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:

$$\begin{aligned} f(x,y,z) &= c_0 + c_1 x + c_2 y + c_3 z \\ &+ c_4 xy + c_5 yz + c_6 xz + c_7 xyz \end{aligned} \quad (3\text{-}1)$$

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:

$$\begin{pmatrix} 1 & x_0 & y_0 & \cdots & x_0 y_0 z_0 \\ 1 & x_1 & y_1 & \cdots & x_1 y_1 z_1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_7 & y_7 & \cdots & x_7 y_7 z_7 \end{pmatrix} \begin{pmatrix} c_0 \\ \vdots \\ c7 \end{pmatrix} = \begin{pmatrix} s_0 \\ \vdots \\ s_7 \end{pmatrix}$$

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:

$$\mathbf{p} = \mathbf{e} + t\vec{d} \quad (3\text{-}2)$$

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:

$$f(t) = w_3 t^3 + w_2 t^2 + w_1 t + w_0, t \in [t_{back}, t_{front}] \quad (3\text{-}3)$$

PUC-Rio - Certificação Digital Nº 0921332/CA

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

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

PUC-Rio - Certificação Digital Nº 0921332/CA

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

PUC-Rio - Certificação Digital Nº 0921332/CA

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: $[0.73, 1]$.

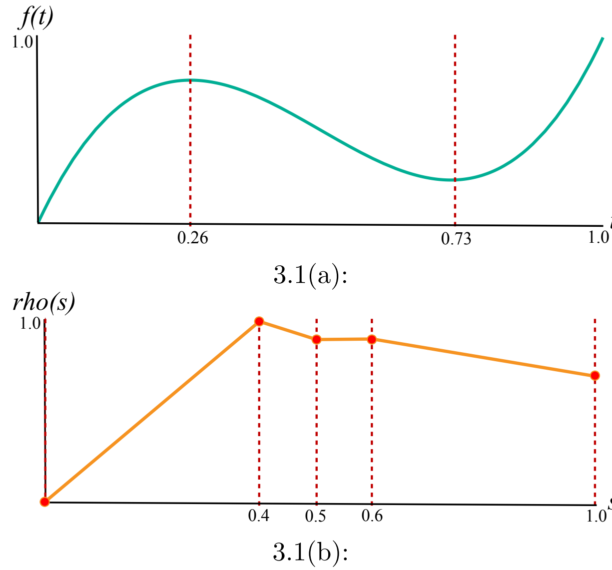


Figure 3.1: Example of scalar field variation inside a hexahedral cell: (a) Maximum and minimum values of a trilinear function along the ray inside an hexahedron; (b) Transfer function represented by a piecewise linear variation.

PUC-Rio - Certificação Digital Nº 0921332/CA

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:

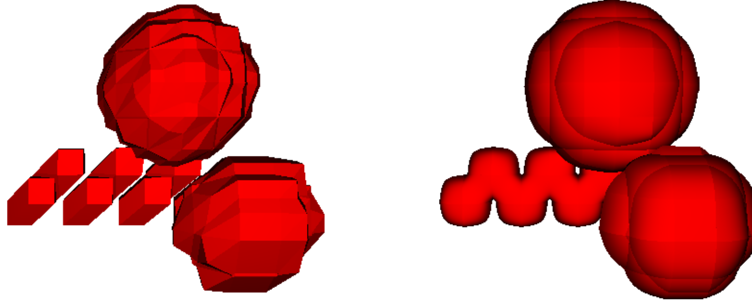
$$
\begin{aligned}
\vec{n} &= \nabla f(x,y,z) = < \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} > \\
\vec{n} &= \begin{bmatrix} c_1 + c_4 y + c_6 z + c_7 yz \\ c_2 + c_4 x + c_5 z + c_7 xz \\ c_3 + c_5 x + c_6 y + c_7 xy \end{bmatrix}
\end{aligned}
\tag{3-10}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

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

PUC-Rio - Certificação Digital Nº 0921332/CA

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.

PUC-Rio - Certificação Digital Nº 0921332/CA

Table 3.2: Ray-casting Algorithm

```
 1: color ← (0,0,0,0)
 2: cell.id ← VolumeBoundary()
 3: while color.a < 1 and ray inside volume do
 4:    t_back, cell.nid ← 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, (t_i + t_{i+1})/2)
17:       else
18:          t_cp = t_{i+1}
19:       end if
20:       color ← 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
```

PUC-Rio - Certificação Digital Nº 0921332/CA