IMPLEMENTAÇÃO ROBUSTA E EFICIENTE EM USO DE MEMÓRIA
DO ALGORITMO DE RAYCAST

Aline Aparecida de Pina

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

_____
Prof. Ricardo Cordeiro de Farias, Ph.D.

_____
Prof. Claudio Esperança, Ph.D.

_____
Prof. Luiz Fernando Campos Ramos Martha, Ph.D.

RIO DE JANEIRO, RJ - BRASIL
MARÇO DE 2005

PINA, ALINE APARECIDA DE

Implementação Robusta e Eficiente em Uso de Memória do Algoritmo de Raycast [Rio de Janeiro] 2005

XI, 79 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2005)

Tese - Universidade Federal do Rio de Janeiro, COPPE

1. Visualização Volumétrica

2. Ray-Casting

I. COPPE/UFRJ II. Título (série)

À minha Família.

A vitória e o sucesso de cada um de nós sempre representará a vitória e o sucesso para todos nós. Amo vocês.

AGRADECIMENTOS

Agradeço a todos aqueles que direta ou indiretamente tenham colaborado de alguma forma para a realização deste trabalho, em especial ao professor Ricardo Cordeiro de Farias, meu orientador.

Agradeço ao professor Claudio Esperança que, como meu professor em diversas disciplinas no curso de Mestrado, acreditou em mim e fez com que eu me dedicasse com afinco a meus estudos, ajudando-me a realizar minha pesquisa para o desenvolvimento desta tese.

Agradeço ao professor Cláudio T. Silva, por tão prontamente ter me ajudado quando precisei.

Agradeço ao meu colega André de Almeida Maximo, por ter me auxiliado nos testes das minhas implementações e sempre ter se mostrado à minha disposição quando precisei.

Agradeço ao apoio financeiro fornecido pelo CNPq durante todo o decorrer do trabalho. Enquanto existirem pessoas interessadas em financiar pesquisas para o avanço da tecnologia e conseqüente melhoria da sociedade, o homem continuará evoluindo.

Agradeço à minha Família pelo apoio incondicional demonstrado em todos os momentos. É muito bom saber que existe alguém sempre torcendo e vibrando com nossas conquistas, rezando para que tudo transcorra sem problemas. Essa força foi e sempre será essencial para o término deste e de outros futuros trabalhos.

Acima de tudo e de todos, agradeço a DEUS pela chance de poder desenvolver este trabalho, completando mais um ciclo de minha vida com sucesso, tornando real mais um de meus sonhos. Muito obrigada.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

IMPLEMENTAÇÃO ROBUSTA E EFICIENTE EM USO DE MEMÓRIA DO ALGORITMO DE RAYCAST

Aline Aparecida de Pina

Março/2005

Orientador: Ricardo Cordeiro de Farias

Programa: Engenharia de Sistemas e Computação

Este trabalho apresenta duas implementações do algoritmo de Raycast para conjuntos de dados na forma de grades irregulares. Na primeira implementação, que chamamos de ME-Ray (Memory Efficient Ray-Casting), visamos desenvolver um método que fosse competitivo com a implementação do algoritmo de ray-casting desenvolvida por BUNYK *et al*. (1999) tanto em uso de memória quanto em tempo de renderização. Na segunda, que chamamos de EME-Ray (Enhanced Memory Efficient Ray-Casting), demos maior importância à economia de memória do que ao tempo gasto para o processamento. Através de dados comparativos, pôde-se verificar que os métodos desenvolvidos utilizam menos memória que o método de BUNYK *et al*. e geram imagens de boa qualidade.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

MEMORY EFFICIENT AND ROBUST IMPLEMENTATION
OF THE RAYCAST ALGORITHM

Aline Aparecida de Pina

March/2005

Advisor: Ricardo Cordeiro de Farias

Department: Systems Engineering and Computer Science

This work presents two implementations of the Raycast algorithm for irregular grids datasets. In the first implementation, which we call ME-Ray (Memory Efficient Ray-Casting), we intend to develop a method to compete with BUNYK *et al.*'s ray-casting algorithm's implementation (1999) in terms of both memory and render time. In the second one, which we call EME-Ray (Enhanced Memory Efficient Ray-Casting), we care more for spending less memory than spending less time. By means of comparative data, we could verify that the developed algorithms spend less memory than BUNYK *et al.*'s method and generate images of good quality.

SUMÁRIO

ÍNDICE DE TABELAS

ÍNDICE DE FIGURAS

1. INTRODUÇÃO

Os métodos de renderização volumétrica são utilizados para visualizar campos escalares ou vetoriais modelando-se o volume como se fosse composto por células de material semitransparente capaz de emitir, transmitir e absorver luz, permitindo que seja visualizado o interior do volume.

Ray-casting é um dos algoritmos de renderização volumétrica mais utilizados. Em (BUNYK *et al*., 1999) foi apresentado uma técnica de ray-casting simples e eficiente em tempo, mas muito custosa em termos de memória, por utilizar muitas estruturas de dados para guardar informações de adjacência com o intuito de acelerar o cálculo do caminho dos raios através do volume.

Neste trabalho apresentamos duas implementações do algoritmo de Raycast para conjuntos de dados representados na forma de grades irregulares. Estas implementações são versões otimizadas em uso de memória, fazendo uso apenas de uma estrutura de dados auxiliar para representar as adjacências.

Na primeira implementação, que chamamos de ME-Ray (Memory Efficient Ray-Casting), visamos desenvolver um método que fosse competitivo com a implementação feita por BUNYK *et al*. tanto em uso de memória quanto em tempo de execução. Na segunda, que chamamos de EME-Ray (Enhanced Memory Efficient Ray-Casting), demos maior importância à economia de memória do que ao tempo gasto para o processamento.

Mesmo que não consideremos o uso de memória e o tempo de execução das implementações, nossos métodos são superiores ao desenvolvido por BUNYK *et al*.

porque, além de nossos códigos poderem processar modelos com células tetraedrais e/ou hexaedrais, também possuem um tratamento completo para casos degenerados.

A Tese está organizada como segue. No próximo capítulo, revisaremos os conceitos fundamentais de renderização volumétrica e discutiremos alguns dos algoritmos de renderização volumétrica mais conhecidos. No Capítulo 3, apresentaremos as características principais de nossos métodos ME-Ray e EME-Ray. No capítulo 4, mostraremos a análise de performance de nossas implementações em alguns conjuntos de dados. Finalmente, são apresentadas as conclusões no Capítulo 5.

2. TRABALHOS RELACIONADOS

Neste capítulo revisaremos conceitos fundamentais de renderização volumétrica. Primeiramente, faremos uma breve explicação sobre representação de dados volumétricos. Em seguida, apresentaremos as etapas do *Pipeline* de Renderização Volumétrica. Por fim, discutiremos alguns dos algoritmos de renderização volumétrica mais conhecidos.

2.1. Representação de Dados

Renderização volumétrica é um campo muito pesquisado da área de visualização. O objetivo da renderização volumétrica é gerar imagens (bidimensionais) a partir de conjuntos de dados volumétricos (tridimensionais).

Um conjunto de dados volumétrico consiste de informações sobre localizações no espaço. A informação pode representar um campo escalar (como densidade), ou um campo vetorial (como velocidade), ou mesmo uma combinação desses dois, tais como, energia, densidade, e momento em uma simulação computacional de dinâmica dos fluidos.

Os conjuntos de dados volumétricos são freqüentemente representados na forma de grades retilíneas, como uma grade 3D de elementos volumétricos chamados *voxels* (*volume element*, uma analogia ao caso 2D, *pixel*). Cada *voxel* é uma unidade do volume e possui associada a ele algumas propriedades do objeto ou fenômeno em estudo. Se todos os *voxels* são cubos idênticos o conjunto de dados é dito ser regular (Figura 1.a).

Uma aplicação importante que faz uso de dados na forma de grades retilíneas regulares são as imagens médicas.

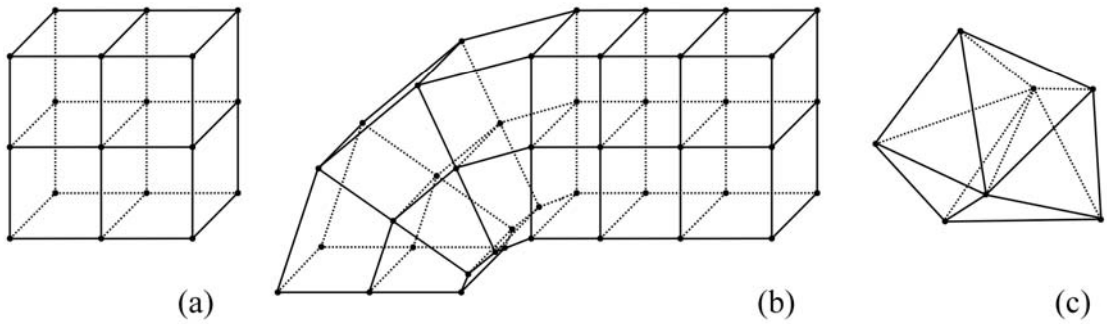


Figura 1. Tipos de representação de dados volumétricos.

(a) Grade retilínea regular; (b) Grade curvilínea; (c) Grade irregular.

Uma variação das grades regulares são as grades curvilíneas (Figura 1.b), que podem ser vistas como o resultado de um "arqueamento" da grade regular feito com o propósito de criar uma amostragem mais densa dos pontos ao redor de uma região particular do espaço. Como esse "arqueamento" é causado por uma transformação não-linear, os lados de cada célula não precisam necessariamente ser retos. Uma vez que as grades curvilíneas podem seguir uma superfície curva, elas são usadas em aplicações de dinâmica dos fluidos computacional para ajustar a forma da grade à superfície de um objeto qualquer.

As grades irregulares (Figura 1.c) consistem de uma lista de pontos com relações de vizinhança que formam células poliedrais com formas arbitrárias, sem relação particular com as grades regulares.

Técnicas de renderização volumétrica tais como *Splatting*, *Shear-Warp* e *Ray-Casting* podem ser aplicadas à dados na forma de grades regulares, mas apenas a última pode ser aplicada a conjuntos de dados na forma de grades irregulares.

No presente trabalho, nos preocupamos em apresentar soluções eficientes para visualização de conjuntos de dados volumétricos dados na forma de grades irregulares.

Primeiramente, precisamos definir alguns termos e conceitos importantes. Suponha que seja dado um conjunto V de n pontos no espaço 3D ($\Re^3$). Se não são dadas conexões entre os pontos, tal conjunto é dito representar uma *nuvem de pontos*. Em grades irregulares, são dadas informações sobre as conexões entre os pontos que servem para organizá-los, em uma decomposição poliedral de $\Re^3$, em vértices, arestas, faces e células:

- Vértices são os pontos de entrada do conjunto V (0-dimensionais).
- Arestas são segmentos de reta que conectam dois vértices (unidimensionais) (Figura 2.a).
- Faces são polígonos bidimensionais representados por um ciclo de arestas (Figura 2.b).
- Células são regiões conectadas tridimensionais limitadas por faces (Figura 2.c).

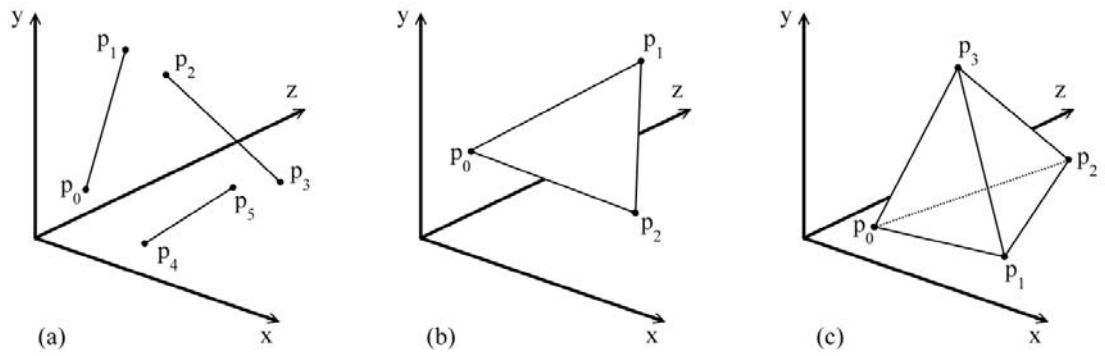


Figura 2. Conectividade geométrica básica.

(a) Arestas; (b) Face triangular; (c) Célula tetraedral.

Se todas as células de uma grade irregular são tetraedrais, dizemos que tal conjunto de dados é representado por uma grade *tetraedral*. Se as células são dadas como hexaedros, dizemos que o conjunto de dados é representado por uma grade *hexaedral*. Mencionamos explicitamente esses dois tipos de células, uma vez que essas são as representações mais comuns de grades irregulares encontradas na literatura e, também, porque os métodos que iremos apresentar neste trabalho são capazes de tratar ambos.

2.2. *Pipeline* de Renderização Volumétrica

Os algoritmos de renderização volumétrica geram imagens bidimensionais a partir de conjuntos de dados volumétricos. Estes algoritmos são especialmente apropriados para a visualização de volumes que representam campos escalares, vetoriais e tensoriais.

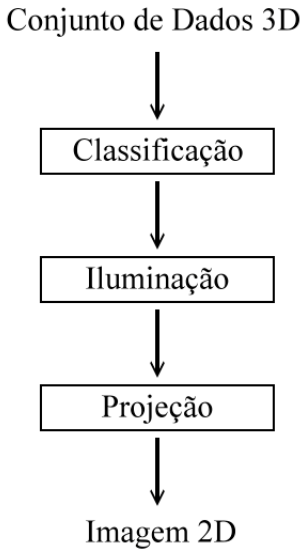


Figura 3. *Pipeline* de Renderização Volumétrica.

A maioria dos algoritmos de renderização volumétrica possuem suas operações organizadas entre os passos apresentados na Figura 3. Normalmente, os dados volumétricos de entrada são obtidos através da utilização de equipamentos como os *scanners* de ressonância magnética e tomografia computadorizada. A primeira etapa do processo de renderização volumétrica é a classificação dos dados. Essa etapa tem por objetivo permitir que o usuário identifique as estruturas existentes no conjunto de dados, podendo isolá-las e definir sua forma e extensão. Na classificação, os valores escalares a serem visualizados são mapeados em valores de cor e opacidade. Isso é feito através da

utilização de funções de mapeamento específicas para cada aplicação (funções de transferência). A seguir, o volume classificado passa pela etapa de iluminação. Esta etapa descreve a forma como a luz interage sobre os materiais dos *voxels* que formam o volume de dados, realçando a percepção tridimensional na imagem final. O último passo baseia-se na projeção do volume, previamente classificado e iluminado, em um plano de visualização, gerando, dessa forma, uma imagem bidimensional que será visualizada pelo usuário.

Cada etapa do *pipeline* de renderização volumétrica é dependente das anteriores, ou seja, qualquer alteração em uma das etapas tornará necessário que as etapas seguintes sejam executadas novamente. Por exemplo, se a posição do observador for alterada, apenas a etapa de projeção precisa ser executada. Entretanto, se o usuário mudar uma das funções de transferência, todas as etapas precisam ser executadas novamente.

Nas seções seguintes detalharemos cada uma das etapas do processo de renderização volumétrica.

2.2.1. Classificação

Podemos dizer que o passo de classificação é a etapa inicial para os algoritmos de renderização volumétrica, porque a manipulação do conjunto de dados volumétrico independe do algoritmo que será utilizado. Portanto, a classificação pode ser vista como uma etapa de pré-processamento para a adequação do volume de dados.

A etapa de classificação dos dados no *pipeline* de renderização volumétrica tem o objetivo principal de tornar possível a identificação de estruturas internas do volume, ou seja, possibilita a visualização do interior de um objeto e a exploração de suas estruturas ao invés de apenas visualizar a superfície desse objeto.

Se existe uma estrutura no conjunto de dados, ela pode se tornar visível ou invisível através do processo de classificação. O estágio de classificação atribui uma nova propriedade, chamada opacidade, a cada *voxel* do conjunto de dados. A opacidade, representada por um valor entre 0 (totalmente transparente) e 1 (totalmente opaco), descreve a quantidade de luz absorvida pelo *voxel* quando a luz incide nele. O importante é que a classificação permite que o usuário, ao atribuir uma opacidade alta a uma estrutura em um conjunto de dados, torne essa estrutura visível. Da mesma forma, ao atribuir um valor de opacidade muito pequeno a uma estrutura de pouco interesse, faz com que esta fique transparente.

A classificação é feita através de um mapeamento dos valores dos *voxels* em valores de cor e opacidade. Esta tarefa é sujeita a erros e exige que o usuário tenha noção do que espera encontrar no volume.

