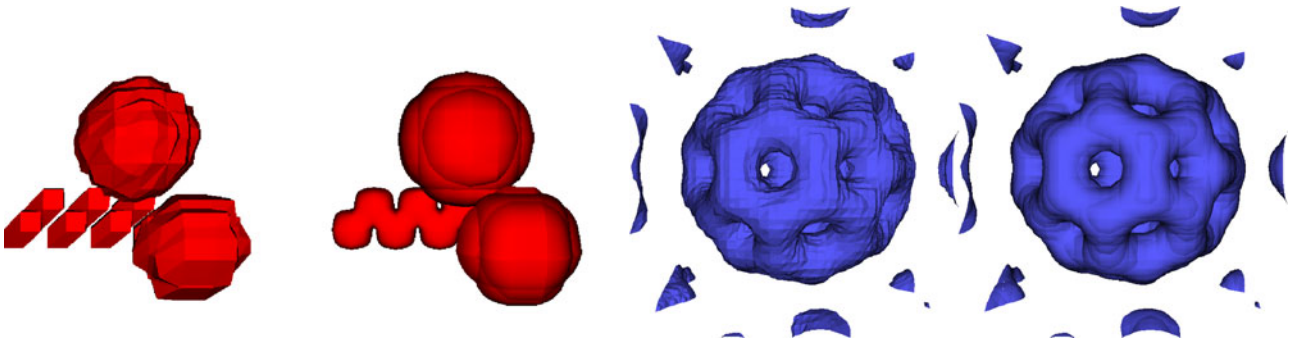
3.5 Ray traversal

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 the external volume boundary to a texture, storing the corresponding cell ID for each pixel on the screen. The second step will fetch the texture and initiate the mesh traversal, starting at the stored cell. The algorithm proceeds by traversing the mesh until the ray exits the volume. In models with holes or gaps, the ray can reenter the volume. At each peel, the ray reenters at the volume boundary, and the color and opacity from previous peels are accumulated. The ray-casting algorithm is finished when the last external boundary of the model is reached.

As mentioned, to compute the intersection between the ray and a cell, we subdivide each cell face in two triangles and compute the distance between the eye and each triangle plane. This intersection test is an approximation. In order to improve accuracy, we have implemented both a ray-bilinear patch intersection test and a ray intersection test in Plücker space [7]. However, neither of these solutions performed well in our implementation, due to their necessity to perform more math operations than a ray-triangle intersection test. We then decided to maintain a simpler ray-plane intersection algorithm.

3.6 Isosurfaces

We can extend our volume rendering algorithm to also handle isosurface rendering. In fact, this is very simple, because we already compute all control points along the ray. We only need to consider the isosurface values as additional control



(a) Tetrahedron subdivision (b) Accurate proposal (c) Tetrahedron subdivision (d) Accurate proposal

Fig. 4 Isosurface rendering of the Atom9 and Bucky datasets

points. The surface normal is given by the gradient of the scalar field:

$$\mathbf{n} = \nabla f(x, y, z) = \left\langle \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right\rangle \tag{19}$$

where $\{x, y, z\}$ is the intersection between the ray and the iso-surface, and is given by $\mathbf{p} = \{x, y, z\} = \mathbf{e} + t_{\text{cp}}\mathbf{d}$.

Figures 4a and 4b present an isosurface rendering of the Atom9 dataset from [2]. If compared to tetrahedron subdivision scheme, our proposal depicts the isosurface shape with improved accuracy. Figures 4c and 4d present an isosurface rendering of the Bucky dataset, and it is possible to notice the smoother isosurface presented by our proposal. Although not generating the exact isosurface, our proposal does produce an isosurface shape with improved accuracy, compared to a tetrahedron subdivision scheme.

3.7 Ray-casting algorithm

The algorithm in Table 2 summarizes our approach. We traverse the mesh accumulating each cell contribution to the pixel color. We first calculate the minimum and maximum values of the scalar field along the ray as it traverses through each cell, clamping values outside the cell boundary; t_{near} and t_{far} represent the distance to the eye of the closest and farthest min/max values. We then iterate through t_f, t_{near}, t_{far}, t_b, fetching the texture described in Sect. 3.2 to find if there are control points in each interval. If any control point exists, the algorithm integrates from the current position t_i to the first cp (with length t_{cp}), and then to the others, updating the value of t_i until the back point is reached.

