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.

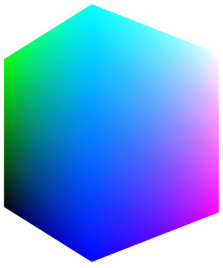
Table C.2: Ray-casting Algorithm

```
color ← (0, 0, 0, 0)
cell.id ← VolumeBoundary()
while color.a < 1 and ray inside volume do
    t_back, cell.nid ← 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 = (s_cp − s_front) / (s_back − s_front)
    else
        t_cp = t_back
    end if
    color ← Integrate(t_front, s_front, t_cp, s_cp)
    cell.id = cell.nid
    t_front = t_back
end while
```

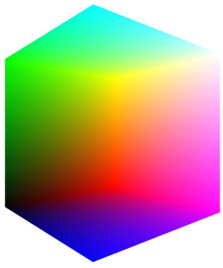
D
Regular data

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

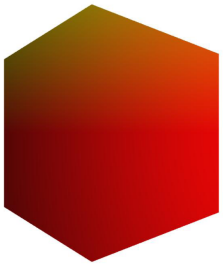
The regular data ray-casting follows the simple steps: Steps 1 and 2 render an unitary cube front and back faces to a texture using a FBO (frame buffer object). These two textures are then passed to a shader in Step 3 that computes the ray direction and ray length, saving the result to another texture. 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.



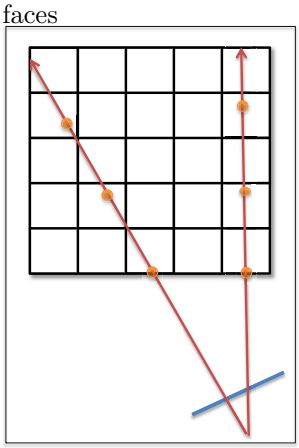
D.1(a): Step 1: cube front faces



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



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

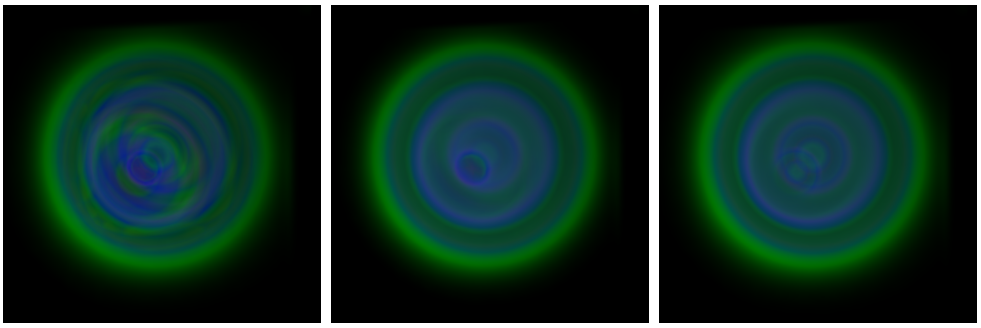


D.1(d): Step 3: ray traversal

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**