Uma ferramenta eficiente para auxiliar na escolha da função de mapeamento é o histograma do volume. Um histograma representa o número de vezes que cada valor

associado aos *voxels* está presente no conjunto de dados. Um histograma é um plano com os valores dos *voxels* no eixo horizontal e o número de ocorrências, ou freqüência, no eixo vertical (Figura 4). Ele fornece a informação sobre a distribuição de todos os valores possíveis do *voxel*. Assim, pode-se conhecer um pouco da estrutura dos dados do volume e decidir os valores da estrutura que se deseja visualizar. O modelo natural para realizar esse mapeamento consiste em considerar apenas a intensidade do *voxel*.

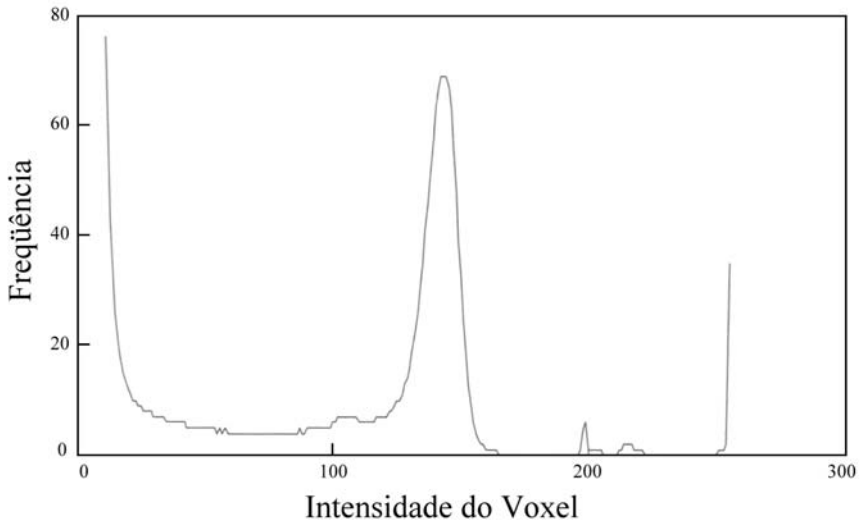


Figura 4. Histograma do conjunto de dados *engine* feito por tomografia computadorizada (LICHTENBELT *et al*., 1998).

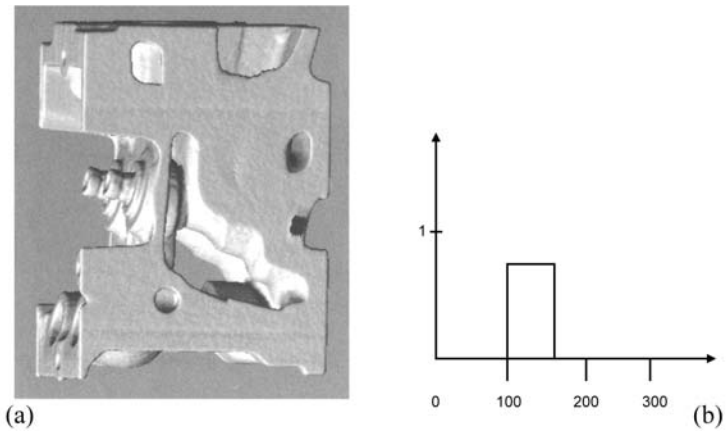


Figura 5. (a) Parte externa opaca; (b) Função de Transferência que gerou (a) (LICHTENBELT *et al*., 1998).

Na Figura 5 é possível ver uma renderização volumétrica de um conjunto de dados (*engine*) obtido por tomografia computadorizada. Se olharmos o histograma do conjunto de dados original (Figura 4), vemos que, num conjunto de dados obtido por tomografia computadorizada, existem diferentes escalas de intensidades de *voxel*. Se fixarmos a função de transferência de opacidade de forma que apenas para *voxels* com intensidades entre 100 e 170 seja atribuído um valor alto de opacidade, digamos 0.9, teremos a renderização da Figura 5.a. A função de transferência é mostrada na Figura 5.b. Com esta função de transferência, selecionamos a parte de fora da peça. Se, por outro lado, fazemos o mesmo mas com *voxels* com intensidades no intervalo entre 185 e 235, teremos a Figura 6.a. A função de transferência é mostrada na Figura 6.b. Agora, selecionamos as partes internas da peça, juntamente com uma placa de trás. Aparentemente existem, no mínimo, dois materiais diferentes nessa peça. Podemos também fazer as partes de fora da peça transparentes e as de dentro opacas. Isso é mostrado na Figura 7.a. Fizemos isso atribuindo um valor de opacidade pequeno aos *voxels* no intervalo de 100 a 170 e uma alta opacidade aos *voxels* de intensidade no

intervalo entre 185 e 235 (Figura 7.b). Claro que isso mostra apenas uma renderização possível. Mudando a função de transferência é possível fazer as partes de fora da peça mais ou menos transparentes. Essa é uma característica muito poderosa da renderização volumétrica.

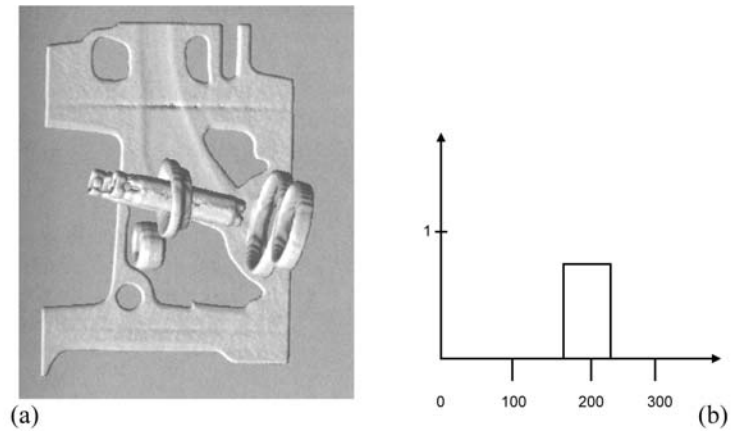


Figura 6. (a) A placa de trás e as partes internas da peça mecânica são de materiais diferentes; (b) Função de Transferência que gerou (a) (LICHTENBELT *et al*., 1998).

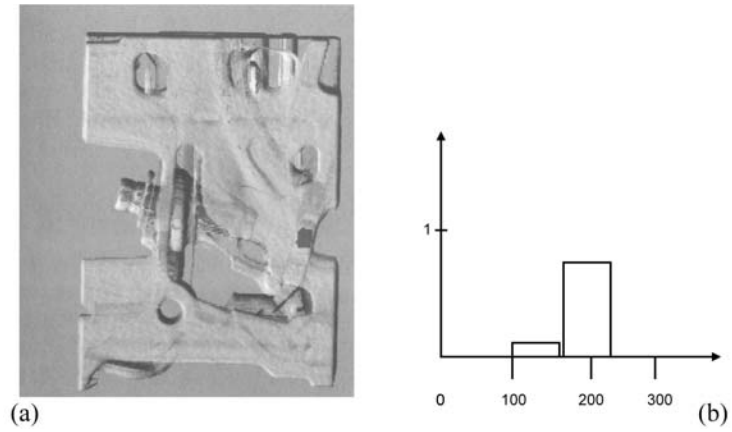


Figura 7. (a) Parte externa parcialmente transparente;

(b) Função de Transferência que gerou (a) (LICHTENBELT *et al*., 1998).

DREBIN *et al*. (1988) propuseram um sistema baseado no modelo de classificação que depende exclusivamente da intensidade do *voxel* para o caso particular de dados obtidos por tomografia computadorizada (raios X). Neste trabalho os autores observaram que é possível que, em um mesmo *voxel*, existam vários materiais diferentes. Para cada material é atribuído um valor de probabilidade ao *voxel*. Esse valor de probabilidade indica a porcentagem total do *voxel* que contém o material. Cada *voxel* possui informações de intensidade e cor para cada material nele contido. As fronteiras entre partes internas do volume podem ser detectadas pela variação de intensidade (valor do *voxel*). Para obter a função de mapeamento utiliza-se um método probabilístico levando em consideração o número de materiais presentes no volume e a porcentagem de material em cada *voxel*. A Tabela 1 mostra um exemplo do mapeamento de intensidade em materiais e destes em cores e opacidade.

Tabela 1. Classificação dos materiais.

Intensidade:	De 30 a 120	De 120 a 180	De 180 a 255
Material:	Gordura	Tecido	Osso
Cor(R, G, B):	(1.0, 0.8, 0.1)	(1.0, 0.5, 0.3)	(1.0, 1.0, 1.0)
Opacidade:	0.2	0.8	1.0

Outra função de classificação foi proposta por LEVOY (1988). Tal função combina a magnitude do gradiente com a intensidade do *voxel* para atribuir uma alta opacidade aos *voxels* que ficam na superfície ou perto dela, ou numa seção de rápida mudança do conjunto de dados.

Essa função de classificação inicia-se pela associação de uma opacidade α_i aos *voxels* que possuem valor f_v. A todos os *voxels* com valor de intensidade f_v será atribuída uma alta opacidade, assim como a todos os *voxels* em torno de f_v cuja

magnitude do gradiente também seja significante. Quanto maior a magnitude do gradiente, mais distante a intensidade do *voxel* pode estar de f_v. Essa função de classificação tem dois parâmetros: f_v e o parâmetro r que é o valor máximo que a intensidade do *voxel* pode separar-se de f_v de forma que seja atribuída ao *voxel* uma opacidade maior do que zero. A fórmula para essa função é:

$$\alpha_i(r, f_v) = 1 - \frac{1}{r\,|\nabla_i|}\,|f_v - I_i|, \qquad \text{se } |\nabla_i| > 0 \text{ e } I_i - r\,|\nabla_i| \le f_v \le I_i + r\,|\nabla_i|$$

$$\alpha_i(r, f_v) = 1, \qquad \text{se } |\nabla_i| = 0$$

$$\alpha_i(r, f_v) = 0, \qquad \text{nos demais casos}$$

onde $|\nabla_i|$ é a magnitude do gradiente no *voxel* i com intensidade I_i e $\alpha_i(r,f_v)$ é a opacidade no *voxel* i. Repare que se $f_v = I_i$, então $\alpha_i(r,f_v) = 1$.

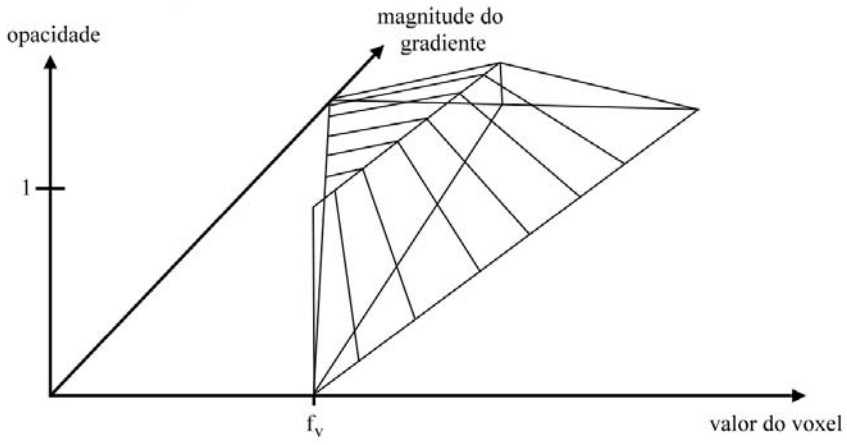


Figura 8. Função de classificação gerada pelo esquema de LEVOY (1988).

A Figura 8 ilustra a forma de uma função de classificação gerada dessa maneira. Repare que ambos, a magnitude do gradiente e a intensidade do *voxel*, são usados. As intensidades do *voxel* estão no eixo *x*, a magnitude do gradiente no eixo *y* e a opacidade resultante está no eixo *z*.

UPSON e KEELER (1988), desenvolveram um método de classificação mais avançado onde definiram duas funções de transferência, uma para cor e outra para opacidade, para a faixa de valores escalares do volume. Esses valores eram utilizados junto com os gradientes obtidos para o cálculo de luz e sombra na imagem final. Na realidade a função de transferência de cor compreende três funções de transferência, uma para cada canal de cor (vermelho, verde, azul). Essas três funções de transferência podem ser diferentes uma da outra. Se forem iguais, será gerada uma imagem em escala de cinza.

Um método de classificação mais sofisticado foi apresentado em (HE *et al*., 1996). Este trabalho discute métodos estocásticos para a geração de funções de transferência. Essa abordagem define o que deve ser visualizado através de medidas objetivas, como entropia máxima da imagem ou variação do histograma. Alguns resultados mais avançados são descritos em (MARKS *et al*., 1997).

Apesar de tantas técnicas já terem sido desenvolvidas, com o aprimoramento dos métodos de renderização volumétrica serão necessárias melhores técnicas de seleção de funções de transferência.

2.2.2. Iluminação

Essa é a etapa do *pipeline* de renderização volumétrica na qual a cor para cada *pixel* da imagem é computada. Vários modelos físicos podem ser usados. Quanto mais elaborado for o modelo, mais realísticas serão as imagens geradas e mais custoso e lento esse passo do *pipeline* se tornará.

Os modelos óticos utilizados em renderização volumétrica são baseados em modelos físicos de interação entre luz e material. A equação matemática para esse propósito é chamada de Integral de Renderização Volumétrica (BLINN, 1982; KRUEGER, 1991; MAX, 1995; KAJIYA & VON HERZEN, 1994). Analisaremos o modelo de iluminação que utilizamos em nossa implementação, onde as partículas que abrangem o material do objeto visualizado supostamente absorvem e emitem luz.

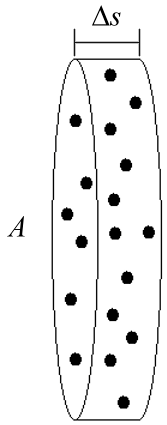


Figura 9. Uma fatia de um objeto onde A é a área da base e Δs é a largura da fatia.

Considerando-se apenas os efeitos de iluminação locais, ou seja, absorção e emissão de luz, podemos derivar a Integral de Renderização Volumétrica analisando a quantidade de luz absorvida e emitida por uma pequena fatia cilíndrica do conjunto de

dados. Considere que a base B dessa fatia cilíndrica possui área A e altura Δs, como mostrado na Figura 9, com a luz incidindo perpendicularmente à base. Para simplificar, assuma que as partículas são esferas idênticas de raio r, resultando em uma área projetada de $S = \pi r^2$. A fatia possui volume $V = A\Delta s$ e, sendo ρ a densidade de partículas no volume, existem $N = \rho A \Delta s$ partículas na fatia cilíndrica. Se Δs é pequeno o suficiente de forma que as projeções das partículas na base B tenham uma pequena probabilidade de se sobreporem, a área total obstruída por elas em B pode ser aproximada por $N\pi r^2 = \rho A \pi r^2 \Delta s$ ou $\rho \pi r^2 \Delta s$ por unidade de área da fatia. No limite quando $\Delta s \to 0$, a probabilidade de ocorrer sobreposição também se aproxima de zero, e temos a seguinte equação diferencial

$$\frac{dI}{ds} = \rho(s)\pi r^2 I_e(s) - \rho(s)\pi r^2 I_a(s) \tag{1}$$

onde os termos $I_e(s)$ e $I_a(s)$ são, respectivamente, as intensidades da luz emitida e absorvida pela fatia.

Normalmente, podemos expressar a taxa em que a luz é absorvida por $\tau(s) = \rho(s)\pi r^2$. Chamamos $\tau(s)$ de coeficiente de absorção. Então, a equação (1) pode ser escrita da seguinte maneira

$$\frac{dI}{ds} = E(s) - \tau(s) I_a(s) \tag{2}$$

onde $E(s) = \tau(s) I_e(s)$ é chamado de termo de emissão.

A equação (2) pode ser resolvida passando-se o segundo termo do lado direito para o lado esquerdo e multiplicando-se os dois lados da equação pelo termo $exp\left(\int_0^s \tau(s)dt \right)$, da seguinte maneira:

$$\left(\frac{dI}{ds}+\tau(s)I_a(s)\right)exp\left(\int_0^s \tau(t)dt\right)=E(s)exp\left(\int_0^s \tau(t)dt\right) \tag{3}$$

ou

$$\frac{dI}{ds}\left(I_a(s)exp\left(\int_0^s \tau(t)dt\right)\right)=E(s)exp\left(\int_0^s \tau(t)dt\right) \tag{4}$$

Fazendo a integração a partir de $s = 0$, na borda do volume, até $s = z$, na posição do observador, obtemos:

$$I_a(z)exp\left(\int_0^z \tau(t)dt\right)-I_0=\int_0^z\left(E(s)exp\left(\int_0^s \tau(t)dt\right)\right)ds \tag{5}$$

Passando I_0 para o outro lado, e multiplicando por $exp\left(-\int_0^z \tau(t)dt\right)$ podemos resolver para $I(z)$:

$$I_a(z)=I_0exp\left(-\int_0^z \tau(t)dt\right)+\int_0^z\left(E(s)exp\left(-\int_s^z \tau(t)dt\right)\right)ds \tag{6}$$

O primeiro termo representa a luz vinda do fundo multiplicada pela transparência $T(s)=exp\left(-\int_0^s \tau(t)dt\right)$ no intervalo entre 0 e s. O segundo termo é a integral para a contribuição do termo de emissão $E(s)$ em cada posição s, multiplicada pela transparência $T^*(s)=exp\left(-\int_s^z \tau(t)dt\right)$ entre s e o observador. Então

$$I(z)=I_0T(z)+\int_0^z E(s)T^*(s)ds \tag{7}$$

Transparência e opacidade, $O(s)$, são relacionadas da forma: $T(s) = 1 – O(s)$.