Table 2 Ray-cast algorithm

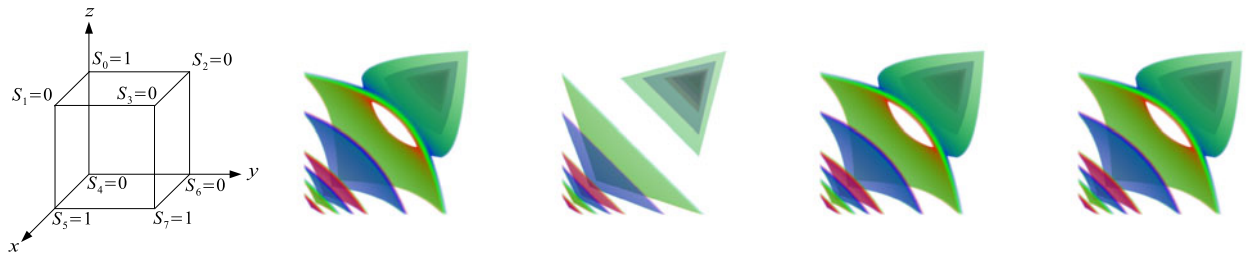
```
color ← (0, 0, 0, 0)
cell.id ← VolumeBoundary()
while color.a < 1 and ray inside volume do
    t_b, cell.nid ← Intersect(t, cell)
    t_min, t_max = Solve(cell.f'(t) = 0)
    t_min = clamp(t_min, t_f, t_b)
    t_max = clamp(t_max, t_f, t_b)
    t_near = min(t_min, t_max)
    t_far = max(t_min, t_max)
    t_i = [t_f, t_near, t_far, t_b]
    s_i = [s_f, s_near, s_far, s_b]
    i = 0
    while i < 3 do
        {Find control points}
        s_cp = FindCp(s_i, s_{i+1})
        if s_i < s_cp < s_{i+1} or s_i > s_cp > s_{i+1} then
            t_cp = Newton(cell.f(t) = s_cp, (t_i + t_{i+1})/2)
        else
            t_cp = t_{i+1}
        end if
        color ← Integrate(t_i, s_i, t_cp, s_cp)
        t_i = t_cp
        if t_i >= t_{i+1} then
            i++
        end if
    end while
    cell.id = cell.nid
    t = t_b
end while
```

4 Results

For the evaluation of our proposals, 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.

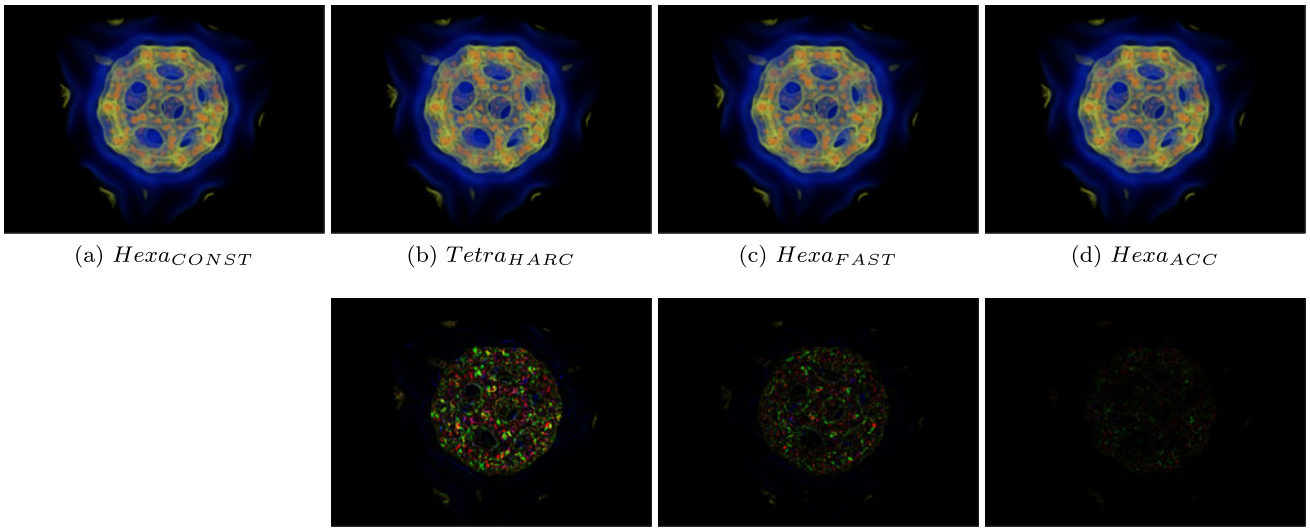
We tested the following algorithms for comparison:

