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

PUC-Rio - Certificação Digital Nº 0921332/CA

PUC-Rio - Certificação Digital Nº 0921332/CA

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.

PUC-Rio - Certificação Digital Nº 0921332/CA