Pontifícia Universidade Católica
do Rio de Janeiro

Rodrigo de Souza Lima Espinha

Visualização Volumétrica Interativa de Malhas Não-Estruturadas Utilizando Placas Gráficas Programáveis

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do título de Mestre pelo Programa de Pós-Graduação em Informática da PUC-Rio.

Orientador: Prof. Waldemar Celes Filho

Rio de Janeiro, março de 2005

Pontifícia Universidade Católica
do Rio de Janeiro

Rodrigo de Souza Lima Espinha

Visualização Volumétrica Interativa de Malhas Não-Estruturadas Utilizando Placas Gráficas Programáveis

Dissertação apresentada como requisito parcial para obtenção do título de Mestre pelo Programa de Pós-Graduação em Informática da PUC-Rio. Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Waldemar Celes Filho
Orientador
Departamento de Informática - PUC-Rio

Prof. Marcelo Gattass
Departamento de Informática - PUC-Rio

Prof. Paulo Cezar Pinto Carvalho
Instituto Nacional de Matemática Pura e Aplicada (IMPA)

Prof. Luiz Fernando Martha
Departamento de Engenharia Civil - PUC-Rio

Prof. José Eugênio Leal
Coordenador Setorial do Centro Técnico Científico - PUC-Rio

Rio de Janeiro, 18 de março de 2005

Rodrigo de Souza Lima Espinha

Graduou-se em Engenharia de Computação pela Pontifícia Universidade Católica do Rio de Janeiro, onde continuou seus estudos no programa de Mestrado em Informática. Durante a permanência nesta instituição, atuou em projetos voltados para a indústria do petróleo, no laboratório Tecgraf.

Ficha Catalográfica

Espinha, Rodrigo de Souza Lima

Visualização volumétrica interativa de malhas não-estruturadas utilizando placas gráficas programáveisl / Rodrigo de Souza Lima Espinha; orientador: Waldemar Celes Filho. – Rio de Janeiro: PUC, Departamento de Informática, 2005.

86 f. ; 30 cm

Dissertação (mestrado) – Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui referências bibliográficas.

1. Informática – Teses. 2. Visualização volumétrica. 3. Programação em placas gráficas. 4. Malhas não-estruturadas. 5. Visualização interativa. 6. Função de transferência. I. Celes Filho, Waldemar. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Aos meus pais, meu irmão e minha namorada.

Agradecimentos

A Deus, pois sem Ele nada teria sido possível.

À minha família e minha namorada, Herisangela, pelo incentivo em todos os momentos.

Ao meu orientador, professor Waldemar Celes, pelo apoio, motivação e o conhecimento que me foi transmitido ao longo do curso.

À CAPES, Tecgraf e PUC-Rio, pelo financiamento e oportunidade de trabalhar e estudar em um lugar onde posso aprender todos os dias.

Aos meus colegas e amigos, especialmente Frederico Abraham (Fred), Antonio Calomeni, Luiz Gustavo, Márcio Pereira, Rodrigo Hermann, Michel e Manuel (peruanos), que também percorreram esse mesmo caminho ao longo dos últimos dois anos.

A Antonio Miranda e Ivan Menezes, do Tecgraf, pela paciência e pelas malhas de elementos finitos utilizadas neste trabalho.

A todos do Tecgraf e da PUC-Rio.

Resumo

Espinha, Rodrigo de Souza Lima. **Visualização Volumétrica Interativa de Malhas Não-Estruturadas Utilizando Placas Gráficas Programáveis.** Rio de Janeiro, 2005. 86p. Dissertação de Mestrado - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

A visualização volumétrica é uma importante técnica para a exploração de dados tridimensionais complexos, como, por exemplo, o resultado de análises numéricas usando o método dos elementos finitos. A aplicação eficiente dessa técnica a malhas não-estruturadas tem sido uma importante área de pesquisa nos últimos anos. Há dois métodos básicos para a visualização dos dados volumétricos: extração de superfícies e renderização direta de volumes. Na primeira, iso-superfícies de um campo escalar são extraídas explicitamente. Na segunda, que é a utilizada neste trabalho, dados escalares são classificados a partir de uma função de transferência, que mapeia valores do campo escalar em cor e opacidade, para serem visualizados. Com a evolução das placas gráficas (GPU) dos computadores pessoais, foram desenvolvidas novas técnicas para visualização volumétrica interativa de malhas não-estruturadas. Os novos algoritmos tiram proveito da aceleração e da possibilidade de programação dessas placas, cujo poder de processamento cresce a um ritmo superior ao dos processadores convencionais (CPU). Este trabalho avalia e compara dois algoritmos para visualização volumétrica de malhas não-estruturadas, baseados em GPU: projeção de células independente do observador e traçado de raios. Adicionalmente, são propostas duas adaptações dos algoritmos estudados. Para o algoritmo de projeção de células, propõe-se uma estruturação dos dados na GPU para eliminar o alto custo de transferência de dados para a placa gráfica. Para o algoritmo de traçado de raios, propõe-se fazer a integração da função de transferência na GPU, melhorando a qualidade da imagem final obtida e permitindo a alteração da função de transferência de maneira interativa.

Palavras-chave

visualização volumétrica, programação em placas gráficas, malhas não-estruturadas, visualização interativa, função de transferência

Abstract

Espinha, Rodrigo de Souza Lima;. **Interactive Volume Visualization of Unstructured Meshes Using Programmable Graphics Cards.** Rio de Janeiro, 2005. 86p. MSc. Dissertation - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Volume visualization is an important technique for the exploration of three-dimensional complex data sets, such as the results of numerical analysis using the finite elements method. The efficient application of this technique to unstructured meshes has been an important area of research in the past few years. There are two basic methods to visualize volumetric data: surface extraction and direct volume rendering. In the first, the iso-surfaces of the scalar field are explicitly extracted. In the second, which is the one used in this work, scalar data are classified by a transfer function, which maps the scalar values to color and opacity, to be visualized. With the evolution of personal computer graphics cards (GPU), new techniques for volume visualization have been developed. The new algorithms take advantage of modern programmable graphics cards, whose processing power increases at a faster rate than the one observed in conventional processors (CPU). This work evaluates and compares two GPU-based algorithms for volume visualization of unstructured meshes: view-independent cell projection (VICP) and ray-tracing. In addition, two adaptations of the studied algorithms are proposed. For the cell projection algorithm, we propose a GPU data structure in order to eliminate the high costs of the CPU to GPU data transfer. For the ray-tracing algorithm, we propose to integrate the transfer function in the GPU, which increases the quality of the generated image and allows to interactively change the transfer function.

Keywords

volume visualization, GPU programming, unstructured meshes, interactive visualization, transfer function.

Sumário

Lista de figuras

Lista de tabelas

1
Introdução

O termo *visualização* corresponde, no contexto desta dissertação, aos métodos que permitem a extração de informações relevantes a partir de conjuntos de dados complexos, com o auxílio de técnicas de computação gráfica (Paiva et al., 1999), constituindo-se como uma ferramenta para a interpretação e análise de dados computacionais. A área de *visualização científica* se preocupa com a extração de informações de caráter científico a partir de conjuntos de dados que representam fenômenos complexos.

Os dados associados a regiões de volumes (informações tridimensionais) são chamados *dados volumétricos*. Esses dados são geralmente amostras de uma função de difícil reconstrução, que associa valores às posições do espaço. Os valores nas posições intermediárias podem ser obtidos pela interpolação entre os valores vizinhos. O conjunto de técnicas voltadas para a visualização de dados volumétricos é conhecido como *visualização volumétrica* (Drebin et al., 1988), e constitui uma subárea da visualização científica.

Muitas vezes, os dados volumétricos estão dispostos topologicamente como uma grade (ou malha) retilínea composta por células hexaédricas (Figura 1), que pode ser representada como uma matriz de índices (i, j, k). Uma característica dessa representação é que ela apresenta uma estrutura de conectividade implícita entre os elementos (células hexaédricas) de que é composta, permitindo que todos os vizinhos de uma célula sejam obtidos pelo incremento (ou decremento) do índice da célula atual. Essa característica define o que é chamado malha *estruturada* (Speray, 1990; Yagel, 1996). As malhas que não apresentam essa característica são chamadas malhas *não-estruturadas* e as suas células podem ser poliedros arbitrários, que por sua vez podem ser decompostos em tetraedros. Dessa forma, um modelo arbitrário pode sempre ser representado por uma malha não-estruturada de tetraedros. As malhas estruturadas também podem ser incluídas como um caso particular de malhas não-estruturadas. Os dados representados em uma malha são, geralmente, associados às suas células ou aos

vértices dessas células. Exemplos de malhas estruturadas e não-estruturadas são apresentados na Figura 2.

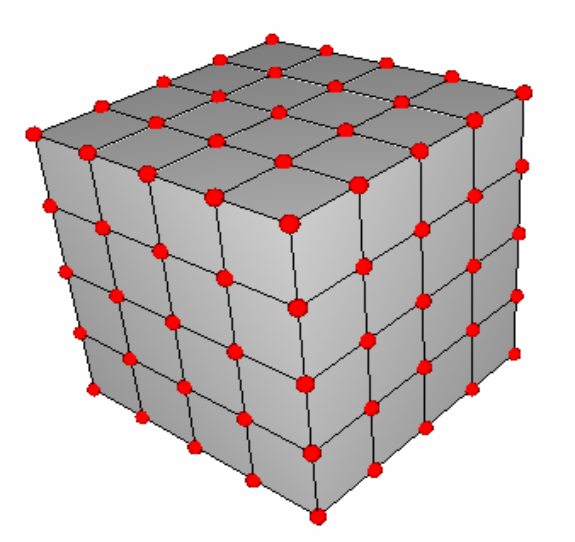


Figura 1 – Dados volumétricos dispostos como uma grade tridimensional.

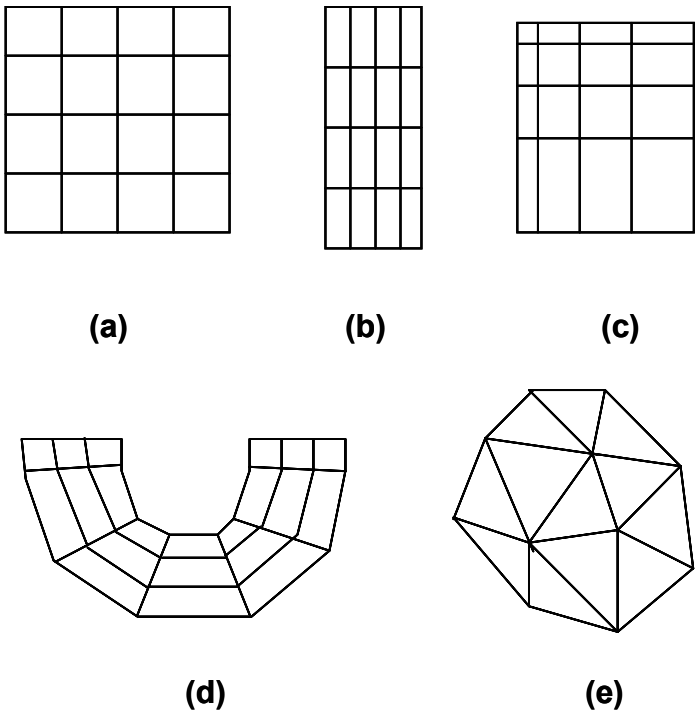
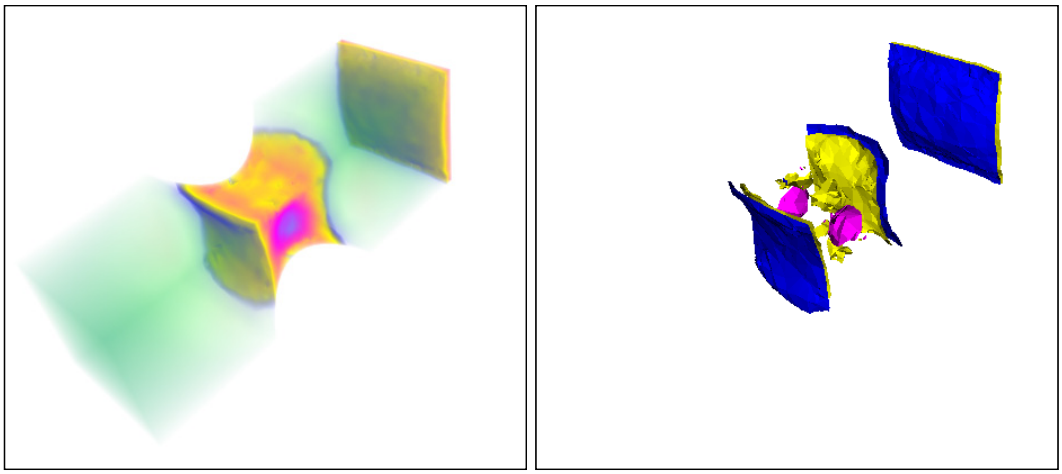


Figura 2 – Exemplos de malhas tridimensionais projetadas. As malhas são apresentadas hierarquicamente, da mais restrita à mais geral: (a) grade cartesiana ou uniforme, (b) grade regular, (c) grade retilínea, (d) malha estruturada, (e) malha não-estruturada de tetraedros.

Há duas abordagens básicas (Paiva et al., 1999) para realizar a visualização volumétrica: extração de superfícies (Figura 3b) e *renderização*[1] direta de volumes (Figura 3a). Na primeira, são construídas representações poligonais de superfícies relacionadas às características desejadas da função de interesse (p. ex., iso-superfícies), que são, em geral, visualizadas utilizando técnicas de renderização de polígonos, suportadas diretamente pelas placas gráficas atuais. Na segunda abordagem, que será a utilizada nesta dissertação, os dados são diretamente visualizados, sem a extração explícita de superfícies.



(a) **(b)**

Figura 3 – Visualização volumétrica dos resultados da análise de elementos finitos para uma malha de tetraedros, usando duas técnicas: (a) renderização direta de volumes, (b) extração de iso-superfícies.

As etapas básicas para a visualização das malhas não-estruturadas utilizando renderização direta de volumes são ilustradas na Figura 4. A *classificação*, que é discutida no Capítulo 2, determina os dados que devem ser visualizados e os separa de acordo com as suas características. Isto é normalmente feito por uma *função de transferência*, que associa valores de cor e opacidade aos dados volumétricos. Após a classificação os dados são projetados para gerar uma imagem final, e, assim, poderem ser visualizados.

[1] No contexto desta dissertação, o termo "*renderização*" está relacionado ao processo de geração de imagens a partir de um conjunto de dados que descrevem a geometria e os atributos necessários para sua visualização (p. ex., coordenadas e cores de vértices de uma primitiva geométrica).

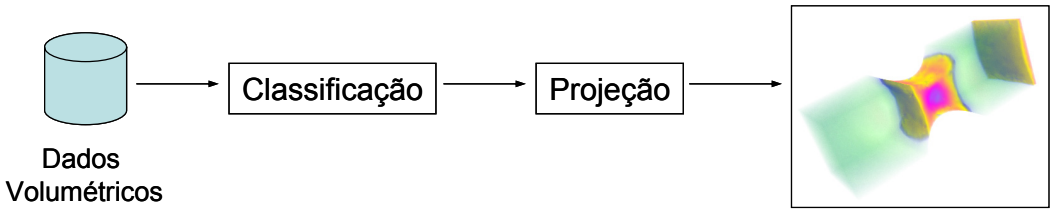


Figura 4 – Etapas da renderização direta de volumes.

Nos últimos anos, a evolução da placas gráficas utilizadas nos computadores pessoais (*PC*) permitiu a visualização interativa de diversos tipos de informações, sem a necessidade de estações gráficas de custo elevado. Uma das inovações mais importantes (e recentes) foi o surgimento das placas com unidades de processamento gráfico (*GPU – Graphics Processor Unit*) programáveis, o que aumentou dramaticamente suas possibilidades de utilização. Apesar de desenvolvidas principalmente para o mercado de jogos eletrônicos, a utilização dessas placas em outras áreas, como a visualização volumétrica, tem sido objeto de diversas pesquisas.