1. **Hexa**$_{CONST}$: Ray-casting for hexahedral meshes that uses a fixed number of 100 steps within each cell to accurately compute the illumination assuming a constant scalar field at each step.
2. **Tetra**$_{HARC}$: Ray-casting using a preintegrated table [3], with each hexahedral cell subdivided into six tetrahedra.



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

Fig. 5 Rendering of a synthetic model composed of one hexahedron



(a) $Hexa_{CONST}$ (b) $Tetra_{HARC}$ (c) $Hexa_{FAST}$ (d) $Hexa_{ACC}$

(e) Difference of $Tetra_{HARC}$ (f) Difference of $Hexa_{FAST}$ (g) Difference of $Hexa_{ACC}$

Fig. 6 Images of the Bucky model

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

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 470 GTX 1 GB RAM. The screen size was set to 800 × 800 pixels in all experiments.

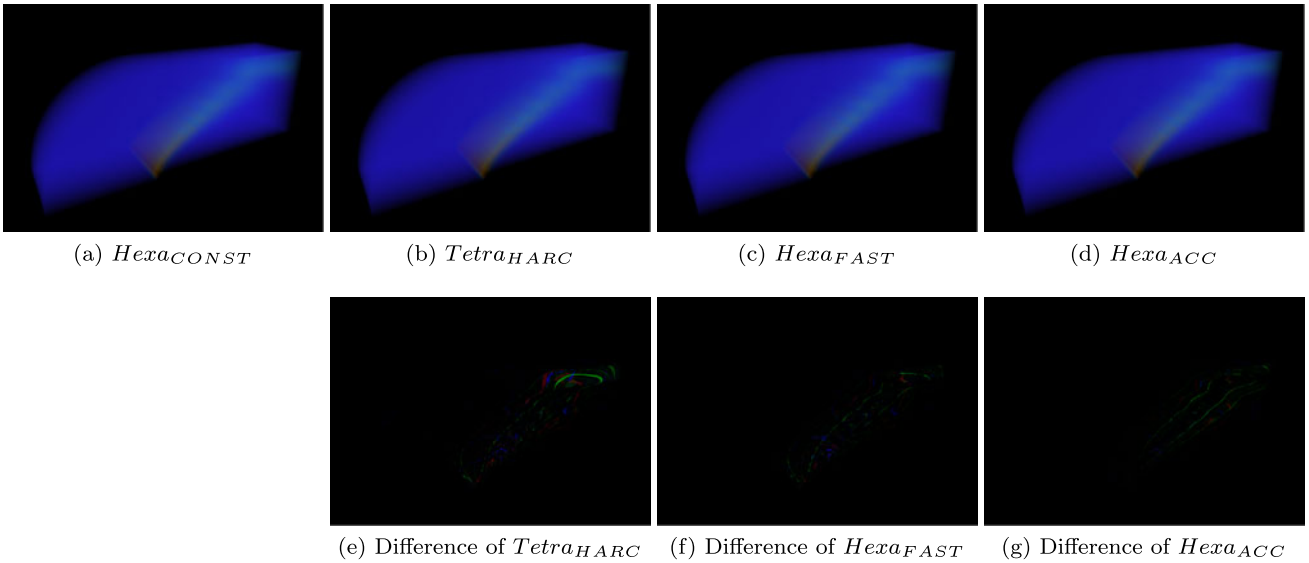
4.1 Rendering quality

We evaluate our ray-casting algorithms regarding rendering quality. We first devised an experiment using synthetic volume data composed of only one hexahedral cell. Figure 5 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 scalar field variation is rare in actual datasets, this synthetic test aims to show how the tetrahedron subdivision