Considerando as intensidades I como as intensidades em termos de cor, absorvida ou emitida pelas partículas no objeto, podemos reescrever a equação (7) da seguinte maneira

$$C(z)=C_0+\int_0^z c(s)(1-O(s))ds \tag{8}$$

$$O(z) = O_0 + \int_0^z o(s)ds \tag{9}$$

onde $c(s)$ e $o(s)$ são as interpolações lineares de cor e opacidade, respectivamente, entre seus valores na coordenada z atual e na próxima.

Sejam z_c e z_n as coordenadas z das interseções com a célula atual e a próxima célula, respectivamente. Então, as interpolações $c(s)$ e $o(s)$ em z_c e z_n são dadas por

$$c(z) = \frac{c_c(z_n - z) + c_n(z - z_c)}{\Delta z} \tag{10}$$

$$o(z) = \frac{o_c(z_n - z) + o_n(z - z_c)}{\Delta z} \tag{11}$$

onde c_c, c_n são os valores das componentes de cor linearmente interpoladas em z_c e z_n, respectivamente, o_c, o_n são as opacidades linearmente interpoladas em z_c e z_n, respectivamente, e Δz é a diferença entre z_c e z_n.

A fim de realizarmos a integração a partir da coordenada z atual para a próxima coordenada z, mudamos os limites de integração nas equações (8) e (9) do intervalo de $(0, s)$ para o intervalo de (z_c, z_n). Portanto, as equações (8) e (9) podem ser aproximadas por

$$C_n = C_c - \frac{1}{2}(c_c + c_n)(O_c - 1)\Delta z - \frac{1}{24}(3c_c o_c + 5c_n o_c + c_c o_n + 3c_n o_n)\Delta z^2 \tag{12}$$

$$O_n = O_c + \frac{1}{2}(o_c + o_n)\Delta z \tag{13}$$

onde C_c, O_c são os valores de cor e opacidade acumulados de passos anteriores (inicialmente iguais a 0) e C_n, O_n são os valores de cor e opacidade atualizados.

As expressões analíticas (12) e (13) são responsáveis, respectivamente, pelo cálculo do acúmulo de cor e opacidade no processo de renderização volumétrica.

2.2.3. Projeção

Esta etapa do processo de renderização volumétrica recebe o volume já iluminado e gera uma imagem resultante da projeção e acumulação das contribuições de cor e opacidade de cada um dos *voxels*.

A projeção pode ser ortográfica (também chamada de paralela) ou perspectiva. No entanto, a projeção perspectiva possui a desvantagem do problema da divergência dos raios. O número de raios lançados por *voxel*, diminui à medida que se caminha sobre o raio, ou seja, a amostragem realizada nos *voxels* mais próximos do observador é mais detalhada que a realizada nos *voxels* mais distantes. Dessa forma, pequenos detalhes são perdidos ao serem projetados os *voxels* mais distantes. Uma forma, não muito eficiente, de resolver esse problema é aumentar o número de raios lançados (*oversampling*). Uma outra forma é utilizar um algoritmo adaptativo, aumentando-se a densidade de raios à medida que se afasta do observador (NOVINS *et al*., 1990).

No presente trabalho, utilizaremos projeção ortográfica.

2.3. Algoritmos de Renderização Volumétrica

Existem vários artigos publicados acerca de algoritmos de renderização volumétrica e diversos métodos já foram desenvolvidos e otimizados. Nesta seção faremos uma breve apresentação de alguns desses algoritmos.

Discutiremos os algoritmos *Splatting* (WESTOVER, 1990) e *Shear-Warp* (LACROUTE & LEVOY, 1994), que são aplicados a dados representados na forma de grades regulares, e os algoritmos *ZSweep* (FARIAS *et al*., 2000) e *Ray-Casting* (LEVOY, 1988), que tratam dados irregulares.

2.3.1. Splatting

O algoritmo *Splatting* (WESTOVER, 1990), que pode ser aplicado apenas a dados representados na forma de grades regulares, trabalha no espaço dos objetos e procura mapear cada *voxel* do volume no plano da imagem. Esse mapeamento normalmente é realizado a partir dos *voxels* mais próximos do observador até os mais distantes. Após cada *voxel* ter sido mapeado no plano da imagem, através de um processo de acumulação, sua contribuição é adicionada à formação da imagem. O algoritmo termina quando todas as primitivas tiverem sido mapeadas na tela.

O primeiro passo do algoritmo é determinar a ordem em que o volume será percorrido. Este passo é essencial para o correto cálculo da visibilidade pois a ordem correta de projeção permite acumular adequadamente as opacidades dos *voxels*. Para isso, escolhe-se os dois eixos do volume mais paralelos ao plano da imagem para formar

o *loop* mais interno. O plano formado por esses dois eixos será projetado e terá suas contribuições acumuladas em um *buffer* denominado *sheet*. A projeção é realizada fatia por fatia, ou seja, todos os *voxels* de uma determinada fatia são projetados antes que a próxima fatia seja processada. Como ocorre nos demais algoritmos, o valor de densidade de cada *voxel* é classificado de acordo com as funções de transferência de cor e opacidade e, a seguir, é iluminado utilizando a técnica de estimativa da normal através do gradiente.

A seguir vem o passo de reconstrução, a parte mais importante do algoritmo, onde é calculada a contribuição de cada *voxel* no plano da imagem. Nela o algoritmo procura reconstruir um sinal contínuo a partir de um conjunto discreto de dados, de modo a reamostrar o sinal, na resolução desejada, para gerar a imagem. Para isto, utiliza-se um filtro de reconstrução (*kernel*) para calcular a extensão da projeção do *voxel* sobre o plano da imagem. A projeção do *kernel* é chamada de *footprint* e, no caso de projeções ortográficas (ou paralelas), o *footprint* é o mesmo para todos os *voxels*. Isto significa que ele pode ser gerado em uma etapa de pré-processamento. A extensão do *footprint* está diretamente relacionada com o tamanho do volume e a resolução da imagem. Assim, se a resolução da imagem for maior que o tamanho do volume, a projeção de um único *voxel* pode ocupar vários *pixels*.

O próximo passo é o processo de visibilidade, que recebe a cor e a opacidade do *voxel* e avalia esses valores para gerar a contribuição em todos os *pixels* que estão sob a extensão do *footprint*. Os valores avaliados são compostos no *buffer* de acumulação, utilizando o esquema de acumulação apropriado à ordem de caminhamento no volume e levando em consideração a atenuação provocada pela aplicação do *footprint*. Na verdade, o *footprint* é uma tabela que determina como o *voxel* será "arremessado" sobre

o plano da imagem. Isso significa que a contribuição do *voxel* é maior no centro de projeção sobre o plano da imagem (*pixel* central) e menor nos *pixels* mais afastados. Quando um determinado *pixel* do plano de projeção acumula opacidade próxima de 1.0, este *pixel* não precisa mais ser processado.

O algoritmo *Splatting* permite gerar imagens de boa qualidade, porém é muito sensível ao tamanho da tabela de *footprint*. Tabelas pequenas geram imagens com muitos artefatos enquanto tabelas grandes suavizam demais o volume. Como a projeção é realizada de frente para atrás, uma vantagem do algoritmo é permitir que o usuário acompanhe o processo de geração da imagem final. Ao contrário do algoritmo *Ray-Casting*, o algoritmo *Splatting* projeta uma fatia do volume de cada vez, e não um *pixel* por vez.

Um algoritmo semelhante, denominado *V-buffer*, foi apresentado em (UPSON & KEELER, 1988). Esse algoritmo apresenta como diferença o fato de se basear em células e não em *voxels*. O algoritmo *V-buffer* percorre o interior da célula, interpolando os valores dos vértices e projetando cada valor interpolado no plano de visualização.

Tanto o método de *Splatting* como o *V-buffer* são facilmente paralelizáveis. A paralelização do algoritmo de *Splatting* é simples, pois a projeção de cada *voxel* é realizada de modo independente, ou seja, não é necessário considerar o restante do volume durante a projeção. Entretanto, a ordem correta de projeção deve ser respeitada.

2.3.2. Shear-Warp

O algoritmo *Shear-Warp* (LACROUTE & LEVOY, 1994; LACROUTE, 1995) é o algoritmo mais popular baseado em transformações afins do volume. Tais transformações no volume de dados simplificam a etapa de projeção do *pipeline* de visualização. Assim como o algoritmo *Splatting*, o algoritmo *Shear-Warp* apenas pode ser aplicado a dados representados na forma de grades regulares.

A base do algoritmo é a decomposição da transformação de projeção em duas etapas: uma transformação de cisalhamento (*shear*) e uma de dobra (*warp*). Estas duas etapas podem ser facilmente visualizadas na Figura 10, que apresenta uma representação bidimensional deste processo.

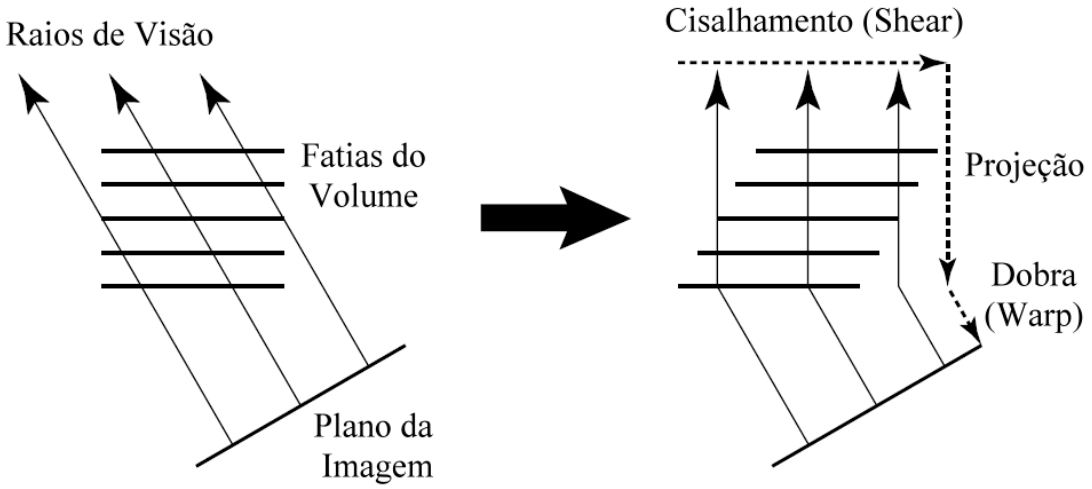


Figura 10. Algoritmo *Shear-Warp* (LACROUTE, 1995).

O cisalhamento é usado para transformar cada fatia do volume de dados para um sistema de coordenadas intermediário de modo que as fatias do volume fiquem paralelas

ao plano da imagem. Isso facilita bastante a projeção das fatias, pois os dados volumétricos são acessados na ordem de armazenamento, gerando uma imagem intermediária distorcida. Por ser apenas uma transformação geométrica afim que simplesmente translada as fatias do volume, o cisalhamento não é, portanto, computacionalmente caro.

As dimensões do volume interferem na resolução da imagem intermediária porque a área total ocupada pela projeção das fatias cisalhadas devem estar contidas nesta imagem. Esse fato é importante para acelerar a etapa de composição dos valores dos *voxels*. Nesta etapa é comum o surgimento de problemas de *aliasing*.

A imagem final é obtida através da aplicação da transformação de dobra (*warp*) na imagem intermediária distorcida. Essa transformação é realizada em 2D e restaura as reais dimensões da imagem que será visualizada pelo usuário.

Como a construção da imagem intermediária é realizada *scanline* a *scanline*, pode-se tornar o algoritmo ainda mais eficiente se utilizarmos uma estrutura do tipo RLE (*run length encoding*) para representar as *scanlines* de cada fatia do volume. Na implementação proposta em (LACROUTE, 1995), a estrutura RLE é utilizada para representar os elementos não transparentes, fazendo com que os conjuntos de *voxels* transparentes adjacentes em uma scanline da fatia do volume não sejam processados. Essa estrutura, que é construída em um passo de pré-processamento, consiste de uma seqüência de *voxels* adjacentes que são todos transparentes ou todos não transparentes.

Utilizando-se a codificação em RLE para a imagem intermediária, pode-se facilmente ignorar *voxels* transparentes ou associados a *pixels* já opacos na imagem, o que permite acelerar o procedimento de criação da imagem intermediária.

2.3.3. Ray-Casting

O algoritmo *Ray-Casting* (LEVOY, 1988) é um algoritmo exato de renderização volumétrica muito utilizado quando se quer obter imagens de alta qualidade. Ele opera no espaço da imagem, diferentemente do algoritmo *Splatting* discutido na seção 2.3.1., que opera no espaço dos objetos. A idéia básica desse algoritmo é lançar um raio através de cada *pixel*, atravessando o volume de dados. A cor e a opacidade encontradas nas partes do volume intersectadas pelo raio são acumuladas para se determinar a cor final do *pixel*.

Suponha que desejamos renderizar, por exemplo, uma imagem a partir de um volume composto por células tetraedrais (sendo possível utilizar modelos com células de formas diferentes). Um raio é disparado a partir do centro de um *pixel* através desse volume (Figura 11).

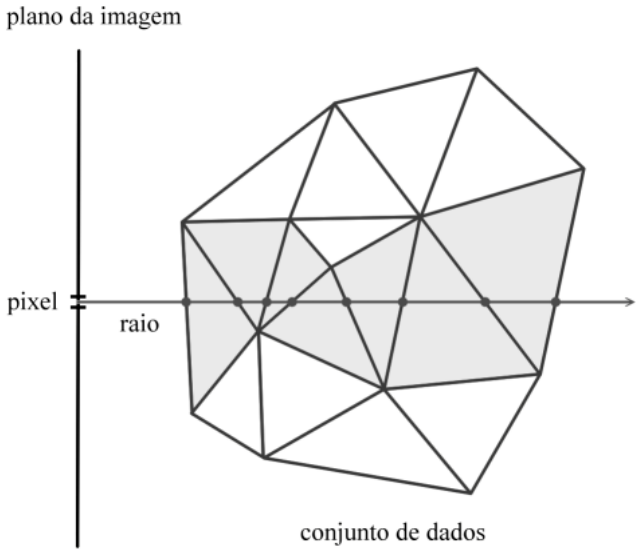


Figura 11. Esquema simplificado do algoritmo de ray-casting.

O primeiro passo do algoritmo é encontrar a primeira face intersectada pelo raio lançado através do *pixel*. Esta face é dita ser uma face externa e, a partir dela, são obtidos os valores iniciais de cor e opacidade.

Depois de encontrar a primeira face intersectada pelo raio, caminha-se através do conjunto de dados por meio de informações sobre a conectividade das células. A cada nova face intersectada pelo raio, são acumulados os valores de cor e opacidade por meio da aplicação da integral de iluminação até que o raio saia inteiramente do volume ou que o valor da opacidade acumulada atinja o valor máximo (1). O resultado final da composição dos valores de cor e opacidade das faces por onde o raio passa é a cor do *pixel*.

Em geral, são utilizadas no algoritmo de *Ray-Casting* estruturas que contenham as células e, junto com elas, informações que permitam determinar suas células vizinhas, para que, após encontrar a primeira face intersectada pelo raio, as demais sejam encontradas caminhando-se através da tetraedração por meio das adjacências. A interseção do raio com uma célula do volume pode ocorrer em uma face, em um vértice ou em uma aresta (Figura 12).