As GPUs modernas (Comba et al., 2003) são otimizadas para o processamento de dados vetoriais, permitindo executar simultaneamente uma operação para cada uma das componentes de um vetor de dimensão 4. Diversas operações aritméticas (p. ex., seno, exponencial, produto interno) comumente utilizadas são eficientemente implementadas com apenas uma instrução em liguagem de máquina. Os dados enviados para a placa gráfica são os vértices de primitivas poligonais (geralmente pontos, linhas ou triângulos) e alguns atributos adicionais, como cor e coordenadas de textura. Os vértices são individualmente processados, na etapa de *geometria*, e enviados para a *rasterização*, que é responsável por preencher cada posição (*pixel*) da tela de projeção pertencente à primitiva. As placas gráficas programáveis permitem intervir nessas duas etapas, por meio de um *programa por vértice* e um *programa por fragmento* (OpenGL ARB, 2005; NVIDIA, 2004a) O primeiro oferece a possibilidade de alterar as características de cada vértice enviado para a placa, enquanto o segundo permite modificar os atributos de um pixel antes que ele seja desenhado. Um pixel nesta condição é chamado de *fragmento*.

1.1. Motivação

A motivação para esta pesquisa foi o desejo de investigar técnicas de visualização volumétrica para serem adicionadas ao Pos3D (Celes, 1990), um programa para pós-processamento de resultados da análise de modelos de elementos finitos tridimensionais, desenvolvido pelo laboratório de Tecnologia em Computação Gráfica (Tecgraf) da PUC-Rio, para a Petrobras.

Uma malha de elementos finitos corresponde à discretização de um domínio geométrico contínuo. A discretização equivale a uma decomposição celular do domínio no qual o interior de cada célula, que é chamada *elemento*, representa uma região disjunta do espaço e é definida por um conjunto de *nós*. Normalmente, um elemento finito é um poliedro (Figura 5) cujos vértices são nós do elemento, e cada nó possui associado um valor de propriedade. O valor da propriedade em qualquer posição no interior do elemento é determinado pela interpolação, linear ou não, dos valores dos nós. Elementos não-lineares possuem nós que não são vértices e podem apresentar faces não-planares (Figura 5c). No caso geral, as malhas de elementos finitos podem ser consideradas como malhas não-estruturadas.

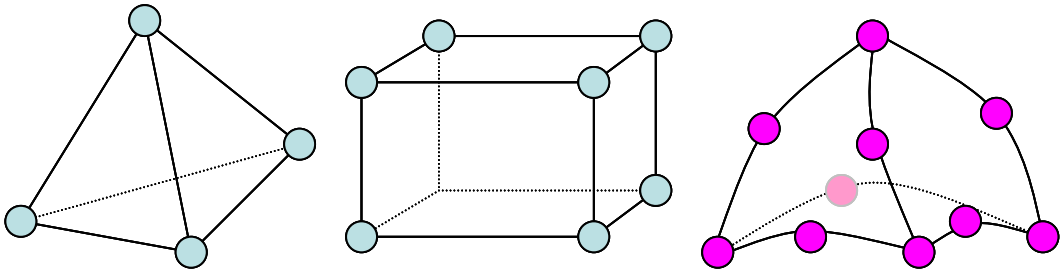


Figura 5 – Exemplos dos principais elementos finitos: (a) tetraedro linear (4 nós), (b) hexaedro linear (8 nós), (c) tetraedro quadrático (10 nós).

A aplicação das técnicas de visualização volumétrica ao Pos3D deve atender a alguns requisitos, como:

- visualizar interativamente malhas em domínios não-convexos, formadas pelos principais tipos de elementos (tetraedros, hexaedros e prismas);
- permitir a modificação interativa da função de transferência;
- permitir cortes volumétricos da malha.

Esta pesquisa se restringe às malhas de tetraedros lineares (faces planas e interpolação linear), nas quais a propriedade visualizada consiste em amostras de um campo escalar.

1.2. Objetivo

O objetivo principal desta pesquisa é investigar a visualização volumétrica interativa de malhas não-estruturadas. Para isso, são estudados e comparados os principais algoritmos que exploram os recursos de programação das placas gráficas modernas. Adicionalmente, são propostas algumas modificações sobre os algoritmos originais. As malhas não-estruturadas enfocadas neste trabalho são as de elementos finitos, com os resultados podendo ser estendidos a outras de características semelhantes, como, por exemplo, malhas de diferenças finitas usadas em simulações de reservatórios naturais de petróleo.

As principais contribuições desta pesquisa são:

- estudo e comparação dos algoritmos de projeção de células e traçado de raios para visualização volumétrica em placas gráficas programáveis;
- proposta de uma nova estruturação dos dados na placa gráfica, para eliminar o gargalo da transferência de dados do algoritmo de projeção de células;
- proposta de aplicar a integração da função de transferência na GPU ao algoritmo de traçado de raios.

O Capítulo 2 discute a classificação de dados volumétricos, fundamental para a renderização direta de volumes. A primeira seção trata do papel da função de transferência neste processo e a segunda apresenta os modelos ópticos mais utilizados, além dos métodos para sua aplicação à visualização de malhas não-estruturadas.

O Capítulo 3 apresenta algoritmos para a visualização interativa dos dados volumétricos. Dois paradigmas são discutidos: *projeção de células* e *traçado de raios*, ambos acelerados pela placa gráfica.

No Capítulo 4, são apresentadas alternativas a alguns aspectos dos algoritmos apresentados no Capítulo 3, além de uma avaliação conceitual desses algoritmos. Adicionalmente, duas modificações são propostas. A primeira (seção 4.1.1) é uma estruturação dos dados para a projeção de células, armazenando os dados na placa gráfica. A outra (seção 4.2) é uma forma de aplicar a integração de segmentos lineares de uma função de transferência (seção 2.3.2) ao paradigma de traçado de raios da seção (3.2).

Os resultados obtidos são discutidos no Capítulo 5, e, finalmente, o Capítulo 6 conclui esta pesquisa e sugere possíveis trabalhos futuros.

2
Classificação dos dados volumétricos

Para que a visualização volumétrica possa ser realizada utilizando a renderização direta de volumes, é necessário definir um *modelo óptico* (Max, 1995) que descreva como uma unidade de volume emite, reflete e absorve um raio de luz, ou espraia o raio refletido.

Baseando-se nesse modelo, um campo escalar associado aos dados volumétricos é mapeado para valores de cor e opacidade, por meio de uma *função de transferência*. É a função de transferência a responsável por classificar e selecionar as características desejadas para serem visualizadas. Essa etapa é conhecida como *classificação* do volume. A classificação normalmente é feita atribuindo-se baixa opacidade (alta transparência) aos valores que não são importantes e alta opacidade (baixa transparência) ao que deve ser visualizado. Além disso, são atribuídas diferentes cores para cada intervalo do campo escalar. Por exemplo, supondo que se quer visualizar a variação de temperatura no interior de uma peça mecânica, representada por um modelo de elementos finitos, pode-se atribuir uma escala de cores aos valores de temperatura (p. ex., azul para as baixas temperaturas e vermelho para as altas). Se a variação das temperaturas mais altas for a característica que se deseja visualizar, pode-se atribuir opacidade alta para esses valores e baixa para os outros.

A cor final em cada ponto da imagem gerada no plano de projeção é computada integrando-se as contribuições dos valores de cor e opacidade, mapeados pela função de transferência, para cada unidade de volume ao longo do raio que passa pelo ponto do plano na direção de projeção.

Nas próximas seções são discutidos os principais modelos ópticos utilizados para renderização direta de volumes e as principais abordagens para a utilização da função de transferência na classificação de volumes representados por malhas de tetraedros lineares.

2.1. Função de transferência

A função de transferência é uma função matemática que mapeia os valores de propriedades dos dados volumétricos para propriedades ópticas, como cor e opacidade (ou o coeficiente de extinção, apresentado na próxima seção). Para um campo escalar $s : (x, y, z) \mapsto F(x, y, z)$, utilizando o sistema de cores RGB, tem-se $FT : s \mapsto (R(s), G(s), B(s), \alpha(s))$, onde α representa a opacidade (Figura 6).

Essa função é fundamental para a classificação de volumes, pois é ela que transforma os dados abstratos em características que podem ser visualizadas. A função de transferência pode ser especificada pelo usuário ou gerada por métodos semi-automáticos (Kindlmann & Durkin, 1998), e ser tão complexa quanto se queira. Por isso, a função de transferência é geralmente implementada como uma tabela de valores, que é consultada pela aplicação (*look-up table*) ou convenientemente definida por segmentos lineares (*piecewise linear segments*) (Williams & Max, 1992; Williams et al., 1998).

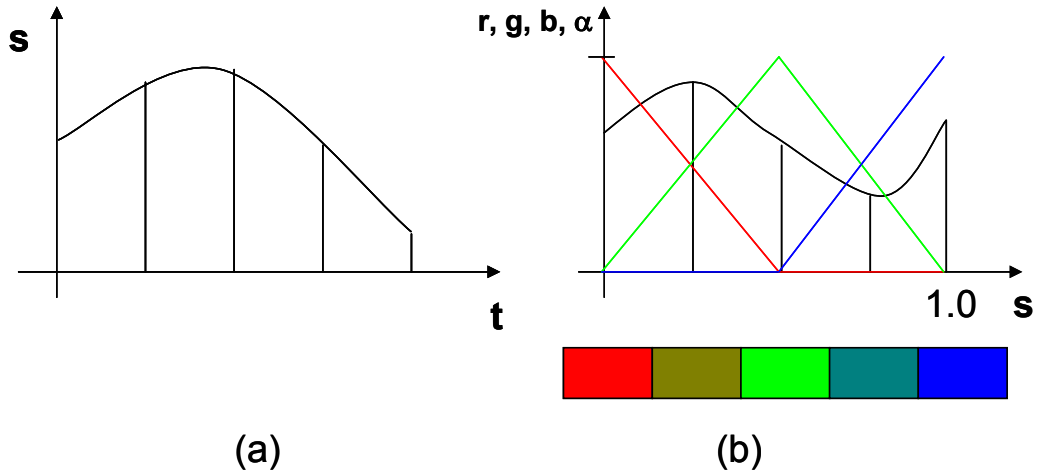


Figura 6 – (a) Valor de um campo escalar ao longo de um raio com parâmetro *t*. (b) Função de transferência definida para valores escalares. Os escalares são mapeados para valores de cor *(r,g,b)* e opacidade α.

É possível também utilizar funções de transferência multidimensionais (Kniss et al., 2002), ou seja, que são definidas em função de mais de uma propriedade relacionada aos dados volumétricos. A cor e a opacidade podem ser definidas, por exemplo, em função de um campo escalar e o gradiente deste

campo: $FT : (s, \vec{\nabla}s) \mapsto (R(s, \vec{\nabla}s), G(s, \vec{\nabla}s), B(s, \vec{\nabla}s), \alpha(s, \vec{\nabla}s))$. A opacidade bidimensional foi recentemente utilizada com sucesso para a visualização de dados sísmicos, em (Silva, 2004), permitindo extrair características importantes desses dados que não poderiam ser visualizadas com uma função de transferência unidimensional.

2.2. Modelos ópticos

Os modelos ópticos utilizados para a geração de imagens com a renderização direta de volumes baseiam-se em modelos físicos criados a partir da simplificação da Teoria de Transporte Radiativo (Krüger, 1991). Assume-se que o meio é formado por partículas que podem emitir, absorver ou refletir raios luminosos, e a intensidade da luz resultante em uma direção é proporcional à densidade de partículas ao longo de um raio que atravessa o volume. Alguns modelos foram inicialmente criados para a visualização de nuvens, como o de (Blinn, 1982), que considera a absorção e a reflexão da luz no meio, assim como a geração de sombras. Entretanto, os principais modelos ópticos para a visualização interativa baseada na renderização direta de volumes consideram que as partículas do meio apenas absorvem e/ou emitem radiação luminosa. Max (1995) examina alguns desses modelos.

O modelo óptico de Williams & Max (1992) é provavelmente o mais utilizado, e será apresentado a seguir, seguindo a abordagem de Max (1995). A idéia desse modelo se baseia na simplificação de modelos realistas de nuvens, reduzindo-os ao mínimo necessário para permitir que a estrutura interna do volume seja visualizada.

Inicialmente, pode-se considerar uma unidade de volume diferencial, com uma densidade ρ de partículas, em que cada partícula absorve toda a intensidade luminosa que a atinge (Figura 7). Todas as partículas são esferas idênticas de área projetada A e o volume tem seção transversal de área E e espessura ds. Dessa forma, o número de partículas é $N = \rho.E.ds$. Assumindo que as projeções de duas partículas na direção de um raio perpendicular à seção transversal não se sobrepõem, então a fração da intensidade luminosa que é obstruída pelas partículas é igual a $N.A/E = \rho.A.ds$. O fator $\tau = \rho.A$ é chamado *coeficiente de*

extinção (ou densidade óptica) e está relacionado à taxa de absorção da luz incidente sobre o volume e varia de acordo com o comprimento de onda da luz. Entretanto, por simplicidade, normalmente se considera que o coeficiente de extinção é o mesmo para todos os comprimentos de onda.

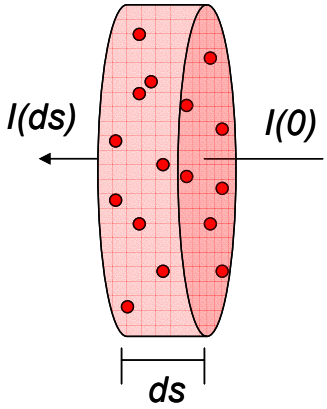


Figura 7 – Volume diferencial composto por partículas que absorvem a intensidade de um raio *I(s)*, ao longo da espessura *ds*.

Isso resulta na seguinte equação diferencial:

$$\frac{dI(s)}{ds} = -I(s).\tau(s), \qquad \textbf{(2.1)}$$

cuja solução é:

$$I(s) = I(0).e^{-\int_0^s \tau(t)dt} \qquad \textbf{(2.2)}$$

A opacidade α de um volume de espessura s é, então, definida como:

$$\alpha = 1 - e^{-\int_0^s \tau(t)dt} \qquad \textbf{(2.3)}$$

É importante observar que a opacidade é dependente da distância percorrida pelo raio, e varia no intervalo [0,1] (objeto totalmente transparente ou opaco, respectivamente), enquanto que o coeficiente de extinção é uma característica diferencial e varia no intervalo [0,∞).

Se, ao invés de absorverem a luz incidente, as partículas apenas emitirem luz, com intensidade C por unidade de sua área projetada, a fração da intensidade gerada pelo volume diferencial será igual a $C.N.A/E = C.\rho.A.ds = C.\tau.ds$, e a equação diferencial resultante será:

$$\frac{dI(s)}{ds} = C(s).\tau(s) \qquad \textbf{(2.4)}$$

Assim como o coeficiente de extinção, a intensidade emitida por cada partícula também varia para os diversos comprimentos de onda. No sistema de cores *RGB (vermelho, verde, azul)* (Foley et al., 1997), poderá haver um valor diferente para cada componente.

Finalmente, se for assumido que cada partícula tanto absorve quanto emite intensidade luminosa, as eq. (2.1) e (2.4) podem ser combinadas, resultando na seguinte equação diferencial:

$$\frac{dI(s)}{ds} = C(s).\tau(s) - I(s).\tau(s) \qquad \textbf{(2.5)}$$

Resolvendo para uma distância D, e um raio de parâmetro t, obtém-se a solução da eq. (2.5), que é conhecida como *integral de renderização de volume*:

$$I(D) = I_0 e^{-\int_0^D \tau(t')dt'} + \int_0^D C(t).\tau(t).e^{-\int_t^D \tau(t')dt'} dt \qquad \textbf{(2.6)}$$

Essa integral composta por uma soma de dois termos pode ser interpretada considerando-se um raio com intensidade inicial I_0, que parte de uma posição atrás do volume visualizado na direção do observador. As posições que não estão no interior do volume não oferecem contribuição, o que implica que as intensidades nessas posições são iguais às dos pontos de entrada e saída do raio. O valor I_0 representa a intensidade de fundo. Essa intensidade é absorvida ao longo da porção do raio que atravessa o volume, o que é representado pelo primeiro termo da eq. (2.6). O segundo pode ser interpretado da seguinte maneira: cada partícula, ao longo do raio, emite intensidade luminosa, que é absorvida pelas partículas do volume que estão à sua frente. A integração das contribuições define a intensidade relativa ao volume, que é, então, adicionada à intensidade de fundo. No sistema RGB, essa integral deve ser resolvida para cada uma das componentes.