of a 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 6 shows and compares the results achieved by the four algorithms for rendering the Bucky model. Figures 6e, 6f, and 6g shows the difference between the images achieved with our two proposals and the subdivision scheme when compared to our quality reference. It is possible to notice a clear difference among the rendering images, with our proposals being more similar to our quality reference. The **Hexa**$_{FAST}$ proposal presents a subtle difference, which is expected because of the approximations of the ray integral.

Figure 7 shows the result of a similar experiment, but now considering the Bluntfin model. As can be noted, the results are equivalent: our algorithms produce images with a better quality.

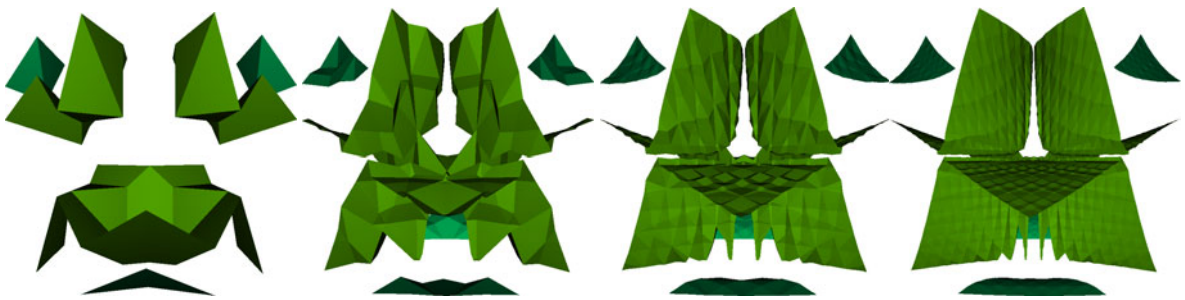
Figures 8, 9, 10 and 11 show the isosurface described by the implicit function of a Steiner surface, with a regular grid of size varying from 3^3 to 20^3 cells. Our proposal presents better results in all grid sizes.



(a) $Hexa_{CONST}$ (b) $Tetra_{HARC}$ (c) $Hexa_{FAST}$ (d) $Hexa_{ACC}$

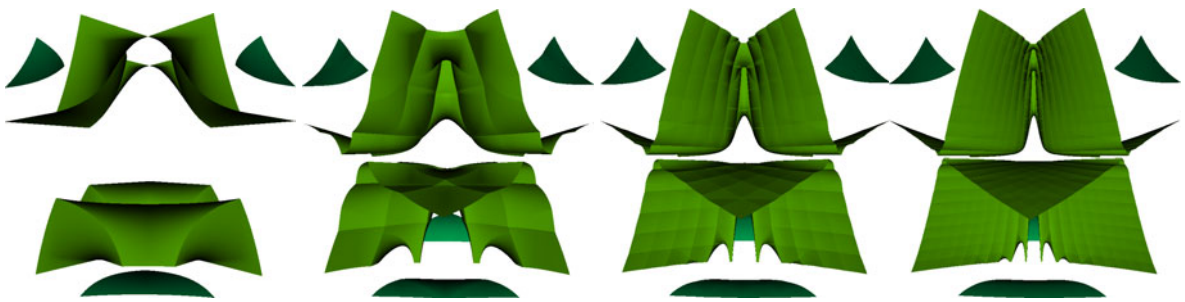
(e) Difference of $Tetra_{HARC}$ (f) Difference of $Hexa_{FAST}$ (g) Difference of $Hexa_{ACC}$