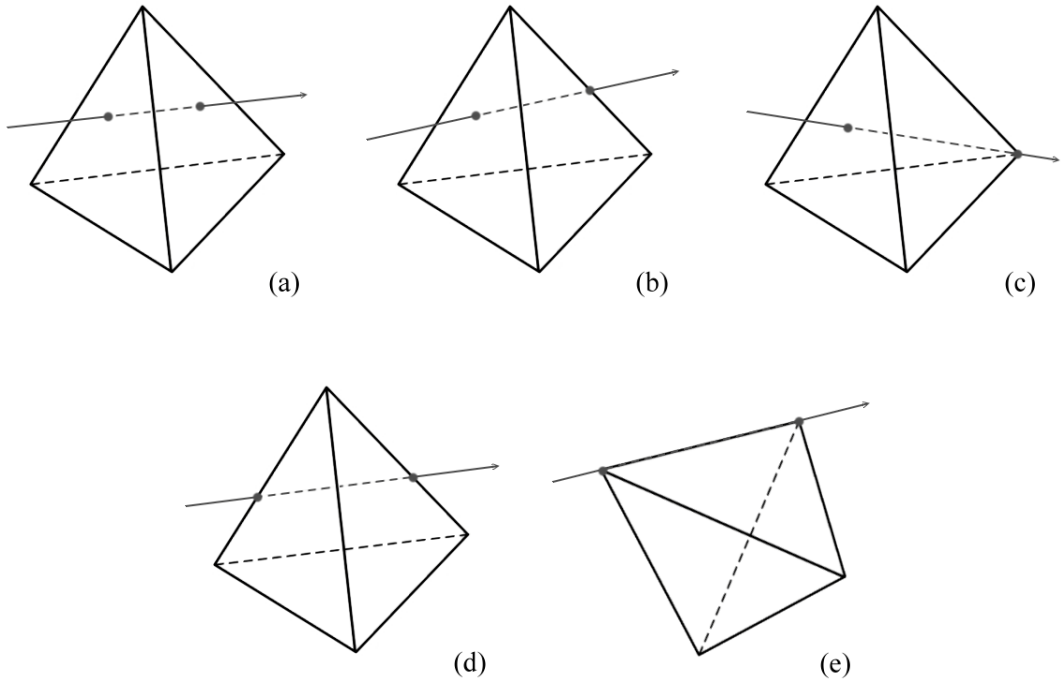


Figura 12. Cinco casos possíveis de interseção do raio com uma célula do volume. O raio intersecta: (a) duas faces; (b) uma face e uma aresta; (c) uma face e um vértice; (d) duas arestas da mesma face; (e) dois vértices da mesma aresta.

Um problema que ocorre no algoritmo de *Ray-Casting* é que, quando o raio intersecta a célula em um vértice ou em uma aresta, pode ser muito difícil determinar qual será a próxima célula e, se ela não for encontrada, a composição de cor e opacidade será interrompida, causando um erro na cor final do *pixel*. Dizemos que este é o caso degenerado e, portanto, a próxima célula intersectada precisa ser encontrada de forma especial. A Figura 13 apresenta em duas dimensões, dois tipos de casos degenerados possíveis. Apesar de os casos degenerados serem difíceis de acontecer, estes precisam ser levados em conta em uma implementação de *Ray-Casting* que pretenda obter bons resultados independentemente do modelo ou da posição em que ele se encontra.

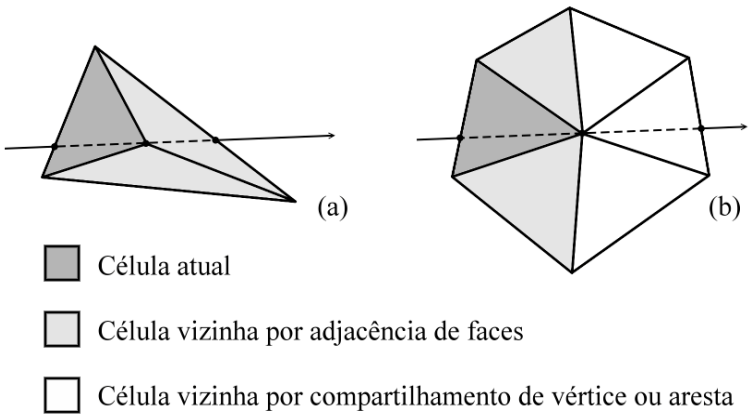


Figura 13. Exemplo em 2D de dois casos degenerados. (a) A próxima célula é vizinha da célula atual por adjacência de faces; (b) A próxima célula é vizinha da célula atual por compartilhamento de vértice ou aresta;

As primeiras implementações do algoritmo de *Ray-Casting* foram projetadas para grades retilíneas regulares (SABELLA, 1988; UPSON & KEELER, 1988; DREBIN *et al*., 1988; LEVOY, 1988) e fizeram uso da regularidade do volume. No entanto, fazer *Ray-Casting* em volumes curvilíneos, é mais difícil do que em volumes retilíneos devido a cálculos não triviais de localização das células e interseções ao longo do curso do raio. WILHELMS *et al*. (1990) descreveram uma rápida alternativa para renderizar grades curvilíneas mapeando-as em grades retilíneas.

Muitos algoritmos para *Ray-Casting* de conjuntos de dados na forma de grades curvilíneas, sem mapeamento no espaço retilíneo, seguem o raio de face a face e calculam as faces de saída e os pontos de interseção para cada célula por onde o raio passa (GARRITY, 1990; USELTON, 1991; KOYAMADA, 1992; MA & PAINTER, 1993; MA, 1995). GARRITY (1990) decompõe hexaedros em 5 ou 24 tetraedros para eliminar complicações devido a possíveis células côncavas. HONG e KAUFMAN (1998; 1999) propuseram uma técnica de *Ray-Casting* muito rápida para grades

curvilíneas. Um método de *Ray-Casting* para grades irregulares foi apresentado em (BUNYK *et al*., 1999). Tal método será discutido mais detalhadamente na próxima seção e serviu de base para o desenvolvimento de nosso trabalho e comparação de nossos resultados.

Apesar de gerar imagens de alta qualidade, o algoritmo *Ray-Casting* pode ter um custo computacional muito grande, dependendo da resolução final da imagem, isto é, da quantidade de raios a serem lançados pelo volume.

Existem vários métodos de aceleração do algoritmo. O mais conhecido deles é o chamado término precoce do raio (*early ray termination*) que permite interromper o processo de composição quando a opacidade acumulada atingir o valor máximo 1. É possível, também, utilizar *octrees* para se passar rapidamente por regiões homogêneas do volume (intensidade constante), o que simplifica muito o processo de integração ao longo do raio. Tais otimizações podem ser encontradas em (LEVOY, 1990a; LEVOY, 1990b).

Uma característica muito importante do algoritmo *Ray-Casting* é a facilidade com que se é possível implementar a criação progressiva de uma imagem. Isso é feito percorrendo-se uma grade regular sobre o plano de visualização e diminuindo-se, a cada etapa, o passo na grade. Dessa forma, a imagem é amostrada rapidamente em baixa resolução (Figura 14.a), sendo refinada até a resolução final (Figura 14.d). Como cada *pixel* da imagem é calculado uma única vez, o tempo de cálculo da cor dos *pixels* não é alterado. A Figura 15 apresenta um exemplo de geração de uma imagem com a utilização de refinamento progressivo.

Como o lançamento dos raios são totalmente independentes entre si, esse algoritmo é facilmente paralelizável.

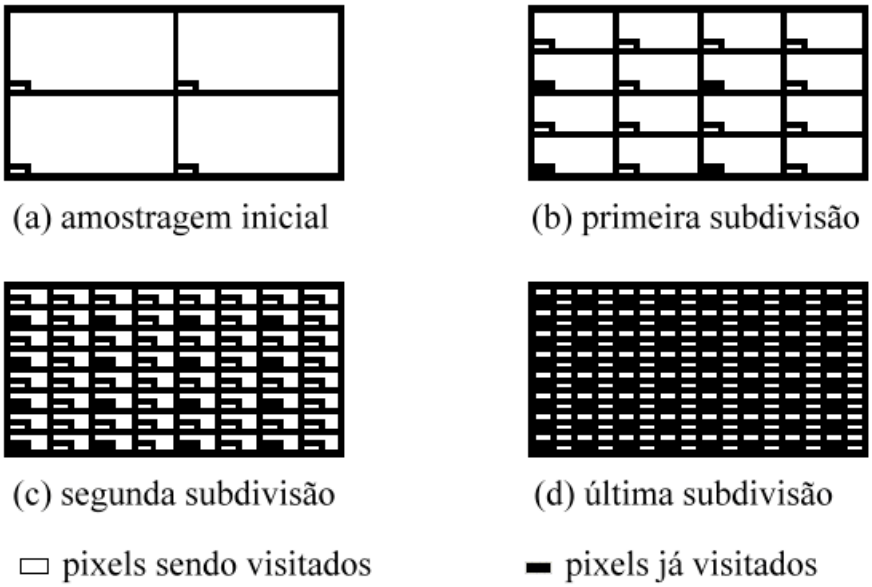


Figura 14. Refinamento progressivo (PAIVA *et al*., 1999).

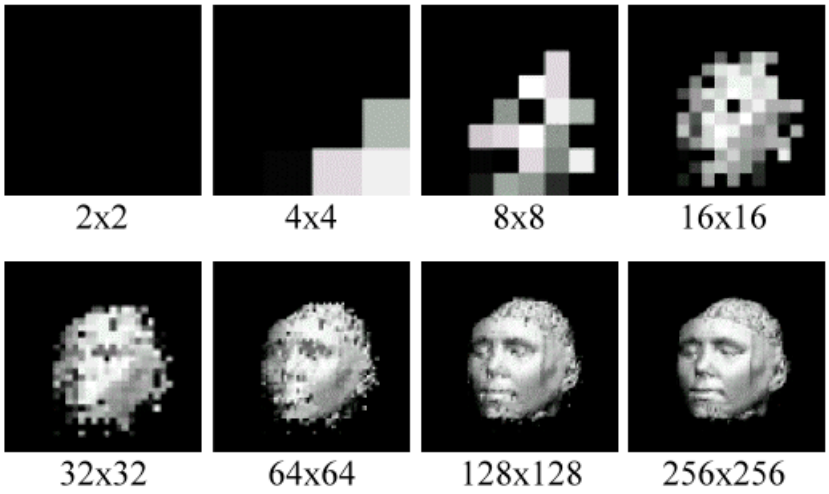


Figura 15. Exemplo de geração progressiva da imagem (PAIVA *et al*., 1999).

2.3.4. Ray-Casting (BUNYK *et al.*, 1999)

A implementação do algoritmo de ray-casting desenvolvida por BUNYK *et al*. (1999) é simples e rápida. No entanto, requer uma quantidade muito grande de memória para o processamento. Esta implementação suporta conjuntos de dados na forma de grades irregulares formadas por células tetraedrais.

Nesta seção apresentaremos as estruturas básicas utilizadas nesta implementação, a forma como o algoritmo de ray-casting foi organizado e como foi feito o tratamento de casos degenerados.

Pré-Processamento e Estruturas Básicas

Na etapa de pré-processamento é realizada, basicamente, a reconstrução da conectividade a partir dos dados. Como o conjunto de dados de entrada consiste em uma coleção de células e vértices compartilhados, à medida que ele é lido, todos os vértices e células (tetraedrais) são armazenados, assim como todas as faces resultantes do desmembramento das células. Uma vez que duas células vizinhas compartilham a mesma face, para evitar que duas faces iguais sejam armazenadas é criada uma lista chamada *referredBy* para cada vértice, composta de todas as faces que o usam. Quando cada face é lida, a lista *referredBy* de seus vértices é atualizada e, para evitar o armazenamento de faces iguais, simplesmente é feita uma busca na lista *referredBy* de seus vértices, antes da inserção, para verificar se a face já foi listada. Essa busca na lista *referredBy* é uma busca linear simples, bastante rápida na prática, uma vez que o grau de cada vértice geralmente é baixo.

Então, as estruturas básicas utilizadas por BUNYK *et al*. são três *arrays* principais que guardam os dados: o *array* de vértices, o *array* de células e o *array* de faces.

- Cada vértice é composto pelas coordenadas x, y e z do vértice, um valor α e uma lista *referredBy* que relaciona o vértice às faces a que ele pertence.
- Cada célula é composta por uma lista chamada de *consistsOf* que relaciona a célula às faces que a formam.
- Cada face é composta por um conjunto de variáveis, uma lista, também chamada de *referredBy*, que relaciona a face às células que a compartilham, e uma lista, também chamada de *consistsOf*, que relaciona a face aos vértices que a formam. O conjunto de variáveis serve para guardar valores necessários para a verificação da interseção do raio com a face e para a interpolação dos valores α dos vértices da face. Se a lista *referredBy* tem tamanho igual à 1, a face está na fronteira do conjunto de dados.

Passos do Algoritmo

Na implementação desenvolvida por BUNYK *et al*., o algoritmo de ray-casting foi apresentado da seguinte maneira:

- Em etapa de pré-processamento, identifique todas as células e faces adjacentes a cada vértice e todas as faces externas.
- De acordo com a rotação fornecida em relação a cada um dos eixos *x*, *y* e *z*, rotacione cada um dos vértices.

- Projetando todas as faces externas na tela, crie, para cada *pixel*, uma lista ordenada com as faces externas por onde o raio disparado pelo *pixel* atravessa o volume, entrando. Tais faces são ditas visíveis.
- Para cada *pixel*, começando pela primeira face externa intersectada, utilize a informação sobre a adjacência da célula para encontrar a próxima face intersectada pelo raio. Cada face interior aponta para suas duas células vizinhas, permitindo que se passe facilmente de célula para célula enquanto a contribuição de cada célula está sendo computada. Encontrada a próxima face, proceda da mesma forma até que o raio saia do volume. Cor e opacidade são acumuladas a cada passo, para formar a cor final do *pixel*.

Tratamento de Casos Degenerados

Para verificar se o raio lançado pelo *pixel* intersecta uma face, foi utilizado um procedimento que faz a transformação da face para o plano *xy* e verifica se o ponto lançado através do *pixel* (x, y) está nela, simplificando o problema de interseção do raio com uma face 3D. No entanto, quando é necessário encontrar a próxima face intersectada pelo raio, duas situações podem ocorrer: encontrar mais de uma face ou não encontrar uma face cuja coordenada z de profundidade seja maior que a atual. Isso acontece se um raio intersecta um vértice ou uma aresta. Nesse caso, o programa tenta encontrar a próxima face entre todas as faces de todas as células adjacentes à atual.

2.3.5. ZSweep

O algoritmo *ZSweep* (FARIAS *et al*., 2000; FARIAS, 2001) realiza projeção de células de forma rápida e eficiente em memória para renderização (exata) de conjuntos de dados na forma de grades irregulares. Baseado no paradigma de varredura (*sweep*), a idéia principal desse algoritmo é varrer os dados com um plano paralelo ao plano da tela, em ordem crescente de z (profundidade), e a medida que os vértices são encontrados pelo plano de varredura, as faces das células incidentes a eles são projetadas. O algoritmo faz a projeção das células projetando cada uma de suas faces, evitando projeção dupla das faces internas e assegurando a ordem de projeção correta. A contribuição para cada *pixel* é computada em estágios, durante a varredura, usando uma pequena lista de interseções de face ordenadas, a qual é sabida ser correta e completa no momento em que é completado cada estágio da computação.

O algoritmo é uma simples varredura com um plano Π, paralelo ao plano de visão, na ordem crescente das coordenadas z. Eventos ocorrem quando Π encontra um vértice v. Em tal ponto as faces das células que são incidentes em v e que se encontram além de v (em relação a coordenada z) são projetadas. Para cada vértice v é criada uma lista, chamada *useset*, que guarda todas as células incidentes em v.

O primeiro passo do algoritmo é ordenar os vértices pela coordenada z em uma lista de eventos (*event list*) para determinar a ordem dos eventos. A estrutura escolhida para a lista de eventos foi uma *heap*, para ordenar os vértices eficientemente. Essa *heap* armazena apenas índices para o *array* de vértices.

O loop principal do algoritmo é a varredura na direção do eixo z, que é realizada simplesmente caminhando-se através da lista de eventos. Quando o i-ésimo vértice, v_i,

da lista de eventos é encontrado, cada face *f* que é incidente em v_i é "projetada", onde v_i é o vértice que tem o *z* mínimo. Necessariamente, tais faces *f* não foram varridas, ou seja, nenhum vértice de *f* foi varrido anteriormente. As faces a serem projetadas são determinadas examinando-se o *use set* de v_i. Veja na Figura 16 uma ilustração em duas dimensões. A palavra "projetada" apareceu entre aspas porque, diferentemente de outros métodos como em (SHIRLEY & TUCHMAN, 1990), no método de *ZSweep* quando as faces são projetadas, simplesmente é computada a interseção do raio e guardado o valor de *z*. Posteriormente é que são realizados os cálculos de iluminação.

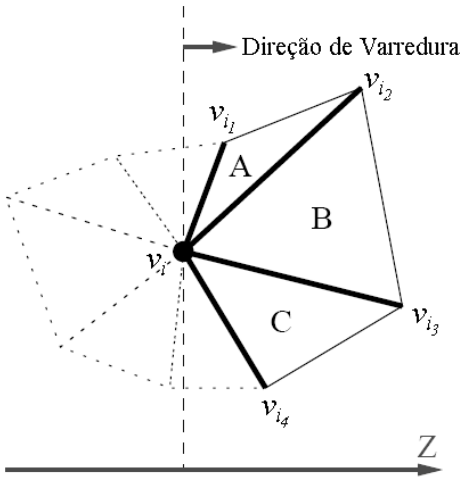


Figura 16. Quando o plano Π encontra o vértice v_i, as células A, B e C são encontradas, então as faces (v_i, v_{i_1}), (v_i, v_{i_2}), (v_i, v_{i_3}) e (v_i, v_{i_4}) são projetadas (FARIAS *et al*., 2000).