Wilhelms & Van Gelder (1991) utilizam o termo $g(t) = C(t).\tau(t)$ como uma propriedade óptica do volume. Assim, duas propriedades devem ser especificadas na função de transferência, $g(t)$ e $\tau(t)$, simplificando a eq. (2.6) para

$$I(D) = I_0 e^{-\int_0^D \tau(t')dt'} + \int_0^D g(t).e^{-\int_t^D \tau(t')dt'} dt \qquad \textbf{(2.7)}$$

Uma desvantagem dessa abordagem é que o aumento do coeficiente de extinção faz com que a imagem fique mais escura, enquanto que no modelo de Williams & Max (1992) há um balanceamento da intensidade emitida pelas partículas pelo coeficiente de extinção.

No caso geral, não se conhece uma solução analítica para a integral de renderização de volume. Porém, em alguns casos específicos, é possível obter uma solução analítica, como nos casos em que as variações do coeficiente de extinção e da intensidade emitida são constantes ou lineares.

Se o coeficiente de extinção e a intensidade emitida forem constantes, a solução para a integral de renderização de volume é a seguinte:

$$I(D) = I_0 e^{-\tau D} + C\int_D^0 -\tau e^{-\tau t}\,dt = I_0 e^{-\tau D} + C(1 - e^{-\tau D}) \qquad \textbf{(2.8)}$$

que equivale a

$$I(D) = I_0(1-\alpha) + C\alpha \qquad \textbf{(2.9)}$$

Dessa forma, a solução da integral corresponde à composição (Porter & Duff, 1984) da cor C, com opacidade α, sobre um fundo de cor I_0. Se o intervalo em que o raio atravessa o volume for dividido em fatias muito pequenas, como intervalos de *somas de Riemann* (Malta et al., 2002), e a cada fatia forem atribuídas cor e opacidade constantes, a integral pode ser resolvida numericamente pela sucessiva composição, de trás para a frente (ou, alternativamente, de frente para trás), das contribuições de cada fatia. A intensidade total ao longo de um raio pode ser, então, aproximada por:

$$I = \sum_{i=0}^{n} \alpha_i C_i \prod_{j=0}^{i-1} (1 - \alpha_j) \qquad \textbf{(2.10)}$$

Assim, uma forma simplificada de resolver a integral consiste em discretizar a função de transferência em pequenos pedaços com cor e opacidade constantes e realizar a composição. Uma característica da eq. (2.9) é que ela é balanceada, ou seja, a cor final nunca será maior que um valor máximo, que, convenientemente, pode ser definido como 1,0.

O termo $C.\alpha$ da eq. (2.9), que resulta da integração do segundo termo da eq. (2.6), é chamado de *cor associada* (Blinn, 1994) e corresponde à cor de um segmento pré-multiplicada pela opacidade. Wittenbrink et al. (1998) demonstram que a interpolação da cor e da opacidade separadamente resulta em erros visuais

(conhecidos como "artefatos"), e Blinn (1994) mostra que a utilização de cores associadas simplifica a composição e permite a associatividade.

Para a variação linear do coeficiente de extinção e da intensidade emitida, a solução analítica (Williams & Max, 1992; Moreland & Angel, 2004) é mais complexa e de maior custo computacional. Para um segmento linear, com $C(t) = C_b.(1-t) + C_f.t$ e $\tau(t) = \tau_b.(1-t) + \tau_f.t$, a solução da integral é:

$$I(D) = \begin{cases} I_0 e^{-D\frac{\tau_b+\tau_f}{2}} + C_f - C_b e^{-D\frac{\tau_b+\tau_f}{2}} + \\ \frac{(C_b - C_f)}{\sqrt{D(\tau_b - \tau_f)}} e^{\frac{D}{2(\tau_b-\tau_f)}\tau_f^2} \sqrt{\frac{\pi}{2}}. \\ \left[erf\left(\tau_b \frac{\sqrt{D}}{\sqrt{2(\tau_b - \tau_f)}} \right) - erf\left(\tau_f \frac{\sqrt{D}}{\sqrt{2(\tau_b - \tau_f)}} \right) \right] \end{cases} \quad \textbf{(tb > tf)}$$

$$I(D) = \begin{cases} I_0 e^{-D\frac{\tau_b+\tau_f}{2}} + C_f - C_b e^{-D\frac{\tau_b+\tau_f}{2}} + \\ \frac{(C_b - C_f)}{\sqrt{D(\tau_f - \tau_b)}} e^{\frac{-D}{2(\tau_f-\tau_b)}\tau_f^2} \sqrt{\frac{\pi}{2}}. \\ \left[erfi\left(\tau_f \frac{\sqrt{D}}{\sqrt{2(\tau_f - \tau_b)}} \right) - erfi\left(\tau_b \frac{\sqrt{D}}{\sqrt{2(\tau_f - \tau_b)}} \right) \right] \end{cases} \quad \textbf{(tb < tf)}$$

$$I(D) = \begin{cases} I_0 e^{-\tau D} + C_b \left(\frac{1}{\tau D} - \frac{e^{-\tau D}}{\tau D} - e^{-\tau D} \right) + \\ C_f \left(1 + \frac{e^{-\tau D}}{\tau D} - \frac{1}{\tau D} \right) \end{cases} \quad \textbf{(tb = tf)}$$

(2.12)

Uma função de transferência genérica poder ser discretizada em segmentos lineares, e a solução da eq. (2.6) aproximada pela composição das contribuições calculadas pela integração analítica desses segmentos. Essa abordagem apresenta uma margem de erro bem menor do que a discretização em segmentos com coeficiente de extinção e intensidade emitida constantes, como ilustrado na Figura 8. A utilização de segmentos lineares representa uma forma conveniente para a

especificação da função de transferência (Williams et al., 1998; Moreland et al., 2004).

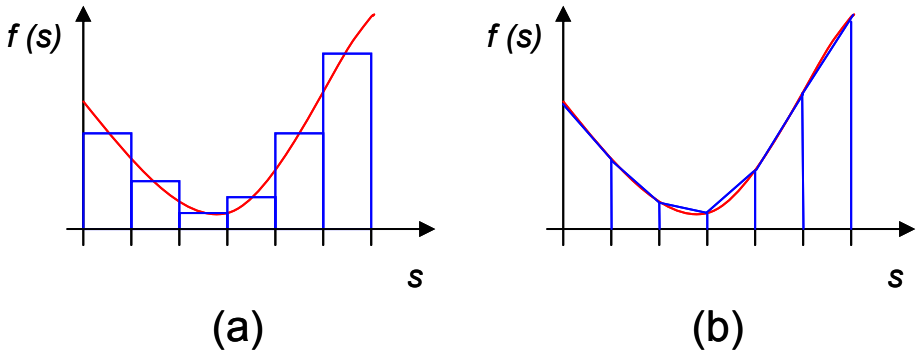


Figura 8 – Aproximação de uma função por segmentos constantes (a) e segmentos lineares (b).

2.3. Aplicação da função de transferência

Uma forma simples e direta de utilizar uma função de transferência arbitrária na visualização volumétrica de malhas de tetraedros consiste em resolver numericamente a eq. (2.6) no momento em que cada tetraedro é desenhado. Podemos convenientemente considerar que a intensidade de fundo I_0 é nula. Assim, a solução da eq. (2.6) resulta em um valor de cor associada e outro de opacidade, que correspondem à contribuição do tetraedro para a geração da imagem final, ao longo de um raio. Isto é equivalente à contribuição de um tetraedro (Figura 9) que possui cor e opacidade constantes, com valores iguais aos da solução obtida. Assim, supondo que os tetraedros são desenhados segundo uma ordenação (de trás para a frente, por exemplo), a eq. (2.9) pode ser utilizada para compor sucessivamente as contribuições de cada um, produzindo a imagem final.

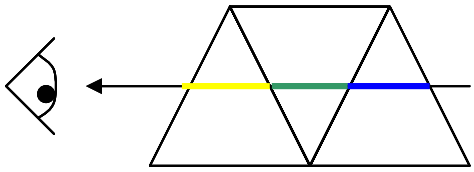


Figura 9 – Composição das contribuições de três tetraedros ao longo de um raio em direção ao observador.

Entretanto, o custo computacional de resolver numericamente a eq. (2.6) para cada posição desenhada é alto, inviabilizando a visualização interativa. Duas abordagens para contornar esse problema são apresentadas nas próximas seções: *pré-integração da função de transferência* e *integração de segmentos lineares*. Cada uma possui algumas vantagens e desvantagens.

2.3.1. Pré-integração

Essa técnica, apresentada por Roettger et al. (2000), consiste na integração da função de transferência em uma etapa de pré-processamento, antes da visualização da malha de tetraedros. A integração resulta em valores de cor associada e opacidade, que são armazenados em uma textura 3D (tabela tridimensional).

Considerando um raio que parte do observador e atravessa um tetraedro linear, como na Figura 10, e dada uma função de transferência cujo domínio é um campo escalar, a integral de renderização de volume (eq. 2.6) pode ser expressa em função de três variáveis:

- s_f : escalar da posição mais próxima do observador (entrada do raio no tetraedro);
- s_b : escalar da posição mais distante do observador (saída do raio do tetraedro);
- l : distância percorrida pelo raio no interior do tetraedro.

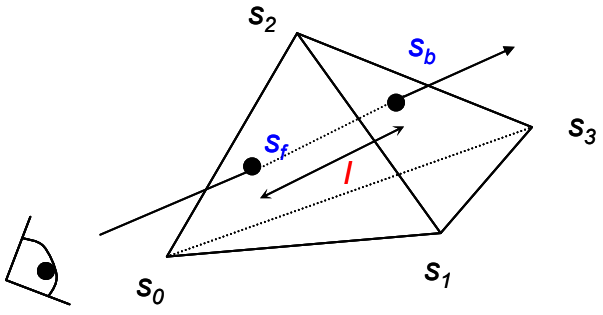


Figura 10 – Um raio que parte do observador e atravessa um tetraedro. Os escalares de entrada e saída, s_f e s_b, são interpolados a partir dos escalares associados aos vértices do tetraedro, enquanto que l é a distância entre a posição de entrada e saída do raio.

Em um tetraedro linear, o campo escalar varia linearmente em relação à distância *l* percorrida pelo raio, entre as posições de valores s_f e s_b. Assim, a eq. (2.2), relativa à opacidade do tetraedro, e o segundo termo da eq. (2.6), que resulta na cor associada *C'*, podem ser reescritos da seguinte forma (Engel et al., 2001), em função de (s_f, s_b, l):

$$\alpha = 1 - e^{-\int_0^1 \tau((1-t).s_f + t.s_b).l.dt} \qquad \textbf{(2.13)}$$

$$C' = \int_0^1 C((1-t).s_f + t.s_b).\tau((1-t).s_f + t.s_b).e^{-\int_0^t \tau((1-t').s_f + t'.s_b).l.dt'} l.dt \qquad \textbf{(2.14)}$$

Dessa forma, a função de transferência deve ser integrada para cada posição (s_f, s_b, l) da textura 3D (Figura 11), de dimensões *m* x *n* x *d*, utilizando simplesmente somas de Riemann (Malta et al., 2002) ou qualquer outro método de integração numérica. Os valores intermediários são aproximados pela interpolação linear da cor associada e da opacidade armazenadas, feita automaticamente pela placa gráfica durante o mapeamento de textura. Para o mapeamento de textura, é conveniente que s_f, s_b e *l* sejam normalizados para o intervalo [0,1).

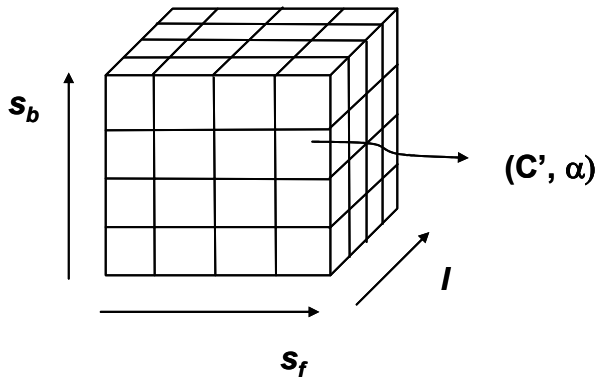


Figura 11 – Textura 3D contendo valores de cor associada e opacidade (C', α), resultantes da pré-integração da função de transferência, em função dos escalares de entrada e saída e a distância do raio (s_f, s_b, l).

A pré-integração apresenta dois problemas. O primeiro é relacionado à amostragem da função de transferência pré-integrada, principalmente para funções com altas freqüências. Se a resolução da textura 3D for muito baixa, podem ser notados erros visuais, devido à baixa amostragem da função de transferência. Por outro lado, texturas com alta resolução ocupam muito espaço na

memória da placa gráfica, apesar de que este é um problema cada vez menor para as placas modernas.

O segundo problema está relacionado à atualização da função de transferência. O custo da integração numérica é $O(ns_f * ns_b * nl * npassos)$, sendo ns_f, ns_b e l, respectivamente, as dimensões da textura 3D quanto às coordenadas s_f, s_b e l, e $npassos$ o número de passos da integração da eq. (2.6) para uma dada posição (s_f, s_b, l). Dependendo do tamanho da textura, a atualização pode levar vários segundos.

Uma grande limitação da pré-integração é que essa técnica é restrita a funções de transferência unidimensionais. Uma função de transferência multidimensional necessitaria de uma textura de dimensão superior a três, o que representa um custo de armazenamento muito alto, além desse tipo de textura não ser suportado pelas placas gráficas atuais.

Guthe et al. (2002) apresentam uma aproximação que permite reduzir a dimensão da textura para dois, sem que seja desprezada a variação do comprimento l para cada valor de (s_f, s_b). Isto é feito, primeiramente, separando a eq. (2.13), o que permite armazenar $\tau'(s_f, s_b)$ em uma textura 2D e $\alpha(l.\tau')$ em uma textura 1D, de alta resolução:

$$\tau'(s_f, s_b) = \int_0^1 \tau(s_f + t(s_b - s_f))dt \qquad \textbf{(2.15)}$$

$$\alpha(l.\tau') = 1 - e^{-l.\tau'} \qquad \textbf{(2.16)}$$

Nas placas gráficas programáveis, o cálculo da opacidade α pode ser realizado diretamente no programa por fragmento. Para determinar a cor resultante, é utilizada uma aproximação polinomial da variação da cor ao longo da distância l, para cada valor de (s_f, s_b). Ao invés de ser utilizada a cor associada resultante da eq. (2.14), o valor é "normalizado", dividindo-se a cor associada pela opacidade na distância considerada:

$$C'(l) = \frac{C(l)}{\alpha(l)} \qquad \textbf{(2.17)}$$

A cor normalizada será, então, aproximada por um polinômio de ordem n, sendo necessário resolver a eq. (2.14) para pelo menos $(n+1)$ valores de l, e a cor associada resultante será aproximada da seguinte forma:

$$C(s_f, s_b, l) \approx \alpha(s_f, s_b, l) \sum_{i=0}^{n} \frac{l^i}{l^i_{\max}} C'_i(s_f, s_b) \qquad \textbf{(2.18)}$$

São necessárias (n+1) texturas 2D em função de (s_f, s_b) para armazenar os coeficientes C'_i do polinômio. A vantagem dessa aproximação é que, utilizando texturas 2D, o custo de armazenamento é menor que no caso de texturas 3D, o que permite fazer uso de texturas de maior resolução. Utilizando texturas 3D, essa técnica poderia suportar alguns tipos de função de transferência bidimensionais, considerando um aumento no custo da pré-integração.

2.3.2. Integração de segmentos lineares

A integração de segmentos lineares foi utilizada com sucesso no sistema HIAC, apresentado por Williams et al. (1998), para a visualização (não-interativa) de malhas de elementos finitos. Nesse sistema, a função de transferência é definida por segmentos lineares, como mostrado na Figura 12. Cada ponto na fronteira de um segmento linear é chamado *ponto de controle* e define uma iso-superfície plana, para o caso de um tetraedro linear. A técnica consiste em dividir cada tetraedro em "fatias" nas quais a variação da função de transferência é linear, ou seja, entre duas iso-superfícies. O resultado de cada fatia é determinado pela integração exata do segmento linear (eq. 2.12) e composto com as contribuições anteriores.