Fig. 7 Images of the Bluntfin model



(a) $3 \times 3 \times 3$ (b) $6 \times 6 \times 6$ (c) $12 \times 12 \times 15$ (d) $20 \times 20 \times 20$

Fig. 8 Isosurface of a Steiner surface (**Tetra**$_{HARC}$) in a grid of varying size



(a) $3 \times 3 \times 3$ (b) $6 \times 6 \times 6$ (c) $12 \times 12 \times 12$ (d) $20 \times 20 \times 20$

Fig. 9 Isosurface of a Steiner surface (**Hexa**$_{ACC}$, **Hexa**$_{FAST}$) in a grid of varying size

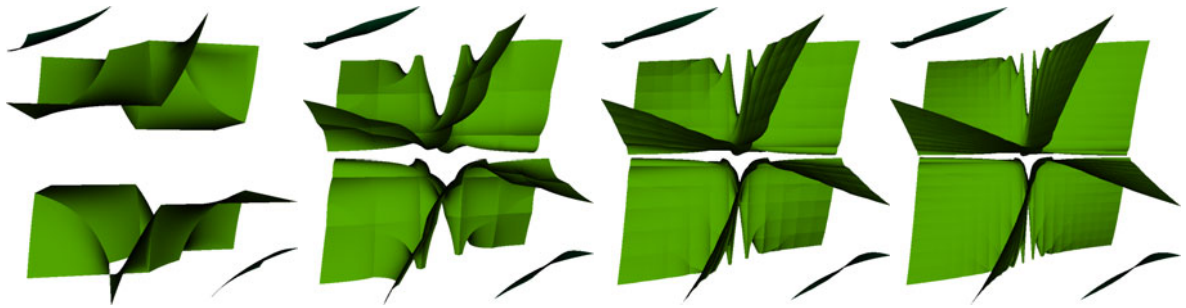
4.2 Memory and performance

Table 3 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 we changed the camera to 64 different positions for each run, averaging the measured time. Since the algorithms are heavily dependent on the transfer function, we tested them with a fixed TF with 23 control points.

Our proposals reduce memory consumption by a factor of 2.2, compared to a subdivision scheme of 6 tetrahedra per hexahedral cell. The performance of our **Hexa**$_{ACC}$ proposal is about 15 % slower than the subdivision scheme. To

(a) $3 \times 3 \times 3$ (b) $6 \times 6 \times 6$ (c) $12 \times 12 \times 12$ (d) $20 \times 20 \times 20$

Fig. 10 Isosurface of a Steiner surface (**Tetra**$_{HARC}$) in a grid of varying size



(a) $3 \times 3 \times 3$ (b) $6 \times 6 \times 6$ (c) $12 \times 12 \times 12$ (d) $20 \times 20 \times 20$

Fig. 11 Isosurface of a Steiner surface (**Hexa**$_{ACC}$, **Hexa**$_{FAST}$) in a grid of varying size

achieve accurate results, we need to perform part of the computation in double precision (specifically Eqs. (4) and (10)); that is the main reason for losing performance. If we had used single precision, our **Hexa**$_{ACC}$ algorithm would be 15 to 20 % faster than the subdivision scheme, but this would bring numerical inaccuracy. Our **Hexa**$_{FAST}$ algorithm, however, presents a performance around 12 % better than the **Tetra**$_{HARC}$ algorithm. Because of the approximation of the scalar function by a piecewise linear function, the algorithm does not have a high computation cost, compared to our accurate proposal.

Table 3 Rendering times and memory footprint of the subdivision scheme and our proposals

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

5 Conclusion

We have presented an accurate hexahedral volume rendering technique suitable for unstructured meshes. Our proposal integrates the trilinear scalar function using a quadrature approach on the GPU. Although our performance was worse than the performance of a tetrahedron subdivision scheme, our proposal produces images with better quality. By using a proper integration of the trilinear function inside a hexahedron, our proposal ensures that no isosurface value is missed during integration.

We also presented a fast algorithm that handles hexahedral meshes and, although it is not as accurate as our quadrature proposal, it achieves good approximations and it is also faster than a tetrahedral volume rendering.