A fim de realizar a projeção das faces, foi utilizada uma conversão muito rápida para triângulos, que não somente determina que *pixel* está na projeção, mas também determina a coordenada *z* (profundidade) para cada ponto do triângulo (não projetado) e computa o valor interpolado para os dados em campo escalar.

Para garantir a precisão do algoritmo de renderização, é importante ter certeza de que a projeção das faces é feita na ordem correta para cada *pixel*. Essa ordem não é, contudo, suficiente para garantir que as faces sejam projetadas automaticamente na ordem correta de profundidade para todos os *pixels*. Fazendo a varredura na ordem de profundidade dos vértices e projetando as faces incidentes sobre os mesmos, o algoritmo executa uma projeção quase correta para cada *pixel*. Para isso o *ZSweep* mantém, para cada *pixel*, uma lista de interseções ordenadas em *z*, projetadas nesse *pixel*. Por exemplo, na Figura 16, as faces (v_i, v_{i_1}) e (v_i, v_{i_2}) são ambas projetadas quando encontramos v_i. Enquanto uma análise local das faces em v_i irá permitir projetar (v_i, v_{i_1}) antes de (v_i, v_{i_2}), também teria que ser projetada a face (v_{i_1}, v_{i_2}) antes de (v_i, v_{i_2}) a fim de que os *pixels* na projeção de (v_{i_1}, v_{i_2}) tenham a correta ordenação das faces projetadas. No entanto, a face (v_{i_1}, v_{i_2}) não é projetada até que o plano Π alcance o vértice v_{i_1}. Enquanto que em duas dimensões é possível projetar faces (arestas de triângulos) em ordem de *z*, em três dimensões a relação de precedência induzida pela ordenação de profundidade pode ter ciclos. Note que todas as células pontilhadas na Figura 16 são consideradas células varridas.

Qualquer passo da varredura tem um chamado *target-z*, que representa o próximo valor de *z* no qual o algoritmo irá parar a varredura momentaneamente e compor os valores que estão nas listas de *pixel*. A varredura continua além do *target-z* e em seguida é determinado um novo *target-z* de modo apropriado.

Inicialmente, o *target-z* é definido como sendo a coordenada máxima entre os vértices adjacentes ao primeiro vértice, v_0, encontrado pelo plano Π. Quando o plano de varredura alcança o *target-z* (vértice *v*, por exemplo), são compostas, em ordem, as

entradas das listas do *pixel* no valor acumulado sendo guardado para cada *pixel*, começando-se pela última coordenada *z* onde a composição terminou para esse *pixel*, e terminando quando é alcançada a profundidade do *target-z*. No exemplo da Figura 16, se v_i for o *target-z* inicial, então, quando ele for encontrado, o novo *target-z* será a coordenada *z* do vértice v_{i_3}.

Para evitar projeção dupla, todas as faces das células no *useset* do vértice atual são guardadas em uma tabela de *hash*. Percorre-se, então, a tabela de *hash* e projetam-se todas as faces.

Antes da projeção, o código chama a função de composição se a coordenada *z* atual do plano de varredura alcançou o *target-z* ou se existe pelo menos uma lista de *pixel* de tamanho maior do que um tamanho *threshold*.

O último passo do algoritmo é chamado de composição atrasada (*delayed compositing*) e é responsável por computar a contribuição de cor e opacidade de todas as faces projetadas, até este momento.

O algoritmo de *ZSweep*, por ser simples, é facilmente adaptável a diversos formatos de células (não tetraedral). É eficiente em memória, uma vez que suas estruturas de dados auxiliares tem apenas que guardar informações parciais tomadas de um pequeno número de "fatias" do conjunto de dados.

3. IMPLEMENTAÇÕES

Durante nossa pesquisa, foram desenvolvidas duas implementações do algoritmo de Raycast. Estas implementações são versões otimizadas em uso de memória, fazendo uso apenas de uma estrutura de dados auxiliar para representar as adjacências.

Na primeira implementação, que chamamos de ME-Ray (Memory Efficient Ray-Casting), visamos desenvolver uma implementação do algoritmo de Raycast que fosse competitiva com a implementação feita por BUNYK *et al*. tanto em memória quanto em tempo de execução.

Na segunda implementação, que chamamos de EME-Ray (Enhanced Memory Efficient Ray-Casting), demos maior importância à economia de memória do que ao tempo gasto para o processamento.

A principal diferença entre o ME-Ray e o EME-Ray é que, no primeiro, armazenamos estruturas muito custosas que permitem acesso direto à informações relevantes ao processo de ray-casting e no segundo essas estruturas não são armazenadas. Esse "não armazenamento" faz com que precisemos recalcular tais valores todas as vezes que forem necessárias e esse é o motivo do EME-Ray ser mais lento e ocupar menos memória.

Mesmo que não consideremos o uso de memória e o tempo de execução das implementações, nossos métodos são superiores ao desenvolvido por BUNYK *et al*. porque, além de nossos códigos poderem processar modelos com células tetraedrais e/ou hexaedrais, também possuem um tratamento completo para casos degenerados. Mostraremos que, em muitos casos, o algoritmo desenvolvido por BUNYK *et al*. gera

imagens com alguns *pixels* não renderizados completamente, e por isso errados, devido ao tratamento simplificado para casos degenerados.

Neste capítulo apresentaremos nossos métodos de renderização volumétrica ME-Ray e EME-Ray.

3.1. ME-Ray (Memory Efficient Ray-Casting)

Nossa implementação do algoritmo de Raycast, ME-Ray (Memory Efficient Ray-Casting), foi desenvolvida tomando por base algumas estruturas de dados utilizadas no algoritmo *ZSweep* (FARIAS *et al*., 2000). Nosso objetivo principal foi criar uma aplicação do algoritmo de ray-casting que combinasse precisão de resultados e eficiência em tempo e memória. Para tanto, escolhemos estruturas de dados que exigissem menos memória do que as utilizadas por BUNYK *et al*. e desenvolvemos um procedimento que trata de forma eficaz os casos degenerados que podem surgir durante o processo de ray-casting.

Nas seções seguintes apresentaremos as estruturas básicas utilizadas no nosso método ME-Ray, como o algoritmo de Raycast foi organizado, a forma de tratamento de casos degenerados e o como foi feito o procedimento para lidar com conjuntos de dados sobrepostos.

3.1.1. Pré-Processamento e Estruturas Básicas

Assim como na implementação de BUNYK *et al*., a etapa de pré-processamento no ME-Ray realiza uma reconstrução da conectividade a partir dos dados. O conjunto de dados de entrada é uma coleção de células com vértices compartilhados. Enquanto o conjunto de dados de entrada é lido e analisado, todos os vértices e células são armazenados.

Então, as estruturas básicas utilizadas em nosso algoritmo são três *arrays* principais que guardam os dados: o *array* de vértices (*array Points_VEC*), o *array* de células (*array Cells_VEC*) e o *array* de faces (*array Faces_VEC*).

- Cada vértice é composto pelas coordenadas *x*, *y* e *z* do vértice, um valor α e uma lista (*Use_set*) que contém todas as células adjacentes a ele, como a estrutura utilizada no *ZSweep* (FARIAS *et al*., 2000).
- Cada célula é composta por um *array* de índices de vértices (*array Vertices*), um *array* de índices de células vizinhas (*array Neighbour*) e um *array* de índices de faces triangulares (*array Triangular_Faces*) (em células hexaedrais, as faces precisam ser desmembradas em duas faces triangulares) e um índice que define seu tipo como tetraedral ou hexaedral.
- Cada face é composta por um conjunto de variáveis e por 3 índices que indicam os pontos que a formam. O conjunto de variáveis serve para guardar valores necessários para a verificação da interseção do raio com a face e para a interpolação dos valores α dos vértices da face.

Para tornar a conectividade mais rápida e fácil construímos o que chamamos de *Use_set* para cada vértice, o que nos fornece uma lista de todas as células incidentes no vértice, ao contrário da lista *referredBy* utilizada por BUNYK *et al*. que fornece uma lista de faces que usam o vértice. O *Use_set* pode ser construído em tempo linear passando pelos dados. Outro passo na fase de pré-processamento é achar as faces vizinhas de cada célula. Cada face do conjunto de dados é verificada apenas uma vez e, a partir de um único cálculo encontramos as duas células que compartilham essa face. A partir daí, cada célula irá guardar os índices de suas células vizinhas. Isso facilita na hora de caminhar através do conjunto de dados durante o ray-casting.

O *array* de faces, que chamamos de *array Faces_VEC*, é criado durante o processo de ray-casting à medida que as faces vão aparecendo nos cálculos de interseção com o raio lançado através do *pixel*. O objetivo é inserir apenas as faces que serão utilizadas na verificação de interseção dos raios, evitando assim, a inserção de faces que não serão encontradas por nenhum raio devido à definição da imagem final.

A Estrutura da Célula

Nossa implementação pode suportar modelos formados por células tetraedrais e/ou hexaedrais. Cada célula é composta por um *array* de índices de vértices (*array Vertices*), um *array* de índices de células vizinhas (*array Neighbour*), um *array* de índices de faces triangulares (*array Triangular_Faces*) e um índice que define seu tipo como tetraedral ou hexaedral. Na Tabela 2 podemos ver o número de vértices, faces, faces triangulares e células vizinhas para cada tipo de célula.

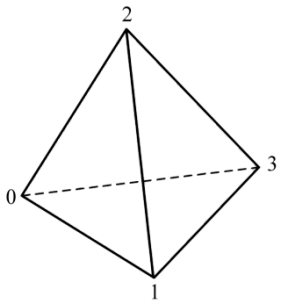
Tabela 2. Número de vértices, faces, faces triangulares e células vizinhas para cada tipo de célula.

Tipo de Célula	Vértices	Faces	Faces Triangulares	Células Vizinhas
Tetraedral	4	4	4	4
Hexaedral	8	6	12	6

Quando a célula é hexaedral, precisamos dividir cada uma de suas faces poligonais em duas faces triangulares, portanto, cada célula possui 12 faces para serem inseridas no *array* de faces.

Pela Tabela 2 vemos que, se cada índice ocupa a memória de um *unsigned* (4 bytes), uma célula ocupa 4*($v + ft + cv + tp$) bytes de memória, onde v é o número de vértices, ft é o número de faces triangulares, cv é o número de células vizinhas da célula e $tp = 1$ indica o índice do tipo da célula. Portanto, uma célula tetraedral ocupa 52 bytes de memória e uma célula hexaedral ocupa 108 bytes de memória.

Na Figura 17.a vemos uma célula tetraedral. Para encontrarmos os vértices da face de uma célula tetraedral a partir do índice dessa face, utilizamos as relações dispostas na Figura 17.b. Na Figura 18.a vemos uma célula hexaedral e na Figura 18.b, a relação entre índice de faces e de vértices para este tipo de célula. Note que a face de uma célula hexaedral é formada por 4 vértices e, portanto, precisa ser desmembrada em duas faces triangulares.

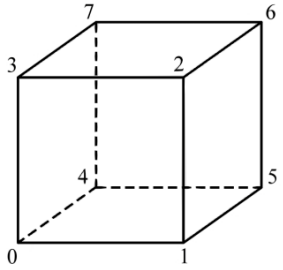


(a)

Face	Vértices		
0	0	1	2
1	0	1	3
2	1	2	3
3	0	2	3

(b)

Figura 17. Estrutura de uma Célula Tetraedral. (a) Célula e seus índices de vértices; (b) Relação entre os índices das faces e os índice dos vértices.



Face	Vértices			
0	0	1	2	3
1	4	5	6	7
2	0	1	5	4
3	3	2	6	7
4	0	3	7	4
5	1	2	6	5

(a) (b)

Figura 18. Estrutura de uma Célula Hexaedral. (a) Célula e seus índices de vértices; (b) Relação entre os índices das faces e os índice dos vértices.

3.1.2. Passos do Método ME-Ray

Em nosso método ME-Ray, o algoritmo de ray-casting foi organizado da seguinte maneira:

- Primeiramente, os dados de entrada são lidos e armazenados nos *arrays* *Cells_VEC* e *Points_VEC*.

- Uma lista *Use_set* é criada para cada vértice contido em *Points_VEC*.
- Para cada célula contida em *Cells_VEC*, são encontradas e listadas, no seu respectivo *array Neighbour*, todas as suas células vizinhas.

  Cada célula vizinha é representada por seu índice em *Cells_VEC*. Por exemplo, suponha que as células c_1 e c_2, cujos índices em *Cells_VEC* são 53 e 124, respectivamente, são vizinhas por compartilharem uma face. O índice dessa face compartilhada em relação a cada célula pode ser diferente, então suponha que, na célula c_1, seu índice seja 0 e, na célula c_2, seja 2. Atribuiremos, portanto, à posição 0 do *array Neighbour* da célula c_1, o valor 124 e, à posição 2 do *array Neighbour* da célula c_2, o valor 53. Se a célula faz parte da fronteira do conjunto de dados, o índice da célula vizinha é o próprio índice da célula em *Cells_VEC*. Dessa maneira, posteriormente, serão identificadas as faces externas.

- Para cada *pixel*, é criada uma lista *Ext_Faces* para armazenar as faces externas visíveis intersectadas pelo raio lançado através do *pixel*.

  Para economizar memória, a lista *Ext_Faces* armazena apenas índices de células e de faces. Esses índices fornecem a informação necessária para se identificar a face que deveria ter sido armazenada.

- A função de transferência é lida e os valores de cor e opacidade referentes a cada intensidade são armazenados em *arrays* (3 para cor e 1 para opacidade).

- De acordo com a rotação fornecida pelo usuário, todos os pontos do conjunto de dados são rotacionados.
- São armazenadas nas listas *Ext_Faces* de cada *pixel* as faces externas visíveis, ou seja, as faces externas cujo ângulo entre o vetor normal à face e o vetor direção do raio lançado através do *pixel* é maior que 90^o.

  Essa verificação é feita da seguinte maneira:

  Corremos o *array Cells_VEC* e identificamos as faces externas. A cada face externa encontrada, calculamos as fronteiras da face projetada no plano da imagem e verificamos, para cada *pixel* contido nesses limites de fronteira, se a face é intersectada pelo raio lançado através dele ou não. Se for, verificamos se ela é visível, pegando o vértice (x_v, y_v, z_v) da célula que não pertence à face testada e comparando a coordenada z_v desse vértice com a coordenada z do plano definido pelos três vértices da face em x_v, y_v. Se z_v é maior que z então a face é visível.

  Sempre que é testada a visibilidade de uma face os valores de seus parâmetros são calculados e a face é inserida no *array* de faces (*array Faces_VEC*). Quando uma nova face *f* é inserida em *Faces_VEC*, as células (no máximo duas) formadas com a ajuda de *f* recebem, no seu *array Triangular_Faces*, na posição referente ao índice de *f* na célula, o valor da posição de *f* em *Faces_VEC*. Por exemplo, se verificamos que a face 3 de uma célula é externa e visível e, no *array Faces_VEC*, seu índice é 25, então, na posição 3 do *array Triangular_Faces* da célula, será inserido o valor 25.

- Com as listas *Ext_Faces* de cada *pixel* devidamente completas, começa o processo de ray-casting.

  Para cada *pixel*:

  - Pegamos sua lista *Ext_Faces* correspondente. Como a lista *Ext_Faces* é formada por índices de células e de faces, criamos uma outra lista temporária com informações mais completas. Nessa lista, que chamamos de *Ext_Faces_tmp*, inserimos a face, o índice da célula, a coordenada *z* da interseção do raio com a face, o valor da interpolação dos valores α dos vértices que compõem a face, o índice da face na célula e o tipo da célula. Todos esses parâmetros são necessários para o nosso processo de ray-casting. A lista *Ext_Faces_tmp* ordena seus elementos, com relação à coordenada *z*, no momento da inserção, funcionando como uma árvore ordenada.

    Quando temos uma célula hexaedral e o raio lançado através do *pixel* intersecta uma face dessa célula exatamente em algum ponto da diagonal onde a face poligonal foi cortada, as duas faces serão admitidas como visíveis para o mesmo *pixel*. O tipo de ordenação feito na lista *Ext_Faces_tmp* também elimina essa duplicação de faces provenientes do desmembramento de faces poligonais.

  - Tendo a lista *Ext_Faces_tmp* completa, pegamos o primeiro elemento dela para começarmos o processo de ray-casting. Procuramos a próxima face intersectada pelo raio, verificando todas as faces da célula atual diferentes da face externa. Se a posição referente à face testada da célula

no *array Triangular_Faces* da célula não foi preenchida, a face e seus parâmetros são calculados e inseridos no *array Faces_VEC* e o *array Triangular_Faces* da célula é atualizado. Senão, pegamos o valor armazenado no *array Triangular_Faces* da célula, na posição referente ao índice da face na célula, e encontramos a face armazenada no *array Faces_VEC* cujo índice seja igual a esse valor armazenado.

Se não encontrarmos a próxima face intersectada pelo raio entre as faces da célula atual, podemos ter a ocorrência de um caso degenerado onde precisaremos verificar todas as faces de todas as células contidas no *Use_set* de cada um dos vértices da célula atual. Uma explicação mais detalhada do procedimento de tratamento de casos degenerados será feita na próxima seção.