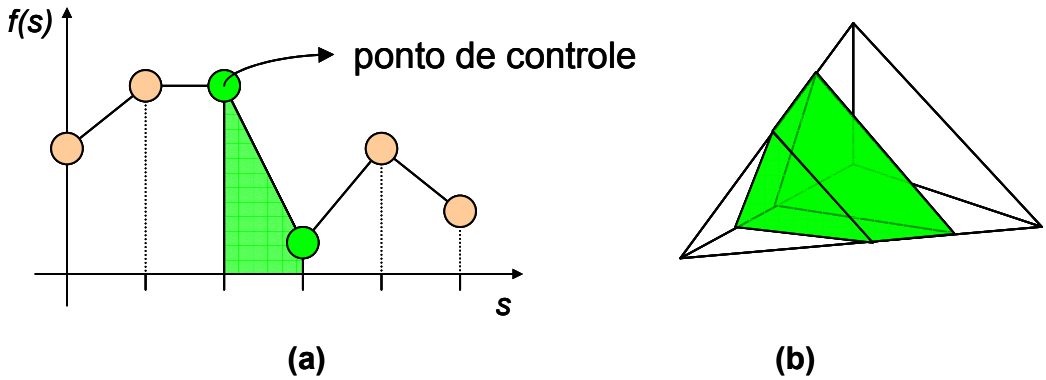


Figura 12 – (a) Função de transferência definida por segmentos lineares. (b) "Fatia" de um tetraedro limitada pelos dois pontos de controle de um segmento linear.

Recentemente, Moreland & Angel (2004) adaptaram essa abordagem para a visualização interativa de tetraedros lineares, com base em projeção de células

(seção 3.1). Para isso, exploram as placas gráficas programáveis para dividir os tetraedros em fatias durante a renderização, e a integração exata dos segmentos lineares é feita utilizando consultas a uma textura 2D contendo informações pré-calculadas.

Considerando um tetraedro linear por onde passam duas iso-superfícies definidas pela função de transferência e um raio que atravessa o tetraedro, os casos a serem considerados são os que aparecem na Figura 13.

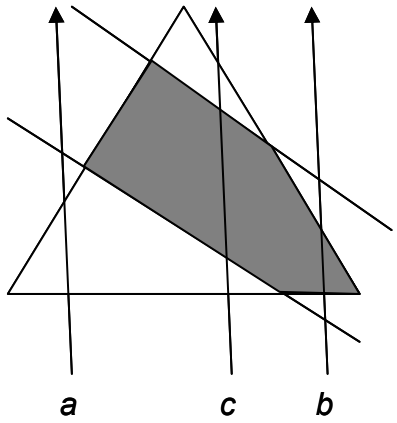


Figura 13 – Projeção de um tetraedro cortado por duas iso-superfícies definidas por pontos de controle da função de transferência. Os exemplos *a*, *b*, e *c* representam os casos fundamentais para um raio que atravessa o tetraedro.

O tetraedro deve ser desenhado uma vez para cada fatia definida por duas iso-superfícies (planos, no caso) que passam pelo tetraedro. Assim, os valores das iso-superfícies que limitam a fatia que está sendo desenhada devem ser computados na CPU e enviados para a placa gráfica. Para cada raio que atravessa o tetraedro, o programa por fragmento será responsável por determinar o caso em que o raio se encontra e se o respectivo fragmento deverá ou não ser desenhado. Supondo que a fatia a ser desenhada é a de cor *cinza*, no caso *a* o fragmento deverá ser descartado, enquanto que, no caso *b*, deverá ser desenhado e, no caso *c*, o raio deve ser projetado para os limites da fatia definidos pelos dois planos.

A contribuição mais importante de Moreland & Angel (2004) é uma nova formulação para a solução analítica da integral de renderização de volume (eq. 2.6), chamada de *pré-integração parcial*, para um segmento linear da função de transferência. A grande vantagem dessa formulação é que parte da solução da eq. (2.6) é tabulada, durante uma etapa de pré-processamento, em uma textura 2D, e é

independente da função de transferência. A solução completa pode, então, ser encontrada rapidamente através de uma consulta à textura e mais algumas poucas operações aritméticas no programa por fragmento.

Com base na eq. (2.6), é definido $C(t) = (1-t/D).C_b + (t/D).C_f$, e a equação fica:

$$I(D) = I_0 e^{-\int_0^D \tau(t)dt} + \int_0^D \left(C_b \left(1 - \frac{t}{D} \right) + C_f \frac{t}{D} \right) \tau(t) e^{-\int_t^D \tau(t')dt'} dt \qquad \textbf{(2.19)}$$

A integração por partes resulta em:

$$I(D) = I_0 e^{-\int_0^D \tau(t)dt} + C_b \left(-e^{-\int_0^D \tau(t)dt} + \frac{1}{D}\int_0^D e^{-\int_t^D \tau(t')dt'} dt \right) + C_f \left(1 - \frac{1}{D}\int_0^D e^{-\int_t^D \tau(t')dt'} dt \right) \qquad \textbf{(2.20)}$$

A partir da eq. (2.20), definem-se os seguintes termos:

$$\zeta(D, \tau(t)) \equiv e^{-\int_0^D \tau(t)dt} \qquad \textbf{(2.21)}$$

$$\psi(D, \tau(t)) \equiv \frac{1}{D}\int_0^D e^{-\int_t^D \tau(t')dt'} dt \qquad \textbf{(2.22)}$$

Substituindo as eq. (2.21) e (2.22) na eq. (2.20), chega-se à seguinte equação, que pode ser facilmente computada, dados ζ e ψ:

$$I(D) = I_0 . \zeta(D, \tau(t)) + C_b \left(\psi(D, \tau(t)) - \zeta(D, \tau(t)) \right) + C_f \left(1 - \psi(D, \tau(t)) \right) \qquad \textbf{(2.23)}$$

Assumindo-se a variação linear do coeficiente de extinção e definindo-se $\tau(t) = (1-t/D).\tau_b + (t/D).\tau_f$, chegamos a:

$$\zeta(D, \tau_f, \tau_b) = e^{-\int_0^D \left(\tau_b (1 - \frac{t}{D}) + \tau_f . \frac{t}{D} \right) . dt} = e^{-\frac{D}{2}(\tau_b + \tau_f)} \qquad \textbf{(2.24)}$$

$$\psi(D, \tau_f, \tau_b) = \frac{1}{D}\int_0^D e^{-\int_t^D \left(\tau_b (1 - \frac{t'}{D}) + \tau_f . \frac{t'}{D} \right) . dt'} dt = \int_0^1 e^{-D\int_t^1 \left(\tau_b (1-t') + \tau_f . t' \right) dt'} dt \qquad \textbf{(2.25)}$$

A eq. (2.25) pode ainda ser reescrita em função de $(\tau_f.D,\ \tau_b.D)$, o que permite armazenar os valores computados em uma textura 2D:

$$\psi(\tau_f D, \tau_b D) = \int_0^1 e^{-\int_t^1 \left(\tau_b D (1-t') + \tau_f t' \right) dt'} dt \qquad \textbf{(2.26)}$$

Para isso, ainda é conveniente normalizar os valores de ψ para o intervalo [0,1), de forma a permitir a utilização de texturas convencionais. Isso pode ser feito através de uma mudança de variáveis, primeiramente definindo:

$$\gamma \equiv \frac{\tau.D}{\tau.D+1} \tag{2.27}$$

que resulta em:

$$\tau.D = \frac{\gamma}{1-\gamma} \tag{2.28}$$

e, finalmente substituindo a eq. (2.28) na eq. (2.26):

$$\psi(\gamma_f, \gamma_b) = \int_0^1 e^{-\int_t^1 \left(\frac{\gamma_b}{1-\gamma_b}(1-t') + \frac{\gamma_f}{1-\gamma_f} t' \right) dt'} dt \tag{2.29}$$

Dessa forma, a eq. (2.23) pode ser resolvida no programa por fragmento calculando-se diretamente a eq. (2.27) e consultando-se a textura com os valores pré-computados da eq. (2.29).

Uma vantagem da integração de segmentos lineares é que essa abordagem permite utilizar funções de transferência multidimensionais, já que a solução da eq. (2.23) independe da função de transferência, o que não pode ser feito com a pré-integração. Outra é a qualidade da imagem gerada, superior à obtida com a pré-integração.

Uma desvantagem é que o custo para se desenhar um tetraedro é proporcional ao número de iso-superfícies que passam por ele. Assim, dependendo da função de transferência e do tipo de dados, pode ser necessário dividir o tetraedro diversas vezes. Moreland & Angel (2004) reportam um aumento de 3% a 4% no número de tetraedros desenhados para cada ponto de controle da função de transferência, considerando algumas malhas com resultados reais.

2.3.3. Iso-superfícies na placa gráfica

Roettger et al. (2000) apresentam uma forma de desenhar iso-supefícies de tetraedros lineares com o auxílio da placa gráfica, sem que seja necessário extraí-las explicitamente. Para isso, é realizada uma espécie de pré-integração simplificada da função de transferência, e o resultado é armazenado em uma

textura 2D. Consideremos um raio que atravessa um tetraedro, com s_f e s_b sendo, respectivamente, os valores de um campo escalar nas posições de entrada e saída do raio. Se $s_{iso1} = 0.4$, $s_{iso2} = 0.5$ e $s_{iso3} = 0.75$ são os valores de três iso-superfícies opacas, com as cores *VERMELHO*, *VERDE* e *AZUL* definidas por impulsos na função de transferência, então a integração do raio ao longo do tetraedro pode ser codificada como exemplificado na Figura 14. Cada posição (s_f, s_b) da textura contém a cor da primeira iso-superfície atravessada pelo raio. As áreas de cor preta representam as posições da textura para as quais nenhuma isso-superfície é atravessada pelo raio.

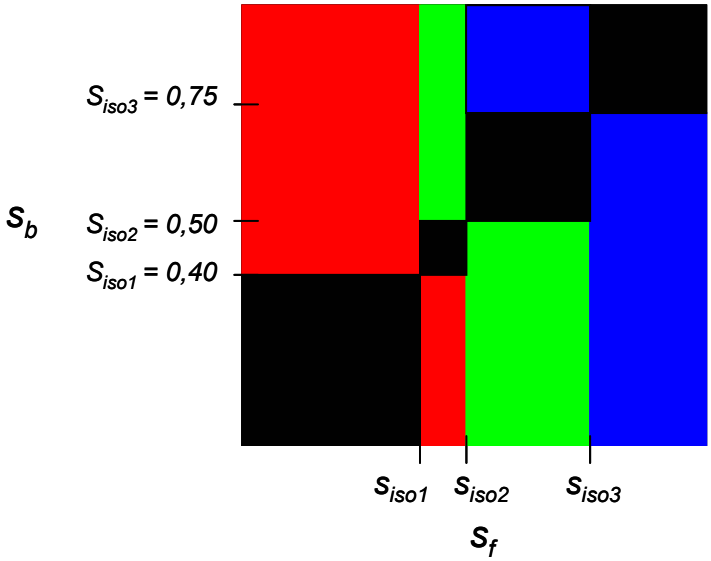


Figura 14 – Textura 2D contendo os valores de cor e opacidade da primeira iso-superfície atravessada por um raio, para cada par de coordenadas (s_f, s_b).

Se, por exemplo, $s_f = 0,2$ e $s_b = 0,8$, a primeira iso-superfície atravessada pelo raio será $s_{iso1} = 0,4$. Assim, como as superfícies são opacas, a cor relativa a s_{iso1}, armazenada na textura 2D, será a cor final do fragmento desenhado. No caso de $s_f = 0,8$ e $s_b = 0,2$, a cor final será a de $s_{iso3} = 0,75$. Por outro lado, se $s_f = 0,6$ e $s_b = 0,7$, nenhuma iso-superfície será cortada pelo raio, e portanto nenhuma cor será associada àquela posição.

Esse método pode apresentar alguns erros visuais, tais como falhas e sobreposições de iso-superfícies junto à adjacência de dois tetraedros. Para contornar esse problema, Roettger et al. (2000) propõem incrementar a espessura das iso-superfícies, aumentando um pouco a sua área na textura 2D (Figura 15).

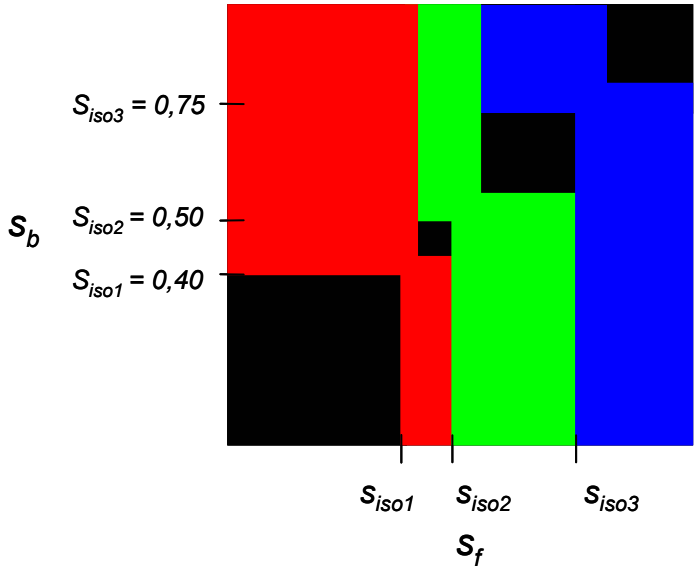


Figura 15 – As áreas relativas às iso-superfícies representadas na textura 2D são aumentadas para corrigir falhas visuais.

Uma vantagem dessa abordagem é que, independentemente do número de iso-superfícies que cortam um tetraedro, este pode ser desenhado em ordem constante (*O(1)*).

3
Algoritmos para visualização volumétrica baseados na GPU

Ao longo dos últimos anos foram utilizadas três principais classes de algoritmos para a renderização direta de volumes de malhas não-estruturadas acelerada por placas gráficas:

- traçado de raios (*ray-tracing*);
- projeção de células (*cell-projection*);
- "fatiamento" (*slicing*).

Nos algoritmos baseados em traçado de raios (Garrity, 1990; Bunyk et al., 1997), um raio é traçado para cada posição (*pixel*) do plano de projeção, e as contribuições do volume ao longo do raio são compostas para a geração da imagem final.

No caso da projeção de células (Shirley & Tuchman, 1990), cada célula do volume é projetada uma de cada vez, e sua contribuição é composta com o resultado das anteriores. Dependendo do modelo óptico utilizado, é necessário que as células sejam desenhadas segundo determinada ordem (normalmente, de trás para frente ou de frente para trás) de forma a garantir a composição correta de volumes semi-transparentes. Há diversos algoritmos para realizar essa ordenação, e alguns são apresentados resumidamente na seção (3.1.1).

Os algoritmos de fatiamento (Yagel et al., 1996; Chopra & Meyer, 2002) utilizam planos perpendiculares a uma determinada direção para "cortar" as células do volume, extraindo, assim, as "fatias" que serão compostas para a geração da imagem final. A qualidade da imagem gerada depende do número de fatias, que deve ser suficientemente alto para garantir uma amostragem adequada dos dados visualizados.

Até recentemente, os algoritmos cujos melhores desempenhos foram reportados na visualização interativa de malhas não-estruturadas eram os baseados em projeção de tetraedros (Moreland & Angel, 2004). O grande sucesso desses

algoritmos se deve à facilidade com que são adaptados para explorar os recursos de rasterização de polígonos das placas gráficas atuais.

Entretanto, utilizando os novos recursos oferecidos pelas placas gráficas programáveis, Weiler et al. (2003a) desenvolveram um algoritmo de traçado de raios que apresenta desempenho competitivo com os algoritmos de projeção de tetraedros, o que mostra o potencial dessas placas para desenvolver novos algoritmos interativos, ou ainda revisitar idéias antigas.

Os resultados reportados na literatura para os algoritmos de fatiamento (Chopra & Meyer, 2002) ainda se apresentam muito distantes dos obtidos com a projeção de células e com o algoritmo de traçado de raios de Weiler et al. (2003a). Também, até o momento, não parece haver uma forma direta de mapeá-los para as placas gráficas programáveis. Dessa forma, as abordagens estudadas e implementadas neste trabalho são baseadas na projeção de tetraedros, que é discutida na seção (3.1), e no algoritmo de traçado de raios de Weiler et al. (2003a), apresentado na seção (3.2).

3.1. Projeção de tetraedros

Um dos primeiros algoritmos para a visualização de malhas não-estruturadas a explorar a aceleração das placas gráficas para a rasterização de polígonos foi proposto por Shirley & Tuchman (1990), e é chamado de *Projected Tetrahedra* (ou Tetraedros Projetados). Esse algoritmo se baseia na projeção de tetraedros lineares, que, primeiramente, são classificados de acordo com o perfil projetado (Figura 16) relativo ao observador. Cada tetraedro projetado é, então, decomposto em triângulos, aos quais são associados valores de propriedades e que são enviados para a placa gráfica para serem desenhados. Recentemente, Wylie et al. (2002) utilizaram um programa por vértice para encapsular as computações dependentes do observador na placa gráfica. Isso proporciona o suporte para a visualização de uma primitiva TETRAEDRO, da mesma forma que as primitivas PONTO, LINHA e TRIÂNGULO, que são suportadas pelas placas gráficas atuais.