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 proposals present a smaller memory footprint than regular tetrahedron subdivision schemes.

In the future, we plan to revisit 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.

Acknowledgements We thank CAPES (Brazilian National Research and Development Council) and CNPq (Brazilian National Council for Scientific and Technological Development) for the financial support to conduct this research. This work was done at the Tecgraf laboratory at PUC-Rio, which is mainly funded by the Brazilian oil company, Petrobras.

References

1. Bernadon, F.F., Pagot, C.A., Comba, J.L.D., Silva, C.T.: GPU-based tiled ray casting using depth peeling. J. Graph. GPU Game Tools **11**(4), 1–16 (2006)
2. Carr, H., Moller, T., Snoeyink, J.: Artifacts caused by simplicial subdivision. IEEE Trans. Vis. Comput. Graph. **12**, 231–242 (2006). doi:10.1109/TVCG.2006.22
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. IEEE Computer Society, Washington (2005). doi:10.1109/SIBGRAPI.2005.29. http://portal.acm.org/citation.cfm?id=1114697.1115365
4. Garrity, M.P.: Raytracing irregular volume data. In: Proceedings of the 1990 Workshop on Volume visualization, VVS'90, pp. 35–40. ACM, New York (1990). doi:10.1145/99307.99316
5. Hajjar, J.E., Marchesin, S., Dischler, J., Mongenet, C.: Second order pre-integrated volume rendering. In: IEEE Pacific Visualization Symposium (2008)
6. Marchesin, S., de Verdiere, G.: High-quality, semi-analytical volume rendering for amr data. IEEE Trans. Vis. Comput. Graph. **15**(6), 1611–1618 (2009). doi:10.1109/TVCG.2009.149
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 (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, pp. 157–168 (2003)
9. Miranda, F.M., Celes, W.: Accurate volume rendering of unstructured hexahedral meshes. In: Lewiner, T., Torres, R. (eds.) Sibgrapi 2011 (24th Conference on Graphics, Patterns and Images), pp. 93–100. IEEE, Maceió (2011). doi:10.1109/SIBGRAPI.2011.3. http://www.im.ufal.br/evento/sibgrapi2011/
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, pp. 9–16. IEEE Computer Society, Washington (2004). doi:10.1109/VV.2004.2
11. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical Recipes in C: The Art of Scientific Computing, 2nd edn. Cambridge University Press, New York (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, pp. 23–28. IEEE Press, Piscataway (2002) http://portal.acm.org/citation.cfm?id=584110.584114
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, pp. 109–116. IEEE Computer Society Press, Los Alamitos (2000). http://portal.acm.org/citation.cfm?id=375213.375226
14. Shirley, P., Tuchman, A.: A polygonal approximation to direct scalar volume rendering. In: Proceedings of the 1990 Workshop on Volume Visualization, VVS'90, pp. 63–70. ACM, New York (1990). doi:10.1145/99307.99322
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, p. 44. IEEE Computer Society, Washington (2003). doi:10.1109/VISUAL.2003.1250390
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, pp. 71–78. IEEE Computer Society, Washington (2004). doi:10.1109/VV.2004.13
17. Williams, P.L., Max, N.: A volume density optical model. In: Proceedings of the 1992 Workshop on Volume Visualization, VVS'92, pp. 61–68. ACM, New York (1992). doi:10.1145/147130.147151
18. Williams, P.L., Max, N.L., Stein, C.M.: A high accuracy volume renderer for unstructured data. IEEE Trans. Vis. Comput. Graph. **4**, 37–54 (1998)

Fabio Markus Miranda received a M.Sc. in computer science from PUC-Rio in 2011 and a B.Sc. in computer science from Universidade Federal de Minas Gerais (UFMG) in 2009. He is currently a researcher at Tecgraf working with scientific visualization.

Waldemar Celes is a researcher professor in the Computer Science Department at PUC-Rio, Brazil, and a former postdoctoral associate at the Program of Computer Graphics, Cornell University. His current research interests in computer graphics include real-time rendering, scientific visualization, physical simulation, and distributed graphics applications. He is also one of the authors of the Lua programming language.