- Encontrada a próxima face, calculamos o acúmulo de cor e opacidade através da aplicação da Integral de Renderização Volumétrica, apresentada na seção 2.2.2.. Esse processo se repete até que o raio saia do volume, ou seja, até que a lista *Ext_Faces_tmp* do *pixel* esteja vazia e nenhuma célula seja encontrada no teste da próxima face.

Se o usuário desejar fazer a visualização do mesmo conjunto de dados em vários ângulos diferentes, realizando a rotação total desejada em pequenos intervalos, não é necessário que os dados sejam lidos novamente. No entanto, como o *array Faces_VEC* já foi construído na primeira imagem gerada, apenas atualizamos os valores dos parâmetros de face de acordo com a nova rotação realizada no volume.

3.1.3. Tratamento de Casos Degenerados

Utilizamos o mesmo algoritmo utilizado por BUNYK *et al*. (1999) para verificar se o raio intersecta a face. Quando o raio lançado pelo *pixel* intersecta um vértice ou uma aresta da célula, são identificadas como faces intersectadas as três faces adjacentes ao vértice ou as duas faces que compartilham a aresta na célula, respectivamente. Como a função identifica mais de uma face como sendo a próxima face intersectada na célula, a primeira a ser verificada será a escolhida. Esse tipo de escolha pode acarretar um erro na verificação das células seguintes e, por isso, verificam-se todas as faces de todas as células contidas no *Use_set* de cada um dos vértices da célula atual.

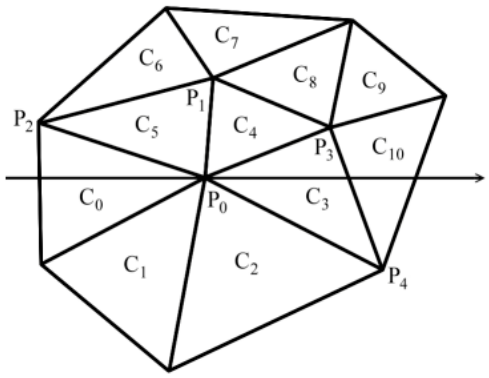


Figura 19. Esquema de caso degenerado

Vejamos como se comporta a função através de um exemplo. Suponha que a Figura 19 representa uma fatia de um conjunto de dados 3D, então cada triângulo representa uma célula e cada aresta do triângulo representa uma face da célula.

A primeira face a ser intersectada pelo raio lançado através do *pixel* é a face externa da célula C_0.

1º passo – Para encontrar a próxima face intersectada pelo raio, podemos aplicar o teste usual, verificando apenas as faces da célula atual (C_0). Encontramos a próxima célula através do *array* de células vizinhas da célula C_0. Suas células vizinhas são C_5 e C_1. Como o ponto P_0 de interseção com o raio é um vértice ou uma aresta, qualquer face que o contenha será escolhida para ser a próxima. Se a face compartilhada pelas células C_0 e C_5 for a primeira testada para a verificação da interseção com o raio lançado através do *pixel*, teremos que a célula C_5 será a próxima célula a ser visitada.

2º passo – Agora, a célula atual é a célula C_5. Se buscarmos a próxima face aplicando o teste usual não obteremos resultado. Com isso, precisamos entrar na função que trata de casos degenerados para verificar se existe uma próxima face. Se verificarmos apenas as faces das células incidentes no vértice P_1, também não obteremos resultado. Por esse motivo, procuramos a próxima face entre todas as faces de todas as células incidentes em cada um dos vértices da célula atual (C_5). Tal busca seria feita, portanto, em todas as faces incidentes em P_0, P_1 e P_2. Dessa maneira encontraremos como resultado a face compartilhada pelas células C_3 e C_{10} e a próxima célula a ser visitada será a célula C_3.

3º passo – Neste momento, a célula atual é a célula C_3. Buscando a próxima face pelo teste usual, não encontraremos nenhuma face intersectada pelo raio lançado pelo *pixel* cuja coordenada *z* seja maior que a da face compartilhada pelas células C_3 e C_{10}. Dessa forma, aplicamos o segundo teste, procurando entre todas as células incidentes nos vértices P_0, P_3 e P_4. Através deste teste encontramos a face externa da célula C_{10} e saímos do volume, terminando, portanto, com a composição de cor e opacidade neste *pixel*.

É importante ressaltar que o algoritmo desenvolvido por BUNYK *et al*. não teria encontrado resposta no 2º passo, pois teria testado apenas as faces das células C_0, C_4 e C_6, adjacentes à célula C_5. Nesse caso, a cor final do *pixel* ficaria errada, pois a composição de cor e opacidade teria sido prematuramente interrompida.

3.1.4. Procedimento para Modelos Sobrepostos

Apenas com o objetivo de aumentar a robustez de nossas implementações, criamos um procedimento para tratar de modelos formados por conjuntos de dados sobrepostos. Em um modelo formado pelos conjuntos de dados C_1 e C_2, as células do conjunto de dados C_1 não possuem nenhuma relação de conectividade com as células do conjunto de dados C_2.

Um algoritmo de renderização volumétrica baseado no método de projeção de células como, por exemplo, o *ZSweep* (FARIAS *et al.*, 2000) pode ser facilmente adaptado para trabalhar com modelos formados por conjuntos de dados sobrepostos uma vez que as células são decompostas em faces triangulares e tais faces são projetadas em ordem no plano da imagem. O principal obstáculo que existe para o método de ray-casting processar esse tipo de modelo provém do fato de que as células dos dois conjuntos de dados que o formam não possuem conexões umas com as outras. No caso de um modelo formado por dois conjuntos de dados formados por células de tipos diferentes podemos, por exemplo, encontrar uma célula tetraedral dentro de uma célula hexaedral e vice-versa.

Para que possamos fazer o ray-casting de um modelo formado por dois conjuntos de dados sobrepostos, precisamos fazer o seguinte:

- Suponha que temos um modelo onde os dois conjuntos de dados que o formaram são compostos por células de tipos diferentes (A e B) e que lançamos um raio através do *pixel* (x, y). Suponha que a lista de faces externas do *pixel* (x, y) contém faces de células de ambos os tipos. A lista de faces externas é ordenada em ordem crescente em relação a distância z da face com relação ao plano da imagem.
- Para iniciar o procedimento de ray-casting, pegamos como ponto de partida a primeira face armazenada na lista de faces externas intersectadas pelo raio lançado através do *pixel* (x, y). Suponha que a célula a que essa face pertence seja do tipo A.
- Se caminharmos através das células por onde o raio passa, sempre encontraremos células do tipo A. Isso acontece porque as células do tipo B não possuem nenhum tipo de conexão com as do tipo A. Sendo assim, só iremos calcular o acúmulo de cor e opacidade a partir das células do tipo A, como se as células do tipo B não estivessem sendo intersectadas pelo raio. Para evitar isso, após encontrar cada próxima face, verificamos se a próxima face existente na lista de faces externas possui um z menor do que o da face encontrada. Caso isso aconteça, a face da lista passa a ser a próxima e a que seria a próxima entra na lista de faces externas. Com isso, conseguimos caminhar através do conjunto de dados seguindo a ordem real em que as faces aparecem, independente do tipo da célula a que ela pertence.

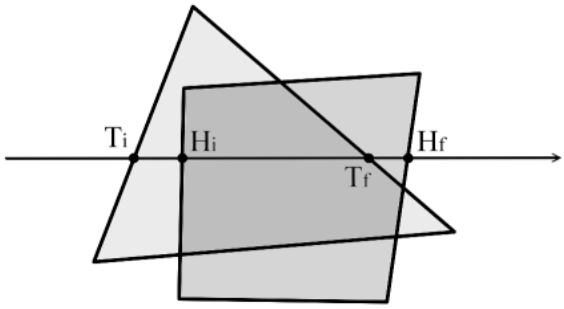


Figura 20. Raio passando por um conjunto de dados sobreposto.

Na Figura 20 vemos uma representação 2D de um conjunto de dados sobreposto sendo intersectado pelo raio. Nomearemos cada face de acordo com seu ponto de interseção com o raio, ou seja, chamaremos de T_i a face cuja interseção com o raio é o ponto T_i e assim por diante. As faces armazenadas inicialmente na lista do *pixel* são as faces T_i e H_i, pois são as faces externas visíveis do conjunto de dados. Entendemos por faces externas visíveis as faces que pertencem a apenas uma célula e cuja normal forma um ângulo maior que 90^o com o raio lançado pelo *pixel*. Começando o processo de ray-casting a partir da face T_i, pois é a primeira face intersectada pelo raio, encontramos como sendo a próxima face intersectada a face T_f. Como a coordenada *z* de T_f é maior que a coordenada *z* de H_i, retiramos a face H_i da lista do *pixel* e inserimos a face T_f. Então, calculamos a acumulação de cor e opacidade em relação às faces T_i e H_i. Sendo H_i a face atual, encontramos a face H_f como sendo a próxima. Como a coordenada *z* de H_f é maior que a coordenada *z* de T_f, retiramos a face T_f da lista do *pixel* e inserimos a face H_f. Então, fazemos a composição de cor e opacidade a partir das faces H_i e T_f. Como a partir da face T_f não encontramos uma próxima face, pegamos a última face existente na lista do *pixel* (H_f) e fazemos a composição de cor e opacidade entre as faces T_f e H_f. Sendo a face H_f a face atual, como não encontramos uma próxima face e a lista

de faces do *pixel* está vazia, o raio saiu do volume. Portanto, o resultado da acumulação de cor e opacidade até esse momento, é a cor final do *pixel*.

3.2. EME-Ray (Enhanced Memory Efficient Ray-Casting)

Nossa implementação do algoritmo de Raycast, EME-Ray (Enhanced Memory Efficient Ray-Casting), foi desenvolvida com o intuito de otimizar o método ME-Ray em custo de memória. Nosso objetivo principal, neste caso, foi criar uma aplicação do algoritmo que ray-casting que tivesse uma grande eficiência em memória, quase similar ao *ZSweep* (FARIAS *et al*., 2000). É claro que a diminuição do custo em memória causou um aumento do tempo necessário para o processamento, porque retiramos estruturas custosas do ME-Ray para economizar memória e essas mesmas estruturas, se não forem guardadas, precisam ser atualizadas a cada passo do algoritmo.

Como as diferenças entre o ME-Ray e o EME-Ray limitam-se às estruturas de dados utilizadas, as explicações sobre o tratamento para casos degenerados e o suporte para modelos sobrepostos, feitas nas seções 3.1.3 e 3.1.4, respectivamente, também se aplicam a esse método. A organização do algoritmo de ray-casting feita nesse método também é muito semelhante. Portanto, na seção seguinte apresentaremos as diferenças entre as estruturas de dados utilizadas no ME-Ray e no EME-Ray.

3.2.1. Diferenças nas Estruturas com relação ao ME-Ray

As estruturas básicas utilizadas no EME-Ray são dois *arrays* principais que guardam os dados: o *array* de vértices e o *array* de células.

- A estrutura dos vértices é idêntica a utilizada no ME-Ray.
- Da estrutura da célula, retiramos o *array Triangular_Faces*. Portanto, nesse método, uma célula tetraedral ocupa 36 bytes de memória e uma célula hexaedral ocupa 60 bytes de memória. Então, no *array* de células, temos uma economia de cerca de $(16T + 48H)$ bytes de memória, onde T é o número total de células tetraedrais e H é o número total de células hexaedrais contidas no conjunto de dados.

Note que retiramos o *array* de faces (*array Faces_VEC*), que era uma das estruturas mais custosas em memória do algoritmo ME-Ray. Devido à ausência do *array* de faces, economizamos cerca de $52F$ bytes de memória, onde F é o número total de faces triangulares contidas no conjunto de dados. Portanto, em um conjunto de dados com mais de 1 milhão de faces, economizamos mais de 52 Mb de memória.

Como não armazenamos mais as faces, precisamos recalcular os parâmetros para verificação de interseção do raio e para a interpolação dos valores α dos vértices cada vez que uma nova face é verificada. Fazendo isso, os parâmetros de uma mesma face podem ser recalculados mais de 100 vezes, deixando o algoritmo mais lento. No entanto, como estamos dando maior importância à economia de memória, podemos não levar em consideração esse atraso.

4. RESULTADOS EXPERIMENTAIS

Nesse capítulo, mostraremos testes que provam que nossos métodos ME-Ray e EME-Ray atingiram os objetivos esperados, comparando seus resultados aos da implementação de ray-casting de BUNYK *et al*. e aos do *ZSweep* (FARIAS *et al*., 2000) que, apesar de não se basear no paradigma de lançamento de raios, é um dos algoritmos de renderização volumétrica mais rápidos e exatos já apresentados até então. Não comparamos nossos resultados com os de algoritmos que utilizam aceleração feita por *hardware*, uma vez que nossas implementações são puramente em *software*.

Nas seções seguintes apresentaremos os conjuntos de dados de teste e a avaliação de performance dos métodos ME-Ray e EME-Ray e a comparação dos resultados em relação à implementação do algoritmo de ray-casting de BUNYK *et al*. e ao *ZSweep*.

4.1. Conjuntos de Dados

Em nossos experimentos foram utilizados 7 conjuntos de dados diferentes, numa tentativa de incluir vários tamanhos e dificuldades. Tais conjuntos de dados são formados por células tetraedrais e/ou hexaedrais. Nossos códigos podem receber dados que representem conjuntos de dados côncavos, desconexos e até com "buracos" na forma de um arquivo similar ao formato *off* do *Geomview*. No momento da leitura dos dados, nosso código é capaz de determinar o tipo de cada célula a partir do número de índices lidos, como é feito no *ZSweep* (FARIAS *et al*., 2000). A imagem resultante pode ser salva em um arquivo no formato ppm.

Dos sete conjuntos de dados testados, quatro deles são disponibilizados pela NASA: *Blunt Fin*, *Combustion Chamber*, *Liquid Oxygen Post* e *Delta Wing*. Uma vez que nossa implementação foi desenvolvida para lidar com dados volumétricos na forma de grades irregulares, utilizamos as versões tetraedralizadas destes conjuntos de dados. Outros dois conjuntos de dados utilizados para teste foram: *SPX*, que é um conjunto de dados tetraedral contendo buracos, e *Hexahedral*, que é um pequeno conjunto de dados hexaedral. Tais conjuntos de dados foram selecionados com o intuito de verificar a funcionalidade de nossa implementação no caso de o volume de dados conter buracos ou ser formado por células hexaedrais. Por fim, para testar nossa aplicação em conjuntos de dados sobrepostos, ou seja, que contenham células tetraedrais e hexaedrais ao mesmo tempo, criamos o conjunto de dados *HexaSPX* que é um misto dos modelos *Hexahedral* e *SPX* (com menor número de células).

A Tabela 3 fornece as informações básicas sobre todos os sete conjuntos de dados utilizados em nossos experimentos. Essa tabela mostra o número de células (tetraedrais/hexahedrais), o número total de vértices e faces e o número de vértices e faces da fronteira de todos os conjuntos de dados.

Tabela 3. Informações básicas sobre os conjuntos de dados.

Informações sobre os Conjuntos de Dados					
Conjunto de Dados	**Vértices**	**Vértices da Fronteira**	**Faces**	**Faces da Fronteira**	**Células**
Blunt Fin	40.960	6.760	381.548	13.516	187.395
Comb. Chamber	47.025	7.810	437.888	15.616	215.040
Oxygen Post	109.744	13.840	1.040.588	27.676	513.375
Delta Wing	211.680	20.736	2.032.084	41.468	1.005.675
SPX	149.224	22.080	1.677.888	44.160	827.904
Hexahedral	2.684	1.352	6.432	1.344	1.920
HexaSPX	5.580	2.732	33.684	4.104	14.856

4.2. Performance dos métodos ME-Ray e EME-Ray

Nessa seção apresentaremos a performance dos métodos ME-Ray e EME-Ray em um Pentium 4 com processador de 2.80GHz e 1GB de memória.

As Tabelas 4 e 5 mostram o tempo de pré-processamento e a memória necessária para criar imagens de diferentes tamanhos no ME-Ray e no EME-Ray, respectivamente. O tempo de pré-processamento é igual nos dois métodos porque consistem das mesmas operações de leitura dos dados, geração do *Use_set* de cada vértice e verificação das células vizinhas. Já durante o pré-processamento o método EME-Ray utiliza menos memória que o ME-Ray uma vez que não armazena índices de faces nas células e não aloca memória para o *array* de faces. Como, para cada *pixel*, é criada uma lista para armazenar as faces externas visíveis intersectadas pelo raio lançado através dele, o uso de memória aumenta linearmente de um tamanho de imagem para o outro.

Tabela 4. Tempo de Pré-processamento e uso de Memória no ME-Ray.