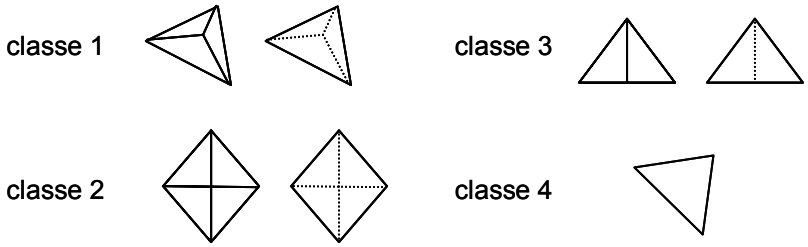


Figura 16 – Classificação de tetraedros projetados de acordo com o número de triângulos necessários para a renderização.

Explorando a programação de placas gráficas, Weiler et al. (2002) propuseram um novo algoritmo para projeção de tetraedros lineares de forma independente do observador, permitindo, também, suportá-lo diretamente pela placa gráfica como uma primitiva TETRAEDRO. Esse algoritmo foi chamado *View-Independent Cell Projection (*VICP*)*, e, devido às limitações das placas existentes na época do seu desenvolvimento, a implementação se restringia à projeção ortográfica. Posteriormente, Weiler et al. (2003b) implementaram o VICP para projeção em perspectiva, utilizando programação por fragmento. Aproveitando a flexibilidade e o desempenho cada vez maiores oferecidos pela programação por fragmentos das placas gráficas mais modernas, o VICP será o algoritmo de projeção de tetraedros utilizado no presente trabalho.

A idéia do VICP se baseia em uma técnica similar ao algoritmo de traçado de raios, mas restrito a apenas um tetraedro. A Figura 17 ilustra um raio que parte do observador e atravessa o tetraedro: s_f e s_b representam, respectivamente, os valores de um campo escalar na posição de entrada e de saída do raio, e l é a distância percorrida pelo raio no interior do tetraedro. Com o campo escalar do volume associado aos vértices do tetraedro, s_f e s_b podem ser obtidos pela interpolação linear desses valores, em relação às faces de entrada e de saída do raio. Se for utilizada pré-integração da função de transferência, (s_f, s_b, l) serão as coordenadas da textura 3D.

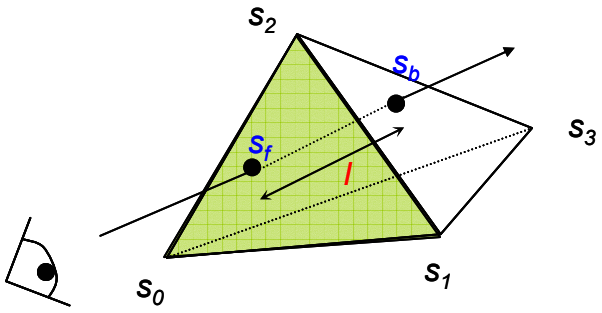


Figura 17 – Raio partindo do observador que atravessa um tetraedro linear.

Neste algoritmo, apenas as faces frontais (*front faces*) são desenhadas. Para manter a independência do observador, todas as faces podem ser enviadas para a placa gráfica, que descartará as outras (*back faces*) automaticamente. O valor de s_f é obtido através da interpolação linear realizada pela placa gráfica durante a rasterização da face, enquanto s_b pode ser calculado com a seguinte equação, onde $\vec{g}$ é o gradiente do campo escalar no tetraedro e $\hat{d}$ é a direção normalizada do raio:

$$s_b = s_f + (\vec{g}.\hat{d})l \qquad \textbf{(3.1)}$$

Em um tetraedro linear, o gradiente é constante e pode ser calculado em pré-processamento.

Se for utilizada projeção ortográfica, a direção do raio será constante para todos os tetraedros. No caso de perspectiva, a direção pode ser facilmente calculada para cada vértice no programa por vértice, subtraindo-se a posição do observador da posição do vértice. O vetor resultante será, então, interpolado automaticamente pela placa gráfica resultando na direção, que deve ser normalizada, para cada fragmento.

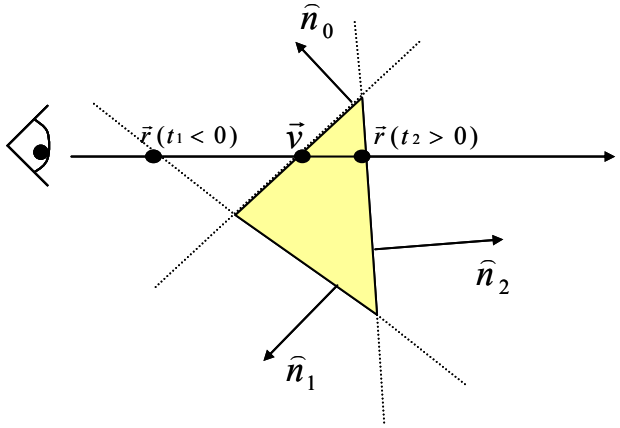


Figura 18 – Projeção lateral de um tetraedro que é atravessado por um raio que parte do observador.

Ainda resta calcular a distância *l* do raio no interior do tetraedro, que é igual à distância entre as posições de entrada e saída do raio. A posição de entrada $\vec{v}$ (Figura 18) é determinada automaticamente pela interpolação linear dos vértices da face frontal desenhada, enquanto que a posição de saída pode ser calculada pela interseção do raio com a face pela qual o raio sai do tetraedro. Considerando a equação do plano de uma face f_i, com normal $\hat{n}_i$, e um ponto $\vec{p}_i$ pertencente ao plano, além da equação paramétrica do raio $\vec{r}$:

$$(\vec{p}_i - \vec{q}).\hat{n}_i = 0 \qquad \textbf{(3.4)}$$

$$\vec{r} = \vec{v} + t\hat{d} \qquad \textbf{(3.5)}$$

o parâmetro do raio t_i, relativo à interseção $\vec{q}$ do raio com a face f_i será:

$$t_i = \frac{(\vec{p}_i - \vec{v}).\hat{n}_i}{\hat{d}.\hat{n}_i} \qquad \textbf{(3.6)}$$

que é igual à distância *l*. Assim, t_i pode ser calculado para as outras três faces do tetraedro, e o problema agora se reduz a descobrir em qual das faces o raio sai do tetraedro. Se $t_i < 0$ (Figura 18), então o raio corta o plano da face f_i antes da posição onde o raio se inicia (no caso, $\vec{v}$). Logo, o raio não pode estar saindo do tetraedro. Como conseqüência, a face de saída será a que tiver o menor t_i positivo, o que significa que f_i é a primeira face cortada pelo raio na direção $\hat{d}$.

Esse algoritmo pode ser implementado utilizando-se a programação por vértice e por fragmento. Para cada um dos 3 vértices das 4 faces do tetraedro, o que resulta em 12 vértices por tetraedro, os seguintes atributos devem ser enviados para a placa gráfica:

- posição do vértice ($\vec{v}$);
- valor do campo escalar no vértice (*s*);
- normais das outras faces ($\hat{n}_a, \hat{n}_b, \hat{n}_c$);
- posição do vértice oposto à face, pertencente às outras três ($\vec{p}_i$);
- gradiente do tetraedro ($\vec{g}$).

A memória necessária para cada vértice, juntamente com seus atributos, é igual a $3+1+3*3+3+3=19\ floats = 76\ bytes$, considerando-se que cada valor é representado por um número de ponto flutuante (*float*) de 4 *bytes*. Como cada tetraedro requer 12 vértices, o custo por tetraedro é igual a 912 *bytes*.

Como atualmente não é possível compartilhar informações entre os vértices de uma mesma face (ou tetraedro) nos programas por vértice, é necessário que algumas informações sejam enviadas de forma redundante. Cada atributo do vértice será interpolado automaticamente durante a rasterização da face. Dessa forma, para cada fragmento, estarão disponíveis a posição $\vec{v}$ de entrada do raio e o escalar s_f nessa posição, além dos outros atributos, que são iguais para todos os vértices da face. Com essas informações, as eq. (3.6) e (3.1) podem finalmente ser resolvidas no programa por fragmento.

Devido aos atributos de um vértice serem diferentes para as 3 faces de que o vértice faz parte, não é possível utilizar uma "faixa" (*strip*) de triângulos para desenhar as faces do tetraedro, o que reduziria o número de vértices enviados para 6 por tetraedro. De fato, como reportam Weiler et al. (2002), o grande volume de dados transferidos para a placa gráfica representa o "*gargalo*" do VICP.

Se for utilizada projeção ortográfica, o número de parâmetros por vértice pode ser reduzido. Para um raio que entra em um tetraedro pela face f_3 (Figura 19), a face de saída do raio deve ser f_0, f_1 ou f_2. Weiler et al. (2002) observam que, para projeção ortográfica, a distância t_i, entre o ponto de entrada do raio e a interseção com a face f_i, pode ser interpolada linearmente ao longo da face de entrada, para cada possível face de saída f_i. Por exemplo, as distâncias do vértice v_1 para as faces f_0, f_1 e f_2 na direção do raio são, respectivamente, *0*, t_1 e *0*, pois esse vértice é compartilhado pelas faces f_0 e f_2 (e também f_3). O equivalente ocorre para os outros dois vértices, v_0 e v_2, da face f_0. Logo, a distância t_i de qualquer posição na face f_3 para as outras faces pode ser determinada pela interpolação linear das distâncias associadas aos vértices de f_3 (Figura 19).

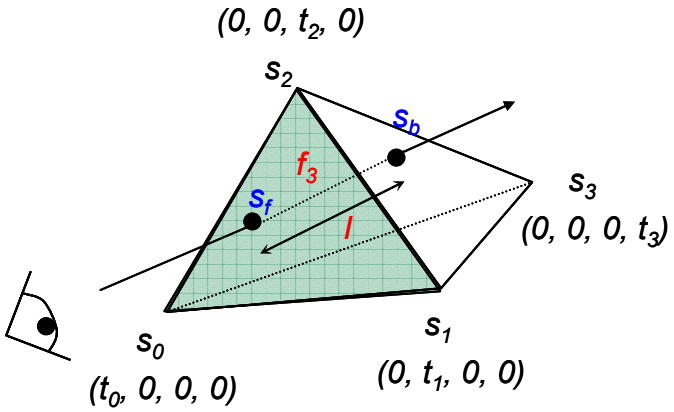


Figura 19 – Raio que parte do observador e atravessa um tetraedro. Cada vértice do tetraedro possui um vetor (t_0, t_1, t_2, t_3) das respectivas distâncias às faces f_0, f_1, f_2 e f_3, na direção do raio.

Dessa forma, para cada vértice v_i, podemos enviar como parâmetro a equação do plano $\vec{o}_i$ da face oposta e calcular a distância t_i no programa por vértice. Os atributos necessários para cada vértice são:

- posição do vértice ($\vec{v}$);
- valor do campo escalar no vértice (s);
- equação do plano da face oposta ao vértice ($\vec{o}_i$);
- gradiente do tetraedro ($\vec{g}$).

Uma vantagem é que, além de reduzir o número de atributos de cada vértice, essa otimização permite utilizar uma faixa de triângulos para desenhar um tetraedro, pois os atributos não dependem da face que está sendo desenhada, mas apenas do vértice.

O custo de armazenamento por vértice será, então, igual a 11 *floats* = 44 *bytes*. Se forem utilizadas faixas de triângulos, o custo por tetraedro será igual a 264 *bytes*.

Em seu sistema para visualização volumétrica, Moreland & Angel (2004), baseando-se no modelo óptico de emissão e absorção e na abordagem de Weiler et al. (2002), restrita à projeção ortográfica, utilizam as idéias do VICP mas defendem uma abordagem dependente do observador. Os algoritmos de projeção de células requerem uma ordenação de visibilidade dependente do observador. Essa ordenação deve ser realizada a cada vez em que a malha é desenhada, o que

implica em enviar novamente os vértices, ou seus índices, para a placa gráfica, o que é reportado por Weiler et al. (2002) como o gargalo do algoritmo. Dessa forma, não haveria custo adicional significativo em se realizarem mais cálculos na CPU. Moreland & Angel (2004) apostam nisso para reduzir a quantidade de atributos enviados por vértice, computando o parâmetro t_i de cada vértice na CPU e enviando-o como atributo do vértice. Além disso, é enviado o valor do campo escalar s_{bi} na posição determinada por t_i, o que permite eliminar o gradiente do tetraedro como um atributo do vértice. O índice local do vértice no tetraedro também é necessário, de forma que o programa de fragmentos possa representar t_i e s_{bi} como tuplas que serão interpoladas. A interpolação de s_{bi} é equivalente à de t_i (Figura 19). Os atributos por vértice são:

- posição do vértice ($\vec{v}$);
- valor do campo escalar no vértice (s)
- distância do vértice para a face oposta, na direção do raio (t_i);
- valor do campo escalar na interseção do raio com a face oposta ao vértice (s_{bi});
- índice local do vértice no tetraedro ($i \in [0,3]$).

Dessa forma, Moreland & Angel (2004) reduzem o custo de memória por vértice para 7 *floats* = 28 *bytes*, e conseqüentemente, com o uso de faixas de triângulos, para 168 *bytes* por tetraedro.

Entretanto, neste trabalho, a restrição à projeção ortográfica não foi uma opção. Assim, nos baseamos na abordagem de Weiler et al. (2003b) e aumentamos o esforço do programa por fragmento para remover o gargalo do VICP, como apresentado na seção (4.1.1).

3.1.1. Ordenação de células

Como mencionado, além de uma forma de desenhar cada uma das células, os algoritmos baseados em projeção de células requerem uma *ordenação de visibilidade*, considerando o modelo óptico de emissão e absorção.

Williams (1992) define que "*uma ordenação de visibilidade de um conjunto de objetos, a partir de um observador, é uma ordenação tal que, se o objeto a obstrui o objeto b, então b precede a na ordenação*".

Diversos algoritmos de ordenação de visibilidade para malhas de poliedros foram desenvolvidos, sendo que os mais populares são baseados no algoritmo *Meshed Polyhedra Visibility Ordering* (MPVO), apresentado por Williams (1992). Esse algoritmo ordena um conjunto de células convexas e de faces planas, de uma malha também convexa e acíclica (i.e., sem ciclos de visibilidade (Figura 20)). A grande vantagem do MPVO é ser capaz de realizar a ordenação em tempo e espaço de armazenamento lineares (*O(n)*) em relação ao número *n* de células, além de sua simplicidade.

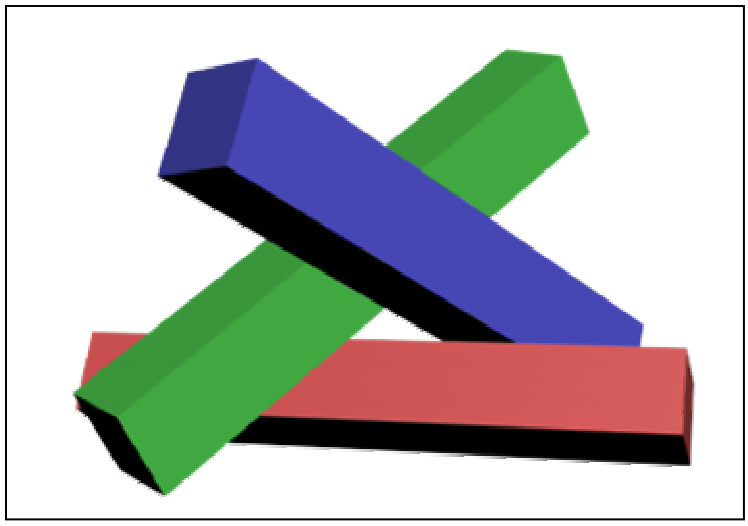


Figura 20 – Exemplo de um ciclo de visibilidade.