Tempo de Pré-processamento e uso de Memória no ME-Ray					
Conjunto de Dados	**Memória Necessária (MB)**				**Tempo (s)**
	128^2	**256^2**	**512^2**	**1024^2**	
Blunt Fin	21,38	21,96	24,26	33,48	0,78
Comb. Chamber	24,37	24,94	27,25	36,46	0,91
Oxygen Post	56,40	56,98	59,28	68,50	2,20
Delta Wing	106,23	106,80	109,11	118,32	4,55
SPX	89,47	90,04	92,35	99,26	3,51
Hexahedral	1,61	2,19	4,49	13,71	0,03
HexaSPX	3,04	3,62	5,92	15,14	0,09

Tabela 5. Tempo de Pré-processamento e uso de Memória no EME-Ray.

Tempo de Pré-processamento e uso de Memória do EME-Ray					
Conjunto de Dados	**Memória Necessária (MB)**				**Tempo (s)**
	128^2	**256^2**	**512^2**	**1024^2**	
Blunt Fin	16,26	16,84	19,14	28,36	0,78
Comb. Chamber	18,49	19,06	21,37	30,58	0,91
Oxygen Post	42,36	42,94	45,24	54,46	2,20
Delta Wing	81,71	82,29	84,59	93,81	4,55
SPX	66,83	67,41	69,71	78,93	3,51
Hexahedral	1,50	2,08	4,38	13,60	0,03
HexaSPX	2,58	3,15	5,46	14,67	0,09

4.3. Comparações com outros Métodos

Comparamos nossos resultados com os de dois algoritmos de renderização para conjuntos de dados na forma de grades irregulares: o algoritmo de ray-casting de BUNYK *et al*. e o algoritmo *ZSweep* (FARIAS *et al*., 2000). Verificamos, para 7 conjuntos de dados diferentes, o tempo de renderização e o uso total de memória.

As Tabelas 6-10 mostram a comparação de performance dos nossos métodos com o ray-casting de BUNYK *et al*. e o *ZSweep*. Essas tabelas relacionam o número de *pixels* renderizados, o tempo total de processamento e a memória necessária para renderizar imagens de quatro tamanhos diferentes.

A Tabela 6 apresenta a comparação no conjunto de dados *Blunt Fin*. Nosso método ME-Ray, para esse conjunto de dados, mostrou-se superior ao método de BUNYK *et al*. em consumo de memória, para todos os tamanhos de imagem considerados, e apresentou uma pequena diferença nos tempos de renderização. Note que a memória necessária para renderizar uma imagem 128x128 no método de BUNYK

et al. é quase a mesma necessária para renderizar uma imagem 1024x1024 no ME-Ray. Em comparação com o método *ZSweep*, o ME-Ray mostrou-se superior em tempo em todos os casos. Como o consumo de memória do *ZSweep* aumenta consideravelmente para imagens maiores por causa das listas de *pixels*, para imagens 512x512 e 1024x1024, nosso método ME-Ray mostrou-se superior também em consumo de memória. Para renderizar uma imagem 1024x1024, nosso método (ME-Ray) chega a ser 3,4 vezes mais rápido e usar quase 2,5 vezes menos memória do que o *ZSweep*. Embora o método de BUNYK *et al*. torne-se mais rápido quanto maior for o tamanho da imagem em comparação com o EME-Ray, este chega a gastar quase 3,5 vezes mais memória para renderizar uma imagem de tamanho 512x512. Para renderizar uma imagem 2048x2048, nosso método usa cerca de 88Mb de memória, quase o mesmo necessário para se obter uma imagem com a metade do tamanho pelo método de BUNYK *et al*.. Já, com relação ao método *ZSweep*, nosso método EME-Ray apresentou bons resultados. Dos quatro tamanhos de imagens considerados, em três deles nosso método mostrou-se superior em tempo e em consumo de memória, chegando a ser cerca de 1,4 vezes mais rápido e usar 4,8 vezes menos memória, para renderizar uma imagem 1024x1024. Note, que para todos os tamanhos de imagem testados, o número de *pixels* renderizados pelo método de BUNYK *et al*. é menor, ou seja, a imagem gerada apresenta *pixels* não-renderizados, devido ao tratamento precário para casos degenerados aplicado nesse método. A Figura 21 mostra uma imagem 512x512 desse conjunto de dados, gerada por um de nossos métodos.

Tabela 6. Uso de tempo e memória do conjunto de dados *Blunt Fin*.

Comparação de Performance no Conjunto de Dados *Blunt Fin*				
Tamanho da Imagem	**Método**	**Número de *Pixels***	**Tempo (s)**	**Memória (MB)**
128^2	BUNYK *et al*.	5.021	0,24	66,97
	ZSweep	5.023	1,07	15,41
	ME-Ray	5.023	0,35	35,38
	EME-Ray	5.023	0,54	16,36
256^2	BUNYK *et al*.	20.239	0,72	67,93
	ZSweep	20.242	3,14	22,72
	ME-Ray	20.242	0,96	41,90
	EME-Ray	20.242	2,03	17,21
512^2	BUNYK *et al*.	81.256	2,26	71,78
	ZSweep	81.266	9,65	51,84
	ME-Ray	81.266	3,13	49,87
	EME-Ray	81.266	7,58	20,58
1024^2	BUNYK *et al*.	325.679	8,91	87,17
	ZSweep	325.710	38,65	164,00
	ME-Ray	325.710	11,25	66,07
	EME-Ray	325.710	27,13	34,05

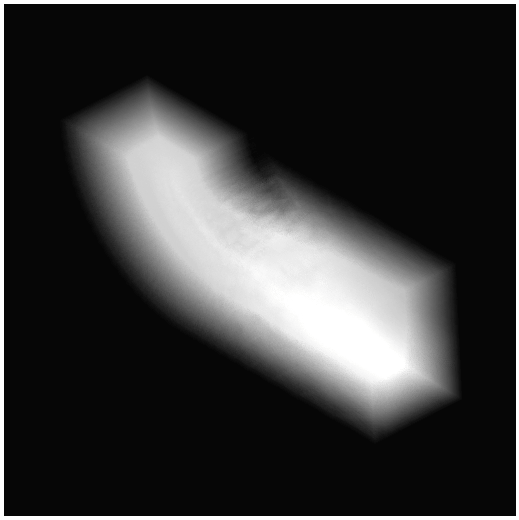


Figura 21. Imagem do *Blunt Fin* criada em 512x512.

A Tabela 7 apresenta a comparação no conjunto de dados *Combustion Chamber*. Para esse conjunto de dados, nosso método ME-Ray, também mostrou-se superior ao método de BUNYK *et al*. em consumo de memória. Note que, para gerar uma imagem 1024x1024 o método de BUNYK *et al*. é apenas 1 segundo mais rápido e usa 1,3 vezes mais memória. Essa diferença no tempo de ray-casting se dá devido aos testes de verificação do tipo de célula (tetraedral ou hexaedral) e ao tratamento mais apurado de casos degenerados feitos em nosso método. Em comparação com o método *ZSweep*, o ME-Ray, embora gaste mais memória na maioria dos tamanhos de imagem considerados, mostrou-se superior em tempo em todos os casos. Para imagens grandes (1024x1024), nosso método é cerca de 2,6 vezes mais rápido, utilizando quase 1,3 vezes menos memória que o *ZSweep*. Já, nosso método EME-Ray, chega a gastar 3,6 vezes menos memória que o método de BUNYK *et al*. para renderizar uma imagem de tamanho 512x512. Para a maioria dos tamanhos de imagens considerados nos testes, nosso método mostrou-se mais rápido e menos custoso em memória que o *ZSweep*. A Figura 22 mostra uma imagem 512x512 desse conjunto de dados, gerada por um de nossos métodos.

Tabela 7. Uso de tempo e memória do conjunto de dados *Combustion Chamber*.

Comparação de Performance no Conjunto de Dados *Combustion Chamber*				
Tamanho da Imagem	**Método**	**Número de *Pixels***	**Tempo (s)**	**Memória (MB)**
128^2	BUNYK *et al*.	5.206	0,33	76,66
	ZSweep	5.206	1,06	16,08
	ME-Ray	5.206	0,61	56,52
	EME-Ray	5.206	0,55	18,58
256^2	BUNYK *et al*.	21.026	0,96	77,65
	ZSweep	21.027	3,14	20,09
	ME-Ray	21.027	1,36	57,82
	EME-Ray	21.027	2,07	19,44
512^2	BUNYK *et al*.	84.364	3,50	81,57
	ZSweep	84.364	9,30	36,10
	ME-Ray	84.364	5,37	61,23
	EME-Ray	84.364	8,14	22,85
1024^2	BUNYK *et al*.	338.047	14,32	97,25
	ZSweep	338.049	39,90	97,80
	ME-Ray	338.049	15,23	74,85
	EME-Ray	338.049	32,09	36,46

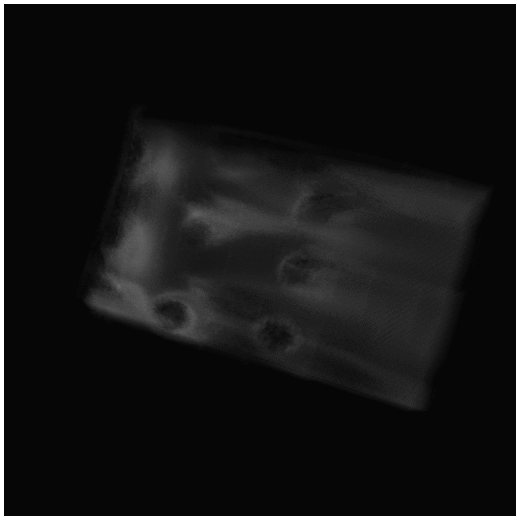


Figura 22. Imagem do *Combustion Chamber* criada em 512x512.

A Tabela 8 apresenta a comparação no conjunto de dados *Liquid Oxygen Post*. Nosso método ME-Ray, para esse conjunto de dados, chega a usar quase a metade da memória necessária no método de BUNYK *et al*. para gerar uma imagem 256x256. Em comparação com o método *ZSweep*, para gerar uma imagem 1024x1024, nosso método ME-Ray é cerca de 2,6 vezes mais rápido e usa 1,5 vezes menos memória. Em comparação com o nosso método EME-Ray, embora o método de BUNYK *et al*. torne-se mais rápido quanto maior for o tamanho da imagem, este chega a gastar 3,9 vezes mais memória para renderizar uma imagem de tamanho 512x512. O EME-Ray, para gerar uma imagem 1024x1024, necessita de 1/3 da memória utilizada na formação de uma imagem 512x512 no método de BUNYK *et al*.. Já em relação ao *ZSweep,* para todos os tamanhos de imagens considerados, nosso método mostrou-se mais rápido. A memória necessária para a criação de uma imagem 256x256 é quase igual nos dois métodos. Em uma imagem 512x512 nosso método foi 1,5 vezes mais rápido e utilizou 1,6 vezes menos memória que o *ZSweep*. Note que, para esse conjunto de dados, o método de BUNYK *et al*. novamente gera imagens defeituosas, chegando a apresentar 38 *pixels* não-renderizados em uma imagem 1024x1024. A Figura 23 mostra uma imagem 512x512 desse conjunto de dados, gerada por um de nossos métodos.

Tabela 8. Uso de tempo e memória do conjunto de dados *Liquid Oxygen Post*.

Comparação de Performance no Conjunto de Dados *Liquid Oxygen Post*				
Tamanho da Imagem	**Método**	**Número de *Pixels***	**Tempo (s)**	**Memória (MB)**
128^2	BUNYK *et al*.	4.836	0,37	176,00
	ZSweep	4.836	2,50	35,96
	ME-Ray	4.836	0,60	78,01
	EME-Ray	4.836	0,74	42,46
256^2	BUNYK *et al*.	19.493	1,09	177,00
	ZSweep	19.497	3,83	43,26
	ME-Ray	19.497	1,50	89,06
	EME-Ray	19.497	2,93	43,41
512^2	BUNYK *et al*.	78.267	3,87	180,00
	ZSweep	78.278	13,40	72,38
	ME-Ray	78.278	5,53	103,00
	EME-Ray	78.278	10,70	46,68
1024^2	BUNYK *et al*.	313.730	13,93	195,00
	ZSweep	313.768	48,83	184,00
	ME-Ray	313.768	19,03	130,00
	EME-Ray	313.768	40,33	60,20

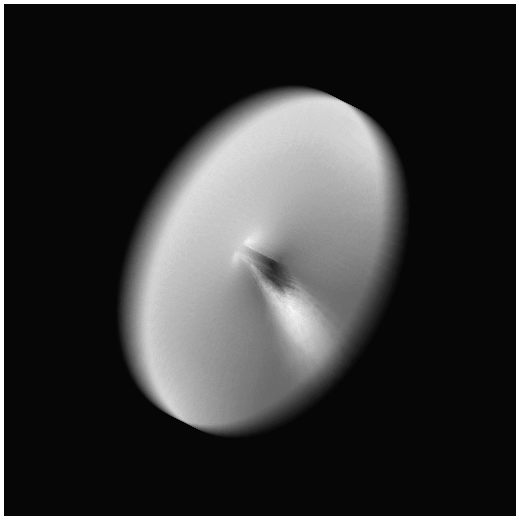


Figura 23. Imagem do *Liquid Oxygen Post* criada em 512x512.

A Tabela 9 apresenta a comparação no conjunto de dados *Delta Wing*. Para esse conjunto de dados, nosso método ME-Ray, apesar de ser 1,5 vezes mais lento que o método de BUNYK *et al*., usa 1,6 vezes menos memória para gerar uma imagem 1024x1024. Nesse conjunto de dados, uma imagem 512x512 gerada pelo método de BUNYK *et al*. chega a apresentar 333 *pixels* defeituosos. Então, essa diferença no tempo de renderização, não representa nenhuma vantagem, uma vez que a imagem final é de uma qualidade muito pior que a gerada por nosso método. Em comparação com o método *ZSweep*, o ME-Ray, embora gaste mais memória em todos os tamanhos de imagem considerados, mostrou-se superior em tempo. Para gerar uma imagem 512x512, nosso método é 3 vezes mais rápido que o *ZSweep*. O método EME-Ray usa menos de 1/4 da memória necessária no ray-casting de BUNYK *et al*. para renderizar uma imagem 512x512. Como, em nossos métodos, guardamos os índices das células vizinhas a cada célula, e esse modelo possui mais de 1 milhão de células, temos, no EME-Ray, um acréscimo considerável na quantidade de memória necessária para armazenar os dados em relação ao método de *ZSweep*. Por esse motivo, para imagens pequenas, o EME-Ray não se mostrou vantajoso em memória. No entanto, em todos os tamanhos de imagem considerados tivemos um desempenho superior ao do *ZSweep* considerando-se o tempo total para a renderização. Para processar uma imagem 1024x1024 nosso método (EME-Ray) mostrou-se 1,6 vezes mais rápido que o *ZSweep*, utilizando 1,5 vezes menos memória. A Figura 24 mostra uma imagem 512x512 desse conjunto de dados, gerada por um de nossos métodos.

Tabela 9. Uso de tempo e memória do conjunto de dados *Delta Wing*.

Comparação de Performance no Conjunto de Dados *Delta Wing*				
Tamanho da Imagem	**Método**	**Número de *Pixels***	**Tempo (s)**	**Memória (MB)**
128^2	BUNYK *et al*.	4.532	0,46	342,00
	ZSweep	4.533	4,25	65,79
	ME-Ray	4.533	0,65	132,00
	EME-Ray	4.533	0,69	81,81
256^2	BUNYK *et al*.	18.313	1,11	343,00
	ZSweep	18.315	6,87	69,81
	ME-Ray	18.315	1,80	151,00
	EME-Ray	18.315	2,26	82,66
512^2	BUNYK *et al*.	73.447	3,49	347,00
	ZSweep	73.454	15,10	85,81
	ME-Ray	73.454	4,88	180,00
	EME-Ray	73.454	8,54	86,04
1024^2	BUNYK *et al*.	294.390	12,06	362,00
	ZSweep	294.407	53,53	146,00
	ME-Ray	294.407	17,85	222,00
	EME-Ray	294.407	32,75	99,54

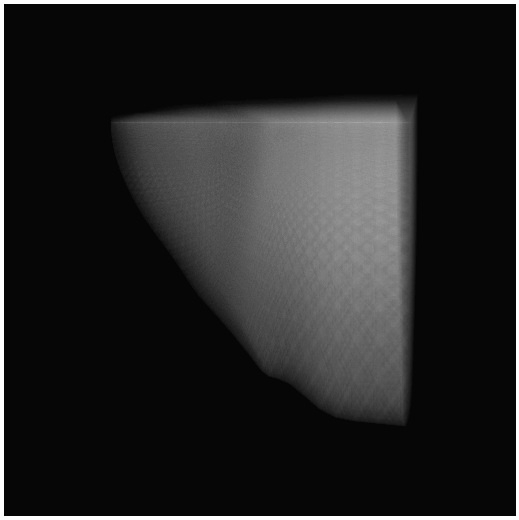


Figura 24. Imagem do *Delta Wing* criada em 512x512.