O MPVO é composto por três etapas. A primeira (pré-processamento) consiste em construir um grafo de adjacências entre as células (Figura 21) e computar as equações dos planos de todas as faces internas da malha. Na segunda etapa, o algoritmo (assumindo que a malha é acíclica) transforma o grafo de adjacências em um grafo orientado acíclico (DAG), atribuindo direções às arestas do grafo de adjacências (Figura 21a). Para isso, avalia-se a equação do plano de cada face interna e a aresta do grafo é marcada como “entrando”, “saindo” ou “nenhum”, se a posição do observador estiver à frente, atrás ou sobre o plano, respectivamente (para projeção ortográfica, a direção de visualização é que será considerada). Assim, para duas células que compartilham uma face, se em uma a face estiver marcada como “entrando”, na outra estará “saindo”, ou ambas estarão marcadas como “nenhum”, o que implica que não existe uma conexão no DAG para a face. Finalmente (Figura 21b), a terceira etapa realiza uma ordenação

topológica do DAG, utilizando busca em largura (BFS) ou profundidade (DFS), o que resulta em uma ordenação de visibilidade das células. No caso da busca em profundidade, as células que não possuem arestas no DAG marcadas como "saindo" são colocadas em uma lista *L*, durante a segunda etapa. Para cada célula da lista, todas as suas adjacências que estejam marcadas como "entrando" são, então, visitadas recursivamente.

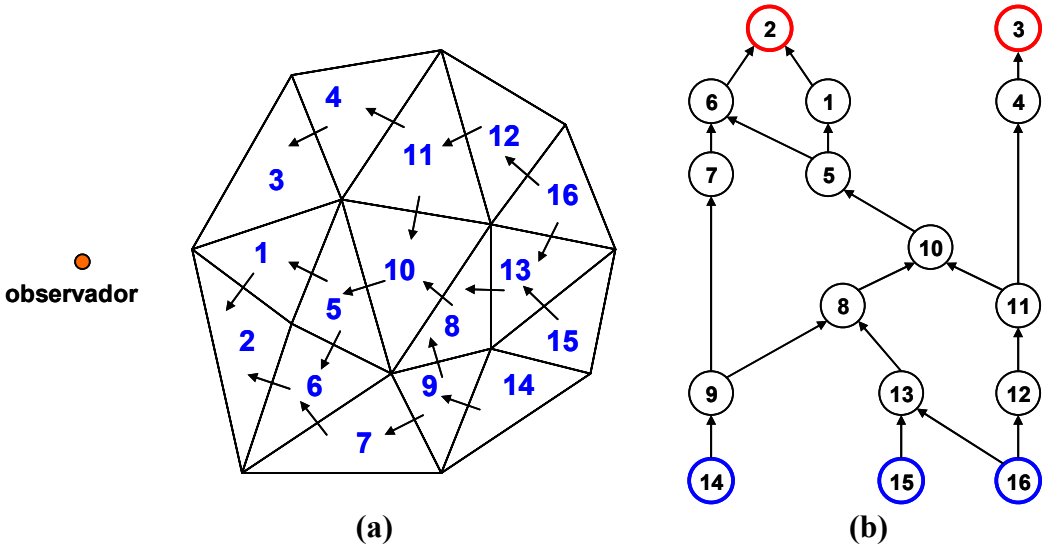


Figura 21 – Ilustração do algoritmo MPVO. (a) As setas representam a classificação das faces internas como "entrando", "saindo" ou "nenhum", em relação ao observador. (b) Grafo orientado criado a partir da classificação das faces.

Adicionalmente, Williams (1992) apresenta uma extensão do MPVO para a ordenação de visibilidade de malhas não-convexas, chamada MPVONC, com complexidade $O(n + b \log b)$, sendo n o número total de células e b o número de células externas (da fronteira da malha). O MPVONC é uma heurística, o que implica que, em alguns casos, podem ocorrer falhas na ordenação de uma malha. Posteriormente, outras extensões foram propostas para o MPVO. A primeira a conseguir ordenar malhas não-convexas foi apresentada por Silva et al. (1998) e utiliza o paradigma de plano de varredura (de Berg et al., 2000) para estender o DAG da segunda etapa do MPVO, com relações adicionais entre as faces externas de uma malha não-convexa (Figura 22). Comba et al. (1999) utilizaram árvores BSP para determinar essas relações adicionais. Kraus & Ertl (2001) apresentam uma extensão para o MPVO que permite ordenar malhas com ciclos de visibilidade. Para isso, observam que um ciclo de visibilidade pode ser

representado como componentes fortemente conexas de um grafo orientado. Assim, modificam a terceira etapa do MPVO para extrair e ordenar esses ciclos, e dividem as células envolvidas para que elas possam ser visualizadas corretamente. Recentemente, Cook et al. (2004) utilizaram o espaço da imagem projetada para determinar as relações adicionais ao MPVO, reportando os melhores tempos entre as extensões do MPVO que ordenam malhas não-convexas. Para criar as relações adicionais entre as faces externas, a placa gráfica é utilizada para desenhá-las, armazenado o resultado em cada elemento (*pixel*) da imagem gerada em uma estrutura similar a um *A-Buffer* (Carpenter, 1984), que mantém as contribuições de cada pixel de forma ordenada. Finalmente, o resultado final de cada pixel é utilizado para criar as relações adicionais.

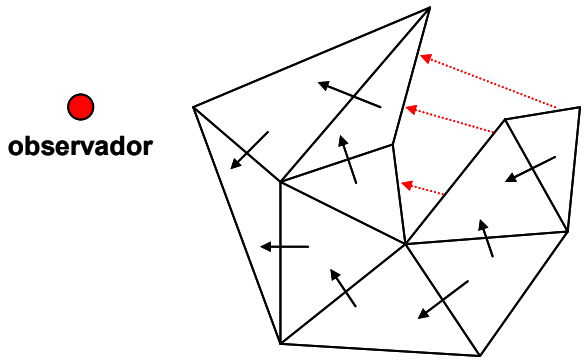


Figura 22 – Extensão das relações de adjacência do MPVO, para malhas não-convexas, representada pelas setas vermelhas.

Outras abordagens para a ordenação de células podem ser encontradas nos trabalhos de Carpenter (1984), Wittenbrink (2001), Cignoni & Floriani (1998), Farias et al. (2000) e Stein et al. (1994).

Neste trabalho, o MPVONC foi utilizado para a ordenação das células. Essa heurística apresentou resultados satisfatórios, tanto quanto ao desempenho como quanto à qualidade das imagens (apesar de ser uma heurística) para as malhas de elementos finitos que foram testadas.

3.2. Traçado de raios na placa gráfica

O algoritmo proposto por Weiler et al. (2003a) é baseado no paradigma de traçado de raios. Com praticamente todos os cálculos sendo realizados através da

programação por fragmento na placa gráfica, Weiler et al. (2003a) conseguiram implementar um sistema interativo para a visualização de malhas de tetraedros lineares, com desempenho competitivo com os algoritmos de projeção de células. Esse algoritmo estende o algoritmo VICP, que utiliza traçado de raios apenas no interior do tetraedro que será projetado, para percorrer uma seqüência de tetraedros ao longo de cada raio que parte do observador.

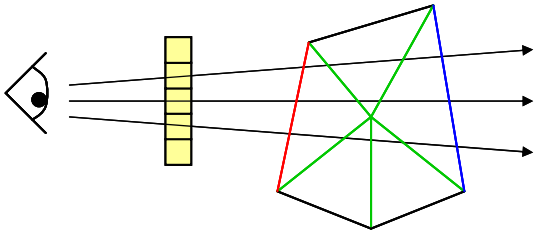


Figura 23 – Propagação de raios que partem do observador e atravessam a malha de tetraedros. A cada passo de um raio, é atravessado um tetraedro.

Seguindo a idéia geral dos algoritmos de traçado de raios (Figura 23), cada elemento (*pixel*) da imagem projetada corresponde a um raio que parte do observador e passa pela posição do pixel. O valor final é igual ao resultado da composição das contribuições de cada tetraedro interceptado ao longo do raio.

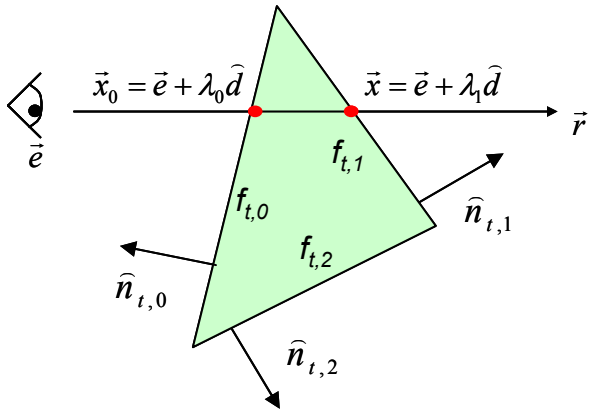


Figura 24 – Interseção de um raio com as faces de um tetraedro.

Inicialmente, é preciso encontrar a primeira interseção, que estará localizada em uma face da fronteira externa da malha. Essa face representa a posição de entrada $\vec{x}_0$ do raio em um tetraedro (Figura 24), assim como no VICP. Como conseqüência, a próxima interseção será a posição de saída $\vec{x}$, que, por sua vez, é

igual à posição de entrada do raio em um tetraedro adjacente, e assim sucessivamente (Figura 23), até que o raio atinja a posição de saída da malha, que será uma face sem um tetraedro adjacente. A cada passo, as contribuições de cor e opacidade de cada tetraedro ao longo do raio são compostas com o resultado anterior, da frente para trás. Assim, para cada raio, as etapas do algoritmo são:

1. Encontre a primeira interseção.
2. Enquanto o raio estiver dentro da malha:
   2.1. Determine a posição de saída do tetraedro.
   2.2. Calcule a contribuição do tetraedro.
   2.3. Componha com o resultado anterior.
   2.4. Avance para o tetraedro adjacente.

Cada passo representa um tetraedro que é atravessado pelo raio, como na Figura 24. Considerando a eq. (3.7), do plano da *i-ésima* face ($f_{t,i}$) do tetraedro *t*, com normal $\hat{n}_{t,i}$ e um vértice $\vec{v}_{t,3-i}$ da face, além da equação paramétrica do raio $\vec{r}$ (eq. 3.8), que parte do observador $\vec{e}$ na direção normalizada $\hat{d}$, temos:

$$(\vec{v}_{t,3-i} - \vec{q}).\hat{n}_{t,i} = 0 \qquad \textbf{(3.7)}$$

$$\vec{r} = \vec{e} + \lambda\hat{d} \qquad \textbf{(3.8)}$$

O parâmetro do raio λ_i, relativo à interseção $\vec{q}$ do raio com a face $f_{t,i}$ será, então:

$$\lambda_i = \frac{(\vec{v}_{t,3-i} - \vec{e}).\hat{n}_{t,i}}{\hat{d}.\hat{n}_{t,i}} \qquad \textbf{(3.9)}$$

Se a face de entrada for conhecida, o parâmetro λ_i deve ser calculado para as outras três faces, e a posição de saída será correspondente ao menor λ_t que seja maior que o valor relativo à posição de entrada.

O valor do campo escalar $s(\vec{x})$ na posição de saída $\vec{x}$ do tetraedro pode ser calculado da seguinte forma:

$$s(\vec{x}) = \vec{g}_t.(\vec{x} - \vec{x}_0) + s(\vec{x}_0) = \vec{g}_t.\vec{x} + (s(\vec{x}_0) - \vec{g}_t.\vec{x}_0) \qquad \textbf{(3.10)}$$

onde $\vec{g}_t$ é o gradiente do campo escalar no tetraedro *t*, e $\vec{x}_0$, neste caso, pode ser uma posição qualquer no interior ou em uma face do tetraedro. Definindo $\hat{g}_t = (s(\vec{x}_0) - \vec{g}_t.\vec{x}_0)$, a eq. (3.10) pode ser reescrita como:

$$s(\vec{x}) = \vec{g}_t.\vec{x} + (s(\vec{x}_0) - \vec{g}_t.\vec{x}_0) = \vec{g}_t.\vec{x} + \hat{g}_t \quad \textbf{(3.11)}$$

Para que seja possível propagar o raio, cada passo deve receber como parâmetros o índice *t* do tetraedro, o parâmetro λ, relativo à posição de entrada do raio no tetraedro e a cor e opacidade compostas até o passo anterior. Assim, as informações podem ser representadas como uma tupla (*t, λ, R, G, B, A*). Além dos parâmetros, deve-se ter acesso a uma estrutura de dados contendo as informações geométricas e topológicas da malha de tetraedros.

A propagação deve ser feita para cada pixel da tela de projeção, o que sugere a utilização de um programa por fragmento como forma de implementação do algoritmo na placa gráfica. Para isso, pode-se desenhar um retângulo do tamanho da tela, apenas como uma forma de permitir que o programa seja executado para todos os *pixels*. Se a placa gráfica suportar fluxo dinâmico (com estruturas do tipo *while ... do ... end*), a propagação completa de um raio pode ser realizada no programa por fragmento, que precisará ser executado apenas uma vez para cada pixel.

Entretanto, na maioria das placas gráficas disponíveis atualmente, essa facilidade ainda não se encontra disponível. Assim, uma forma de realizar a propagação do raio consiste em desenhar vários retângulos do tamanho da tela, de modo que, a cada retângulo desenhado, seja executado um passo da propagação. Os retângulos são desenhados em uma textura 2D do tamanho da tela de projeção, que será utilizada no passo seguinte como uma forma de transferir dados entre passos subseqüentes.

A posição de entrada de um raio na malha de tetraedros pode ser determinada simplesmente desenhando-se as faces externas. Já para descobrir se o raio saiu da malha, basta comparar o índice *t* do tetraedro atual com um valor inválido (o índice *0* (zero) pode ser convenientemente utilizado). Em caso positivo, o raio não deve continuar sendo propagado, o que implica em não mais executar o programa por fragmento para o respectivo pixel. Se for utilizado fluxo dinâmico no programa por fragmento, a implementação é direta, mas, no caso contrário, o problema se torna um pouco mais complicado. Na verdade, dois casos devem ser tratados: *um raio saiu da malha* e *todos os raios saíram da malha*.

Quanto ao primeiro, mesmo que o raio não deva mais ser propagado, é necessário continuar desenhando retângulos para permitir a propagação dos outros

que ainda não deixaram a malha. Enquanto estiverem sendo desenhados retângulos, o resultado final do raio deve ser comunicado para o passo seguinte.

Nas placas gráficas, a determinação da visibilidade de um pixel é realizada utilizando-se o algoritmo de *z-buffer* (Foley et al., 1997). Assim, uma maneira de descobrir se todos os raios saíram da malha, o que significa que se deve parar de desenhar retângulos, consiste em executar um programa por fragmento para colocar no z-buffer um valor mínimo quando o raio não estiver mais na malha, de modo que, no passo seguinte, o fragmento não será mais desenhado. Isso pode ser implementado como um passo adicional, executado após um determinado número de passos. Esse passo é responsável por "marcar" o z-buffer, cumulativamente, a cada vez em que é executado, até que nenhum fragmento passe mais pelo teste de profundidade. A contagem do número de fragmentos desenhados (*occlusion query*) é uma facilidade suportada de forma relativamente eficiente pelas placas gráficas modernas.

Em malhas não-convexas (Figura 25), às vezes é necessário que um raio que deixou a malha entre novamente. Uma limitação do algoritmo de Weiler et al. (2003a) é que, quando o raio sai da malha, não é mais possível explorar a coerência entre tetraedros adjacentes para permitir que ele entre novamente. Assim, na abordagem original, somente malhas convexas são suportadas.

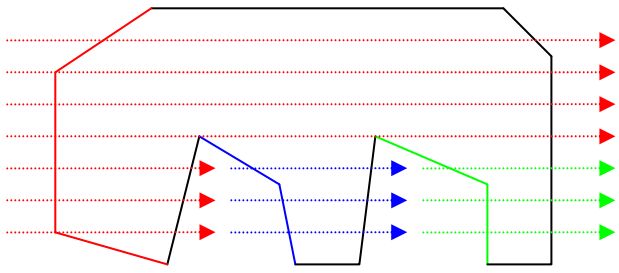


Figura 25 – Os raios de cores diferentes representam as camadas necessárias para a visualização de uma malha não-convexa.

Enquanto Weiler et al. (2003a) tornam as malhas convexas adicionando células "inativas", Weiler et al. (2004), e paralelamente Bernardon et al. (2003), propõem utilizar a técnica de "descamação" (*depth-peeling*) (Everitt, 2001). Como ilustrado na Figura 25, para cada vez que raios entram na malha, é desenhada uma nova "camada" de faces frontais para determinar as novas

posições de entrada, que serão relativas às faces frontais mais próximas do observador, desconsiderando-se as camadas anteriores. A implementação pode ser realizada, considerando o algoritmo de z-buffer, por um programa por fragmento utilizado para comparar a profundidade de um fragmento projetado com o valor relativo à camada anterior, armazenado em uma textura 2D, de forma a permitir somente os valores de profundidade maiores. Isso é complementado com o teste de profundidade convencional da placa gráfica, configurado para permitir apenas os valores de fragmentos menores do que os armazenados no z-buffer. Dessa forma, a cada vez em que é determinado que todos os raios saíram da malha, é necessário desenhar novamente as faces externas para descobrir se há raios que voltam a entrar na malha. O algoritmo termina quando a contagem do número de fragmentos desenhados for igual a zero.

Assim, as etapas do algoritmo modificado são:

1. Desenhe uma camada.
2. Se há fragmentos desenhados:
   2.1. Enquanto o raio estiver dentro da malha:
      2.1.1. Determine a posição de saída do tetraedro.
      2.1.2. Calcule a contribuição do tetraedro.
      2.1.3. Componha com o resultado anterior.
      2.1.4. Avance para o tetraedro adjacente.
   2.2. Vá para o item 1.

3.2.1. Estruturas de dados

As estruturas de dados utilizadas por este algoritmo devem estar armazenadas na memória da placa gráfica, de modo que o programa por fragmento possa acessá-las e, assim, realizar as operações necessárias à propagação do raio ao longo da malha de tetraedros. Portanto, devem ser representadas como texturas 1D, 2D ou 3D, que são atualmente as únicas formas possíveis de acesso a grandes volumes de dados estáticos por um programa por fragmento. Como a dimensão de cada coordenada de uma textura ainda é limitada (as placas atuais suportam, em média, até 4096 posições), as texturas 1D só

podem ser utilizadas para pequenos volumes de dados. Dessa forma, as texturas 2D são as mais utilizadas, transformando-se o índice *i* de uma posição de um vetor unidimensional em um par de coordenadas de textura *(u,v)*, como ilustrado na Figura 26. O índice *t* de um tetraedro, por exemplo, pode ser decomposto em dois índices: t_u e t_v, representados como coordenadas de uma textura 2D contendo as informações do tetraedro.

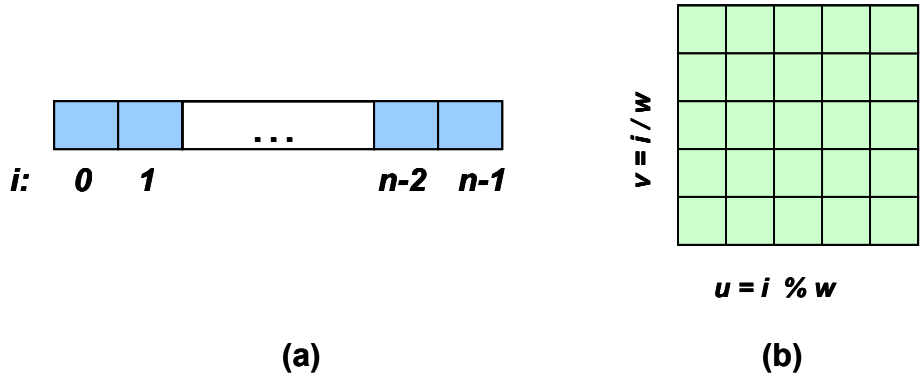


Figura 26 – (a) Vetor unidimensional indexado pelo índice *i*. (b) Textura 2D equivalente ao vetor. O índice *i* é decomposto em duas coordenadas de textura, *u* e *v*. As dimensões da textura são *w* e *h*.

As informações que são passadas de um passo do algoritmo para o seguinte podem ser codificadas em uma textura 2D, em função das coordenadas *(x, y)* dos pixels da tela de projeção, como na Tabela 1. Para cada posição da textura, os dados (*t, λ, R, G, B, A*) são armazenados nas componentes de cor e opacidade *(r, g, b, a)*. As placas gráficas modernas (NVIDIA, 2004b) permitem criar texturas com valores de ponto flutuante (*float*) de 32 *bits* por componente, além de compactar dois valores de "meia-precisão" (*half*), de 16 bits, em um *float* de 32 bits. Segundo NVIDIA (2004b), inteiros no intervalo [-2048, 2048] podem ser exatamente representados como um valor *half*. Assim, informações tais como o índice do tetraedro atual, podem ser representadas por valores do tipo *half* e compactadas em *floats*, para serem armazenadas na textura. As outras informações também podem ser compactadas da mesma forma, permitindo alguns erros de precisão que, em geral, não representam um problema. No caso da cor associada e da opacidade, a precisão de 16 *bits* ainda é melhor do que os 8 *bits* utilizados tradicionalmente.

Tabela 1 – Dados armazenados na textura 2D utilizada para a passagem de parâmetros para um passo da propagação de um raio.

Coordenadas		**Dados**					
u	**v**	**r**	**g**	**b**		**a**	
x	y	t	λ	R	G	B	A

O programa por fragmento também precisa acessar as informações geométricas e topológicas dos tetraedros da malha. Devido às limitações das placas gráficas, quando o algoritmo original foi desenvolvido, Weiler et al. (2003a) utilizaram uma estrutura de dados simples na qual, para cada tetraedro de índice t, são armazenados as posições dos vértices, as normais das faces e o gradiente do campo escalar. As únicas informações topológicas são os índices dos tetraedros adjacentes às faces de cada tetraedro. Essa estrutura pode ser implementada com texturas 3D, como mostrado na Tabela 2. Bernardon et al. (2004) decompõem cada textura 3D em texturas 2D, cujo acesso é normalmente mais eficiente.

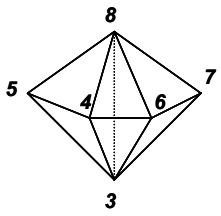
Tabela 2 – Estrutura de dados utilizada para o Traçado de Raios na placa gráfica, originalmente usada por Weiler et al. (2003a).

Textura	**Coordenadas**			**Dados**			
	u	**v**	**w**	**r**	**g**	**b**	**a**
Vértices	t		i	$\vec{v}_{t,i}$			
Normais	t		i	$\hat{n}_{t,i}$			$f_{t,i}$
Gradientes	t			$\vec{g}_t$			$\hat{g}_t$
Adjacências	t		i	$a_{t,i}$			

Na tabela acima, $\hat{n}_{t,i}$ é a normal da face com índice local i no tetraedro t, $\vec{v}_{t,i}$ é o vértice oposto à face, $a_{t,i}$ é o índice do tetraedro adjacente, $f_{t,i}$ é o respectivo índice local da face no tetraedro adjacente, $\vec{g}_t$ é o gradiente do tetraedro e $\hat{g}_t$ é o termo escalar da eq. (3.11). Na abordagem original, o índice local da face de entrada do tetraedro também é passado como parâmetro de cada passo da propagação do raio.

O espaço de memória necessário para a estrutura de dados é igual a $(4*3)T + (4*4)T + 4T + 4T = 36T$, sendo T o número de tetraedros. Isso corresponde, por tetraedro, a *36 floats = 144 bytes*, o que representa um alto custo de memória para grandes malhas. Entretanto, Weiler et al. (2003a) observaram que, futuramente, com a evolução das placas gráficas, o espaço poderia ser reduzido, por exemplo, para *16 bytes* por vértice e *64 bytes* por tetraedro com otimizações como índices para normais e vértices, redução da precisão da normal para o tipo *half* e armazenamento de apenas uma normal para uma face compartilhada por dois tetraedros.

Weiler et al. (2004) propõem a codificação das malhas em faixas de tetraedros (Figura 27) como forma de reduzir o custo de armazenamento. Nesta codificação, a adjacência de uma face de um tetraedro é determinada por duas informações: o *índice da faixa (strip) onde está o tetraedro vizinho* e o *índice do primeiro vértice do tetraedro vizinho na respectiva* faixa. Assim, (2,3) corresponde ao terceiro vértice da segunda faixa. No exemplo da Figura 27, essas informações estão armazenadas no primeiro vértice v_k de cada tetraedro t, e (0,0) representa uma face sem adjacência. Observando que, em um tetraedro do interior de uma faixa, as adjacências a_0 e a_3 pertencem à mesma faixa do tetraedro, a_1 e a_2 podem ser deslocados para o próximo vértice e a_0 e a_3 compactados em a_1 e a_2, como ilustrado na Figura 27c. Para a_0 e a_3, é associado um valor igual a 1, para os vizinhos da mesma faixa ("vizinhos implícitos"), ou 0, para os de outra faixa ("vizinhos explícitos").



faixa (*strip*) 2						
v_k	5	4	3	8	6	7
a_0	2,2	2,3	1,6	-	-	-
a_1	0,0	1,3	3,5	-	-	-
a_2	1,1	3,3	0,0	-	-	-
a_3	3,1	2,1	2,2	-	-	-
	1	2	3	4	5	6

faixa (*strip*) 2 (compactada)						
v_k	5	4	3	8	6	7
a_0	-	1	1	0	-	-
a_1	-	0,0	1,3	3,5	1,6	-
a_2	3,1	1,1	3,3	0,0	-	-
a_3	-	0	1	1	-	-
	1	2	3	4	5	6

(a) **(b)** **(c)**

Figura 27 – (a) Faixa com 3 tetraedros. (b) Representação da faixa, onde v_k é o índice global de um vértice e a_0-a_3 são as adjacências de um tetraedro da faixa, armazenadas na posição do primeiro vértice do tetraedro. (c) Representação compactada da faixa.

Cada uma das adjacências a_1 ou a_2, da faixa compactada, pode ser armazenada em um *float* de 32 *bits*, utilizando 16 *bits* para os índices da faixa e do tetraedro na faixa. Os valores (0 ou 1) de a_0 e a_3 podem ser compactados nos *bits* mais significativos de a_1 e a_2, que serão efetivamente armazenadas, juntamente com o índice do vértice v_k. Assim, para uma faixa cujo comprimento tende ao infinito, o custo de memória por tetraedro, relativo à faixa, é aproximadamente igual a 3 *floats*. As texturas utilizadas para a implementação dessa estrutura de dados são apresentadas na Tabela 3.

Tabela 3 – Texturas 2D utilizadas para o armazenamento de faixas de tetraedros.

Textura	**Coordenadas**		**Dados**			
	u	**v**	**r**	**g**	**b**	**a**
Faixas (*strips*)	t		$v_{t,0}$	$a_{t,1}$	$a_{t,2}$	
Vértices	k		$\vec{v}_k$			s_k
Normais	j		$\hat{n}_j$			o_j
Gradientes	t		$\vec{g}_t$			$\hat{g}_t$
Índice das normais	t		$n_{t,0}$	$n_{t,1}$	$n_{t,2}$	$n_{t,3}$

Na tabela acima, t é o índice do tetraedro, k é o índice de um vértice e j é o índice do vetor normal ao plano de uma face; $v_{t,0}$ é o índice do primeiro vértice de um tetraedro da faixa, enquanto que $a_{t,1}$ e $a_{t,2}$ são as informações de adjacência compactadas; $\vec{v}_k$, $(\hat{n}_j, o_j)$ e $\vec{g}_t$ são a posição de um vértice, a equação do plano de uma face e o gradiente de um tetraedro, respectivamente; s_k é o escalar associado ao vértice, e $\hat{g}_t$ é o termo escalar da eq. (3.11) . Finalmente, $n_{t,i}$ é o índice da equação do plano da *i-ésima* face do tetraedro, na textura de normais.

Apenas as texturas de Faixas e de Vértices são necessárias, segundo observam Weiler et al. (2004), pois o valor do campo escalar $s(\vec{x})$ em qualquer posição do tetraedro pode ser interpolado utilizando-se coordenadas baricêntricas. As equações dos planos das faces do tetraedro também podem ser calculadas a partir da posição dos vértices. Dessa forma, as outras três texturas são opcionais, e são utilizadas para reduzir o tamanho do programa por fragmento. Se apenas as duas primeiras texturas forem utilizadas (*Faixas* e *Vértices*), e assumindo que o

número de vértices V é aproximadamente 1/5 do número de tetraedros T (Beall & Shephard, 1997), então o custo por tetraedro é $3 + 4/5 \approx 3{,}8\,floats \approx 15\,bytes$.

Por outro lado, quando as texturas dos planos das normais são utilizadas, a textura de vértices não é necessária, já que as interseções do raio podem ser computadas diretamente pelas equações desses planos. Considerando as texturas de normais e gradientes, e cada normal compartilhada por dois tetraedros sendo armazenada apenas uma vez, então o custo por tetraedro será aproximadamente $3 + 2 * 4 + 4 + 4 = 19\,floats = 76\,bytes$, o que representa quase metade do custo da estrutura de dados original, utilizada por Weiler et al. (2003a).

A utilização de faixas de tetraedros representa uma alternativa interessante para reduzir a memória utilizada. Entretanto, também introduz alguma complexidade. O problema da minimização do número de faixas para uma malha é NP-completo (Weiler et al., 2004), o que implica na aplicação de heurísticas. Em comparação com a utilização de 4 índices de vértices e 4 de adjacências por tetraedro, Weiler et al. (2004) reportam uma taxa da compressão de até 56% para a heurística que foi utilizada. Também determinam um limite inferior teórico de aproximadamente 37% para o tamanho da malha compactada em relação ao original.

Devido às limitações impostas pelas placas gráficas ao tamanho das texturas, devem ser utilizadas texturas 2D para armazenar as faixas de tetraedros. O método usado por Weiler et al. (2004) consiste em colocar toda uma faixa em apenas uma linha da textura, para que os vértices da faixa possam ser acessados seqüencialmente. Isto resulta em desperdício de memória quando uma faixa não ocupa a linha inteira. Assim, utilizam um algoritmo guloso para tentar maximizar a utilização de cada linha da textura, armazenando mais de uma faixa por linha.

4
Modificações realizadas

Neste capítulo, apresentaremos as modificações realizadas e as escolhas que fizemos para as implementações dos algoritmos VICP (projeção de células) e Traçado de Raios, discutidos no capítulo anterior.

Na seção (4.2.2), é proposta uma forma de utilizar a integração de segmentos lineares, discutida na seção (2.3.2), no algoritmo de Traçado de Raios.

4.1.
Estruturas de dados na placa gráfica

Esta seção trata das estruturas de dados que utilizamos para a implementação dos algoritmos do Capítulo 3. Para o VICP, propomos uma estruturação para que os dados de uma malha de tetraedros sejam armazenados na placa gráfica. O objetivo é eliminar o gargalo na transferência de dados, reportado por Weiler et al. (2002). No caso do Traçado de Raios, que requer estruturas de dados na placa gráfica, utilizamos uma variação das que foram apresentadas na seção (3.2.1), visando obter um balanceamento entre desempenho e custo de armazenamento, com menor complexidade e maior facilidade para realizar modificações do que a estrutura de dados que utiliza faixas de tetraedros.

4.1.1.
Projeção de células (VICP)

Weiler et al. (2003b) reportam que o gargalo do VICP está na transferência de dados para a placa gráfica. Assim, como um trabalho futuro, sugerem que uma das formas de eliminá-lo seria armazenando os dados diretamente na placa gráfica, utilizando, por exemplo, texturas. Isso permitiria inclusive utilizar faixas de triângulos para reduzir o número de vértices enviados por tetraedro. Por outro lado, Moreland & Angel (2004) utilizam a abordagem de Weiler et al. (2002), restrita à projeção ortográfica, e realizam mais cálculos na CPU para reduzir o volume de dados enviados para a placa gráfica (seção 3.1).

Neste trabalho, optamos por explorar mais a programação por fragmentos, armazenando a malha de tetraedros na placa gráfica, conforme sugerem Weiler et al. (2003b). Dessa forma, acreditamos em uma evolução mais rápida dos processadores gráficos (GPU) em relação à CPU, como tem ocorrido recentemente (Lefohn et al., 2004).

A abordagem mais simples seria empregar um vetor contendo, para cada tetraedro, os 12 vértices que devem ser enviados para a placa gráfica (Weiler et al., 2003b). Para desenhar um tetraedro, basta acessar a posição relativa ao primeiro vértice e enviar seqüencialmente os 12 vértices do tetraedro. Um *vertex buffer object* (OpenGL ARB, 2005) pode ser usado para armazenar esse vetor na memória de vídeo, e apenas os índices de cada tetraedro precisam ser enviados para a placa gráfica. Entretanto, o custo de armazenamento dessa abordagem é muito alto.

Por isso, combinamos a utilização de *vertex buffer objects* com estruturas de dados armazenadas em texturas 2D (Tabela 4) na placa gráfica, que podem ser acessadas pelos programas por vértice e por fragmento. É importante notar que isso não exclui uma estrutura de dados na CPU, pois ainda é necessário ordenar os tetraedros de acordo com o observador, o que é feito em CPU.