A Tabela 10 apresenta a comparação no conjunto de dados *SPX*. Para esse conjunto de dados, nosso método ME-Ray, apesar de ser mais lento que o método de BUNYK *et al*., usa menos memória em todos os tamanho de imagem considerados. Note que, para renderizar uma imagem de tamanho 1024x1024, o ME-Ray usa menos memória do que é necessário para que o método de BUNYK *et al*. gere uma imagem de tamanho 128x128. Nesse conjunto de dados, uma imagem 512x512 gerada pelo método de BUNYK *et al*. chega a apresentar 76 *pixels* defeituosos. Já, em comparação com o método *ZSweep*, nosso ME-Ray, mostrou-se mais eficiente em tempo em todos os tamanhos de imagem considerados. Para gerar uma imagem 1024x1024, nosso método é cerca 2,4 vezes mais rápido que o *ZSweep*. O método EME-Ray, para gerar uma imagem 512x512 usa cerca de 1/4 da memória necessária em BUNYK *et al*. para gerar uma imagem 256X256. Para gerar uma imagem 1024x1024, nosso método EME-Ray é 1,3 vezes mais rápido e usa quase 1,2 vezes menos memória que o *ZSweep*. A Figura 25 mostra uma imagem 512x512 desse conjunto de dados, gerada por um de nossos métodos.

Tabela 10. Uso de tempo e memória do conjunto de dados *SPX*.

Comparação de Performance no Conjunto de Dados *SPX*				
Tamanho da Imagem	**Método**	**Número de *Pixels***	**Tempo (s)**	**Memória (MB)**
128^2	BUNYK *et al*.	2.478	0,48	282,00
	ZSweep	2.478	2,81	53,14
	ME-Ray	2.478	0,58	145,00
	EME-Ray	2.478	0,95	66,91
256^2	BUNYK *et al*.	9.944	1,22	282,00
	ZSweep	9.947	4,52	55,51
	ME-Ray	9.947	1,89	194,00
	EME-Ray	9.947	2,18	67,70
512^2	BUNYK *et al*.	39.713	3,66	286,00
	ZSweep	39.718	10,64	64,96
	ME-Ray	39.718	5,00	208,00
	EME-Ray	39.718	7,04	70,84
1024^2	BUNYK *et al*.	159.393	12,45	299,00
	ZSweep	159.411	34,95	100,00
	ME-Ray	159.411	14,47	222,00
	EME-Ray	159.411	27,64	83,40

Figura 25. Imagem do *SPX* criada em 512x512.

Como nossos métodos também podem processar modelos com células hexaedrais e com células tetraedrais e hexaedrais ao mesmo tempo, as Figura 26 e 27 mostram as imagens geradas por um de nossos métodos dos conjuntos de dados *Hexahedral*, formado apenas por células hexaedrais, e *HexaSPX*, formado por células tetraedrais e hexaedrais.

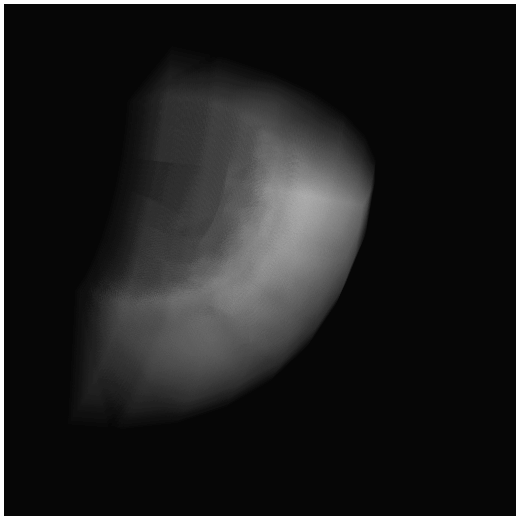


Figura 26. Imagem do *Hexahedral* criada em 512x512.

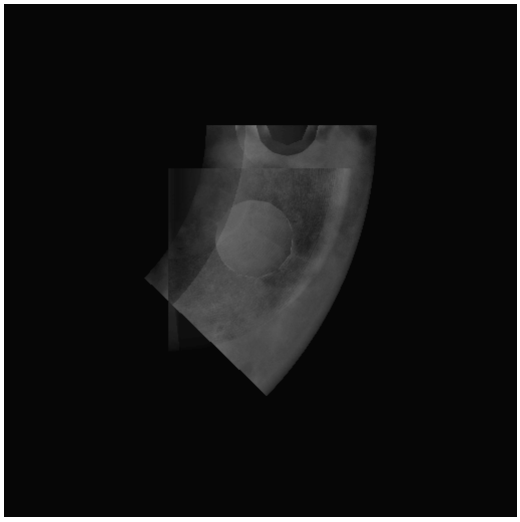


Figura 27. Imagem do *HexaSPX* criada em 512x512.

Se nossos métodos não estivessem habilitados a funcionar para células tetraedrais e hexaedrais, necessitaríamos de cerca de (4*C) bytes a menos de memória para o processamento, pois para cada célula guardamos o seu tipo (tetraedral ou hexaedral), onde C é o número de células contidas no conjunto de dados.

É importante ressaltar, também, que os parâmetros de cálculo da interseção do raio com as faces das células em nossos métodos são do tipo *double* e no método de BUNYK *et al*. são do tipo *float*. Variáveis do tipo *float*, nesse caso, podem retornar resultados não muito precisos, o que, de fato, aconteceu. Testes realizados provaram que, aumentando a precisão desses parâmetros na implementação desenvolvida por BUNYK *et al*., menos *pixels* defeituosos são encontrados na imagem renderizada. No entanto, se aumentarmos a precisão desses parâmetros, o algoritmo de BUNYK *et al*. se tornará ainda mais custoso em memória, tendo um acréscimo de cerca de (12×F) bytes na memória total, onde F é o número total de faces contidas no conjunto de dados volumétrico. Para o conjunto de dados *Delta Wing*, por exemplo, que possui mais de 2 milhões de faces, teríamos um acréscimo de mais de 24 Mb no consumo de memória.

5. CONCLUSÕES

Neste trabalho apresentamos duas implementações do algoritmo de ray-casting, ME-Ray (Memory Efficient Ray-Casting) e EME-Ray (Enhanced Memory Efficient Ray-Casting), para conjuntos de dados representados na forma de grades irregulares. Esses métodos são otimizações em uso de memória e fazem uso apenas de uma estrutura auxiliar para representar as adjacências. As estruturas de dados que utilizamos foram baseadas nas estruturas de dados usadas no *ZSweep* (FARIAS *et al*., 2000).

O método ME-Ray foi desenvolvido com o objetivo de ser uma implementação do algoritmo de ray-casting competitiva com a implementação de BUNYK *et al*. tanto em uso de memória quanto em tempo de renderização. O EME-Ray foi desenvolvido com o intuito de otimizar o método ME-Ray em custo de memória.

As estruturas de dados utilizadas em nossos métodos só armazenam informações essenciais para que o caminho através do conjunto de dados seja feito rapidamente. A lista *Use_set* para cada vértice e o *array Neighbour* para cada célula são criados em fase de pré-processamento e facilitam muito o processo de ray-casting. A lista *Use_set* de cada vértice (igual à usada no *ZSweep*), que nos fornece uma lista de todas as células incidentes no vértice, é utilizada em dois momentos: quando procuramos as células vizinhas durante a fase de pré-processamento (criando o *array Neighbour*) e quando procuramos a próxima face intersectada pelo raio lançado através do *pixel* se ocorrer um caso degenerado. A lista *referredBy* de cada vértice utilizada por BUNYK *et al*. pode ser vista como um *Use_set* que fornece as faces incidentes no vértice e não as células. No entanto, essa lista no método de BUNYK *et al*. é apenas utilizada para impedir o armazenamento de faces iguais na lista de faces. O *array Neighbour* da célula guarda os

índices das células vizinhas da célula. Guardando estes índices das células vizinhas, evitamos que a próxima célula seja procurada em todo o *Use_set* de cada um dos vértices da célula atual cada vez que um raio lançado por um *pixel* atravessa esta célula. Isso acelera muito o processo de ray-casting.

Duas estruturas de dados, que se encontram no ME-Ray e não no EME-Ray, são responsáveis por uma aceleração significativa no processo de ray-casting. Estas estruturas são o *array* de faces (*array Faces_VEC*) e o *array Triangular_Faces* da célula. O *array Faces_VEC*, por ser criado durante o processo de ray-casting, apenas armazena as faces percebidas por algum raio lançado através de um *pixel*. Assim, se um conjunto de dados possui células muito pequenas de forma que nenhum raio lançado através de um *pixel*, para um determinado tamanho de imagem, perceba a presença dessas células, as faces dessas células não serão armazenadas no *array Faces_VEC*. No entanto, no método de BUNYK *et al*. todas as faces do conjunto de dados são armazenadas. No modelo *Delta Wing*, por exemplo, o método de BUNYK *et al*. armazena 2.032.084 faces e o nosso método ME-Ray armazena apenas 354.837 faces, para renderizar uma imagem de tamanho 128x128, consumindo uma quantidade significativamente menor de memória. O *array Triangular_Faces* da célula, por sua vez, serve para acessarmos as informações sobre as faces da célula, no *array Faces_VEC*, em $O(1)$.

Os métodos ME-Ray e EME-Ray, portanto, alcançaram os objetivos desejados. O método ME-Ray mostrou ser uma aplicação rápida e robusta, usando menos memória que o método de BUNYK *et al*.. O método EME-Ray, além de usar menos memória que o ME-Ray, mostrou-se vantajoso em consumo de memória em relação ao *ZSweep* na geração de imagens muito grandes. Apesar de utilizar, em alguns casos, cerca de 1/4 da

memória utilizada pelo método de BUNYK *et al*., o EME-Ray é mais lento. No entanto, quando necessita-se de um algoritmo que utilize pouca memória, sem importar-se com o tempo de renderização da imagem, o EME-Ray é uma boa opção.

Mesmo que não consideremos o uso de memória e o tempo de execução das implementações, nossos métodos são superiores ao desenvolvido por BUNYK *et al*. porque, além de nossos códigos poderem processar modelos com células tetraedrais e/ou hexaedrais, também possuem um tratamento completo para casos degenerados. Esse tratamento para casos degenerados é o principal motivo porque as imagens geradas por nossos métodos não apresentam falhas como as geradas pelo método de BUNYK *et al*..

REFERÊNCIAS BIBLIOGRÁFICAS

- BLINN, J. F., 1982, "Light Reflection Functions For Simulation Of Clouds And Dusty Surfaces", *In Proceedings of SIGGRAPH '92*, v. 16, n. 3, pp. 21-29, Boston, Julho.

- BUNYK, P., KAUFMAN, A., SILVA, C. T., 1999, "Simple, Fast, And Robust Ray Casting Of Irregular Grades", In G. Nielson H. Hagen and F. Post, editors, *Scientific Visualization*. IEEE Computer Society Press.

- DREBIN, R. A., CARPENTER, L., HANRAHAN, P., 1988, "Volume rendering", *Computer Graphics (SIGGRAPH '88 Proceedings)*, v. 22, n. 4, pp. 65–74, Atlanta.

- FARIAS, R., 2001, *Efficient Rendering Of Volumetric Irregular* Grades *Data*, Ph.D. Thesis, State University of New York at Stony Brook.

- FARIAS, R., MITCHELL, J., SILVA, C., "ZSWEEP: An Efficient And Exact Projection Algorithm For Unstructured Volume Rendering", *In 2000 Volume Visualization Symposium*, pp. 91-99, Outubro.

- GARRITY, M. P., 1990, "Raytracing Irregular Volume Data", *Computer Graphics*, v. 24, n. 5, pp. 35-40.

- HE, T., HONG, L., KAUFMAN, A., PFISTER, H., 1996, "Generation Of Transfer Functions With Stochastic Search Techniques", *In Proceedings IEEE Visualization*, pp. 227-234, Outubro.

- HONG, L., KAUFMAN, A., 1998, "Accelerated Ray-Casting For Curvilinear Volumes", *IEEE Visualization '98*, pp. 247-254, Outubro.

- HONG, L., KAUFMAN, A., 1999, "Fast Projection-Based Ray-Casting Algorithm For Rendering Curvilinear Volumes", *IEEE Transactions on Visualization and Computer Graphics*, v. 5, n. 4, pp. 322-332.

- KAJIYA, J. T., VON HERZEN, B. P., 1994, "Ray Tracing Volume Densities", *In Proceeding SIGGRAPH'94*, v. 18, n. 3, pp. 165-174, Julho.

- KOYAMADA, K., 1992, "Fast Traversal Of Irregular Volumes", In T. L. Kunii, editor, *Visual Computing - Integrating Computer Graphics and Computer Vision*, Springer Verlag, pp. 295-312.

- KRUEGER, W., 1991, "The Application Of Transport Theory To Visualization Of 3D Scalar Data Fields", *Computers in Physics*, v. 5, n. 4, pp. 397-406, Novembro.

- LACROUTE, P., 1995, *Fast Volume Rendering Using A Shear-Warp Factorization Of The Viewing Transformation*, Technical Report CSL-TR-95-678, Stanford University, Stanford, CA.

- LACROUTE, P., LEVOY, M., 1994, "Fast Volume Rendering Using A Shear-Warp Factorization Of The Viewing Transformation", *Computer Graphics*, v. 28, n. 4, pp. 451-458, Agosto.

- LEVOY, M., 1988, "Display of Surfaces from Volume Data", *IEEE Computer Graphics and Applications*, v. 5, n. 3, pp. 29-37, Maio.

- LEVOY, M., 1989, *Display of Surfaces from Volume Data*, Ph.D. Thesis, University of North Carolina at Chapel Hill.

- LEVOY, M., 1990a, "Ray Tracing Of Volume Data", *ACM SIGGRAPH '90, Course Notes, Volume Visualization Algorithms and Archtectures*, pp. 120-147, Agosto.

- LEVOY, M., 1990b, "Efficient Ray Tracing of Volume Data", *IEEE Computer Graphics (Proceedings of SIGGRAPH '90)*, pp. 157-167.

- LICHTENBELT, B., CRANE, R., NAQVI, S., 1998, *Introduction To Volume Rendering*. New Jersey, Prentice Hall.

- MA, K.-L., 1995, "Parallel Volume Ray-Casting For Unstructured Grade Data On Distributed-Memory Architectures", *IEEE Parallel Rendering Symposium*, pp. 23-30, Outubro.

- MA, K.-L., PAINTER, J. S., 1993, "Parallel Volume Visualization On Workstations", *Computers and Graphics*, v. 17, n. 1, pp. 31-37.

- MARKS, J., ANDALMAN, B., BEARDSLEY, P. A., FREEMAN, W., GIBSON, S., HODGINS, J., KANG, T., MIRTICH, B., PFISTER, H., RUML, W., RYALL, K., SEIMS, J., SHIEBER, S., 1997, "Design Galleries: A General Approach To Setting Parameters For Computer Graphics And Animation", *In Proceedings of SIGGRAPH '97*, pp. 389-400, New York, Agosto.

- MAX, N., 1995, "Optical Models For Direct Volume Rendering", *IEEE Transactions on Visualization and Computer Graphics*, v. 1, n. 2, pp. 99-108, Junho.

- NOVINS, K. L., SILLION, F. X., GREENBERG, D. P., 1990, "An Efficient Method For Volume Rendering Using Perspective Projection", *Computer Graphics*, v. 24, n. 5, pp. 95-102, Novembro.

- PAIVA, A. C., SEIXAS, R. B., GATTASS, M., 1999, *Introdução à Visualização Cientifica*, Monografia em Ciência da Computação, no 3/99, Departamento de Informática, PUC-Rio, 1999.

- SABELLA, P., 1988, "A Rendering Algorithm For Visualizing 3D Scalar Fields", *Computer Graphics*, v. 22, n. 4, pp. 51-58, Agosto.

- SHIRLEY, P., TUCHMAN, A., 1990, "A Polygonal Approximation To Direct Scalar Volume Rendering", *Computer Graphics (San Diego Workshop on Volume Visualization)*, v. 24, n. 5, pp. 63-70.

- UPSON, C., KEELER, M., 1988, "V-Buffer: Visible Volume Rendering", *Computer Graphics (SIGGRAPH '88 Proceedings)*, v. 22, n. 4, pp.59-64, Atlanta, Agosto.

- USELTON, S, 1991, *Volume Rendering For Computational Fluid Dynamics: Initial Results*, Technical Report RNR-91-026, NAS-NASA Ames Research Center, Moffett Field, CA.

- WESTOVER, L., 1990, "Footprint Evaluation For Volume Rendering", *Computer Graphics (SIGGRAPH '90 Proceedings),* v. 24, n. 4, pp. 367-376, Dallas.

- WILHELMS, J., CHALLINGER, J., ALPER, N., RAMAMOORTHY, S., VAZIRI, A., 1990, "Direct Volume Rendering Of Curvilinear Volumes", *Computer Graphics (San Diego Workshop on Volume Visualization)*, v. 24, n. 5, pp. 41-47.