Os dados mínimos necessários para a renderização de um tetraedro consistem nas posições de seus vértices e nos valores do campo escalar associado. Porém, a ordenação de visibilidade dos tetraedros de uma malha requer uma estrutura de dados que permita extrair eficientemente outras informações, como o conjunto das faces externas ou dos tetraedros adjacentes às faces de um determinado tetraedro. Na literatura, podem ser encontradas diversas estruturas de dados (Garimella, 2002) que atendem a esses requisitos. Celes et al. (2004), por exemplo, propõem uma estrutura de dados para malhas de elementos finitos que é ao mesmo tempo compacta e eficiente. Essa estrutura é a utilizada neste trabalho para o armazenamento de dados na CPU.

Tabela 4 – Texturas 2D utilizadas para o armazenamento de uma malha de tetraedros na placa gráfica, para projeção de células.

Textura	**Coordenadas**		**Dados**			
	u	**v**	**r**	**g**	**b**	**a**
Vértices	k		$\vec{v}_k$			s_k
Normais0	t		$\hat{n}_{t,0}$			$o_{t,0}$
Normais1	t		$\hat{n}_{t,1}$			
Normais2	t		$\hat{n}_{t,2}$			
Normais3	t		$\hat{n}_{t,3}$			$o_{t,3}$
Gradientes	t		$\vec{g}_t$			

Na tabela acima, t é o índice do tetraedro e k é o índice de um vértice; $\vec{v}_k$ e s_k são, respectivamente, a posição e o escalar associado ao vértice k; $\hat{n}_{t,i}$ é a normal da *i-ésima* face do tetraedro, enquanto $o_{t,i}$ é o índice de um vértice pertencente a essa face (apenas dois índices são necessários, pois cada vértice pertence simultaneamente a três faces do tetraedro); e $\vec{g}_t$ é o gradiente do tetraedro.

O vetor de atributos (*vertex buffer object*) dos vértices deverá conter, em cada posição, apenas:

- índice do tetraedro (t);
- índice do vértice (k).

A textura contendo a posição do vértice pode ser acessada no programa por vértice para calcular a direção do raio, necessária no caso da projeção em perspectiva. Os outros dados podem ser acessados pelo programa por fragmento para realizar os cálculos da mesma forma que na abordagem original. Agora, a interseção do raio deve ser computada para todas as quatro faces do tetraedro, pois não é mais possível descobrir qual a face que está sendo desenhada. Por outro lado, como as informações de cada vértice não dependem mais dessa face, pode ser utilizada uma faixa de triângulos para desenhar o tetraedro.

Essa estrutura de dados pode ser facilmente modificada, adicionando-se mais duas texturas de normais para permitir a visualização de hexaedros com

faces planas e campo escalar constante (ou seja, com gradiente nulo) no seu interior. Isto pode ser útil, por exemplo, para a visualização de reservatórios naturais de petróleo, simulados por diferenças finitas.

A memória de textura necessária por cada vértice, assumindo-se o número de vértices como 1/5 do número de tetraedros (Beall & Shephard, 1997), é igual a $(4/5)+4+3+3+4+3=17{,}75$ *floats* $=71$ *bytes*. O espaço por tetraedro para o *vertex buffer object*, considerando-se que serão usadas faixas de triângulos, é igual a 6*2 = 12 *floats* = 48 *bytes*. O custo total é, então, igual a 119 *bytes* por tetraedro, o que ainda consome muita memória, mas muito menos do que os 912 *bytes* necessários quando utilizado apenas o *vertex buffer object* .

4.1.2. Traçado de Raios

Para o algoritmo de Traçado de Raios, optamos por uma variação da estrutura de dados utilizada originalmente por Weiler et al. (2003a). Buscamos uma estrutura de dados mais compacta do que a anterior e, ao mesmo tempo, eficiente e conceitualmente menos complexa do que as faixas de tetraedros (Weiler et al., 2004). A estrutura de dados utilizada, representada como texturas 2D, é ilustrada na Tabela 5.

Tabela 5 – Estrutura de dados utilizada para a implementação do algoritmo de Traçado de Raios.

Textura	**Coordenadas**		**Dados**			
	u	**v**	**r**	**g**	**b**	**a**
Normais0	t		$\hat{n}_{t,0}$			$o_{t,0}$
Normais1	t		$\hat{n}_{t,1}$			$o_{t,1}$
Normais2	t		$\hat{n}_{t,2}$			$o_{t,2}$
Normais3	t		$\hat{n}_{t,3}$			$o_{t,3}$
Gradientes	t		$\vec{g}_t$			$\hat{g}_t$
Adjacências	t		$a_{t,0}$	$a_{t,1}$	$a_{t,2}$	$a_{t,3}$

Na tabela, t é o índice do tetraedro, $(\hat{n}_{t,i}, o_{t,i})$ é a equação do plano da *i-ésima* face do tetraedro, $\vec{g}_t$ e $\hat{g}_t$ são, respectivamente, o gradiente e o termo escalar da eq. (3.11), e $a_{t,i}$ é o índice do tetraedro adjacente à *i-ésima* face do tetraedro. O custo de armazenamento por tetraedro é, então, igual a $4*4+4+4=24\ floats = 96\ bytes$, contra os 144 *bytes* da estrutura de dados original, mas ainda superior aos 76 *bytes* necessários para as faixas de tetraedros, considerando-se que os planos das faces são armazenados.

4.2. Integração de segmentos lineares

Para computar a contribuição de um raio no interior de um tetraedro, Weiler et al. (2003a, 2004) utilizam uma função de transferência pré-integrada, armazenada em uma textura 3D. Moreland & Angel (2004) aplicam a integração de segmentos lineares ao algoritmo VICP, baseando-se na abordagem de Weiler et al. (2002), restrita à projeção ortográfica. Para cada "fatia" de um tetraedro, definida por dois pontos de controle da função de transferência (seção 2.3.2), este é enviado para a placa gráfica com os valores dos pontos de controle como parâmetros adicionais. Assim, as "fatias" são determinadas na CPU e cada tetraedro pode ser enviado diversas vezes para a placa gráfica, que é responsável por "cortá-lo" de acordo com os pontos de controle.

Nesta seção, propomos uma adaptação para realizar a integração de segmentos lineares com todos os cálculos realizados diretamente na GPU. Isso é particularmente interessante para o algoritmo de Traçado de Raios, que requer as estruturas de dados na GPU. Nesta proposta, combinamos a técnica de iso-superfícies em GPU, de Roettger et al. (2000) (seção 2.3.3), com a formulação de Moreland & Angel (2004) (seção 2.3.2) para a resolução da integral de renderização de volume para um segmento linear. Ao aplicarmos essas técnicas ao Traçado de Raios, buscamos obter uma melhor qualidade de imagem, em relação à pré-integração, e permitir a modificação interativa da função de transferência.

Em uma função de transferência composta por segmentos lineares, cada *ponto de controle* representa uma iso-superfície ao longo de um raio que atravessa um tetraedro. Para detectar as iso-superfícies, utilizamos uma textura 2D, como a da seção (2.3.3), mas contendo, em cada posição, apenas duas componentes.

Considerando um campo escalar normalizado ($s \in [0,1]$), os dados de uma posição da textura são:

- o valor da primeira iso-superfície atravessada pelo raio (s_{iso}), ou -1;
- o valor da próxima iso-superfície ($prox\_s_{iso}$), ou -1.

Se, por exemplo, forem consideradas três iso-superfícies: $s_{iso1} = 0{,}25$, $s_{iso2} = 0{,}5$ e $s_{iso3} = 0{,}75$, a textura será, então, codificada como ilustrado na Figura 28.

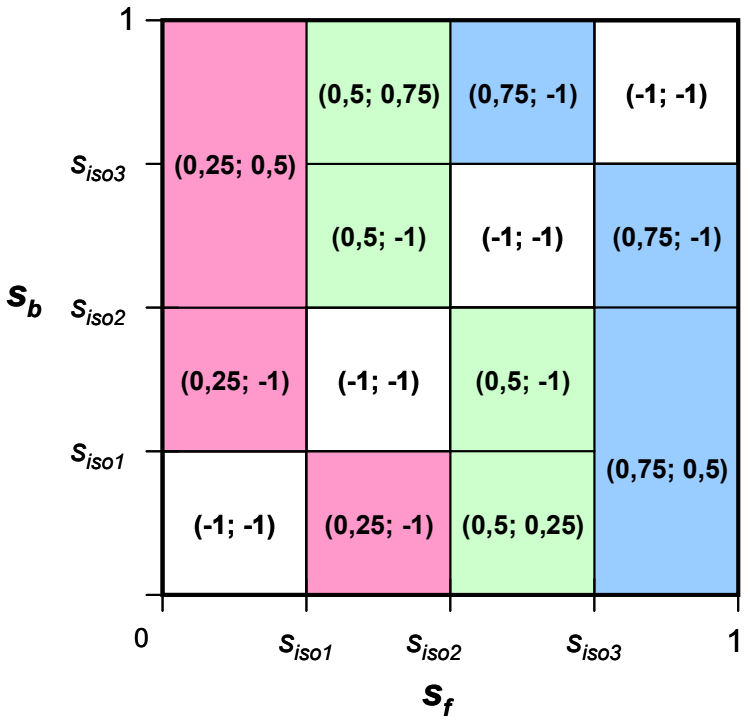


Figura 28 – Textura 2D utilizada para a determinação das iso-superfícies definidas pelos pontos de controle da função de transferência. Em cada posição são armazenados *(s_{iso}; $prox\_s_{iso}$)*, relativos à primeira interseção de um raio entre s_f e s_b, e o valor da próxima iso-superfície. Se s_{iso} for igual a -1, então nenhuma iso-superfície é atravessada.

Os valores dos escalares s_f e s_b, das posições de entrada ($\vec{x}_0$) e saída ($\vec{x}$) do tetraedro, são as coordenadas da textura. Se o valor de s_{iso}, na posição acessada na textura, for -1, então nenhuma iso-superfície foi atravessada pelo raio, o que implica que s_f e s_b estão dentro de um mesmo segmento linear da função de transferência e o raio pode ser diretamente integrado para o segmento. Porém, no caso de haver uma iso-superfície ($s_{iso} \neq -1$), o segmento deverá ser integrado entre s_f e s_{iso}, e os parâmetros passados para o próximo passo do algoritmo devem ser relativos à posição de s_{iso}, ao invés de s_b. Assim, considerando que os

parâmetros recebidos pelo passo atual do raio são (t, λ, R, G, B, A), os parâmetros (t', λ', R', G', B', A'), do próximo passo, podem ser calculados da seguinte forma, onde l é a distância entre $\vec{x}_0$ e $\vec{x}$, e (r,g,b,a) são o resultado da integração do raio entre s_f e s_{iso}:

$t' = t$ (raio permanece no tetraedro atual);

$$\lambda' = \lambda + l\frac{s_{iso} - s_f}{s_b - s_f};$$

$$R' = A * R + (1 - A) * r$$
$$G' = A * G + (1 - A) * g$$
$$B' = A * B + (1 - A) * b$$
$$A' = A + (1 - A) * a$$

O algoritmo pode, então, ser expresso pelo seguinte pseudo-código:

```
1.  siso, prox_siso = Textura2D(sf, sb)
2.  if (siso >= 0)  // Corta iso-superfície.
3.  {
4.    if (sf == siso)
5.      siso = prox_siso;
6.
7.    if (abs(sb - sf) > abs(siso - sf))
8.    {
9.      t' = t;   // Continua no mesmo tetraedro.
10.     l  = l*(siso - sf)/(sb - sf);
11.     lambda' = lambda + l;
12.     sb = siso;
13.   }
14. }
15. r, g, b, a = Integra_Segmento_Linear(sf, sb, l);
16  R', G', B', A' = A*(R, G, B, 1) + (1-A)*(r, g, b, a);
```

Supondo que $s_f = 0{,}9$ e $s_b = 0{,}3$, e considerando as iso-superfícies ilustradas na Figura 28, o resultado da textura será: $s_{iso} = 0{,}75$ e $prox\_s_{iso} = 0{,}5$. O raio deve ser, então, integrado entre 0,9 e 0,75. No passo seguinte, $s_f = 0{,}75$ e $s_b = 0{,}3$. Como s_f já é igual ao valor de uma iso-superfície (linha 4), é preciso avançar para a próxima, que é 0,5. O raio será, então, cortado e integrado entre 0,75 e 0,5. No próximo passo, $s_f = 0{,}5$ e $s_b = 0{,}3$. Novamente, é necessário avançar para a próxima iso-superfície, que é 0,25. Entretanto, $s_b = 0{,}3$. Dessa forma, a iso-supefície não será cortada e o raio será integrado entre 0,5 e 0,3. Isso é determinado pela linha 7 do pseudo-código, que compara os valores absolutos das distâncias $(s_b - s_f)$ e $(s_{iso} - s_f)$. Como $abs(0{,}3 - 0{,}5) < abs(0{,}25 - 0{,}5)$, o raio não

será mais cortado e poderá avançar para o próximo tetraedro. É importante notar que esse teste também funciona para os casos em que, ao avançar para a próxima iso-superfície, o valor seja -1.

5
Comparação e Resultados

Neste capítulo, são apresentados e comparados os resultados obtidos com os algoritmos discutidos ao longo desta dissertação. Os testes foram realizados com as seguintes implementações:

- *VICP (CPU)*: VICP com estrutura de dados apenas na CPU e função de transferência pré-integrada;
- *VICP (GPU)*: VICP com estrutura de dados na placa gráfica, proposta na seção (4.1.1), e função de transferência pré-integrada;
- *Traçado de Raios (Pré-integração)*: Traçado de Raios, com função de transferência pré-integrada;
- *Traçado de Raios (Integração na GPU)*: Traçado de Raios com a integração de segmentos lineares na GPU (seção 4.2).

As duas implementações do algoritmo de Traçado de Raios utilizam a estrutura de dados em GPU descrita na seção (4.1.2).

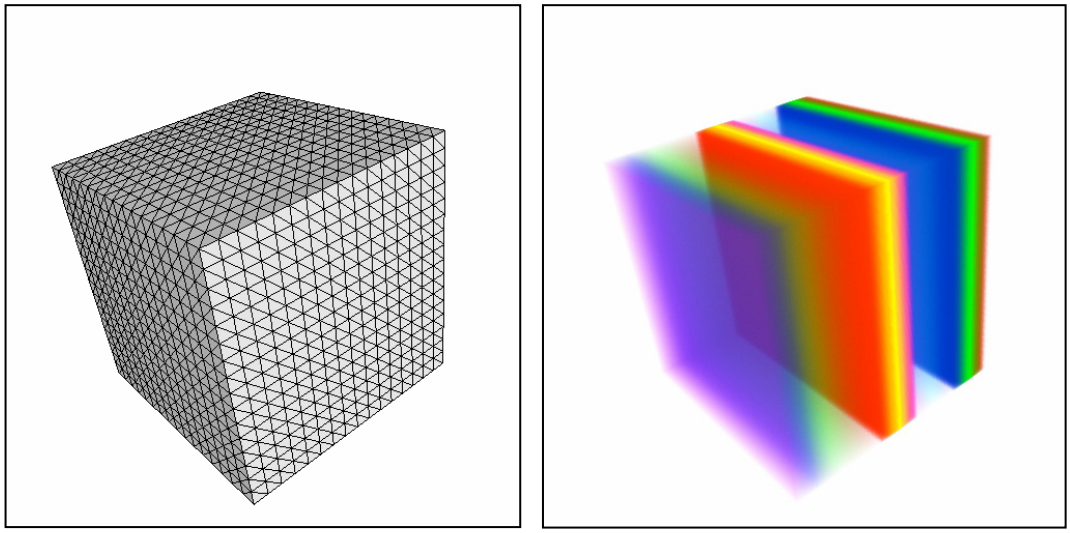
Além do Traçado de Raios, a integração de segmentos lineares da seção (4.2) foi aplicada também ao VICP. Entretanto, apesar do código fonte adicionado ao Traçado de Raios ser quase que diretamente mapeado para o VICP, o desempenho obtido foi insuficiente para permitir a visualização interativa, pois o programa por fragmento passou a apresentar um custo muito elevado. Assim, descartamos a utilização dessa técnica no VICP.

Os modelos utilizados para os testes são descritos na Tabela 6, enquanto as imagens geradas a partir desses modelos são apresentadas nas Figuras 29 a 33.

Tabela 6 – Características das malhas utilizadas para os testes de desempenho.

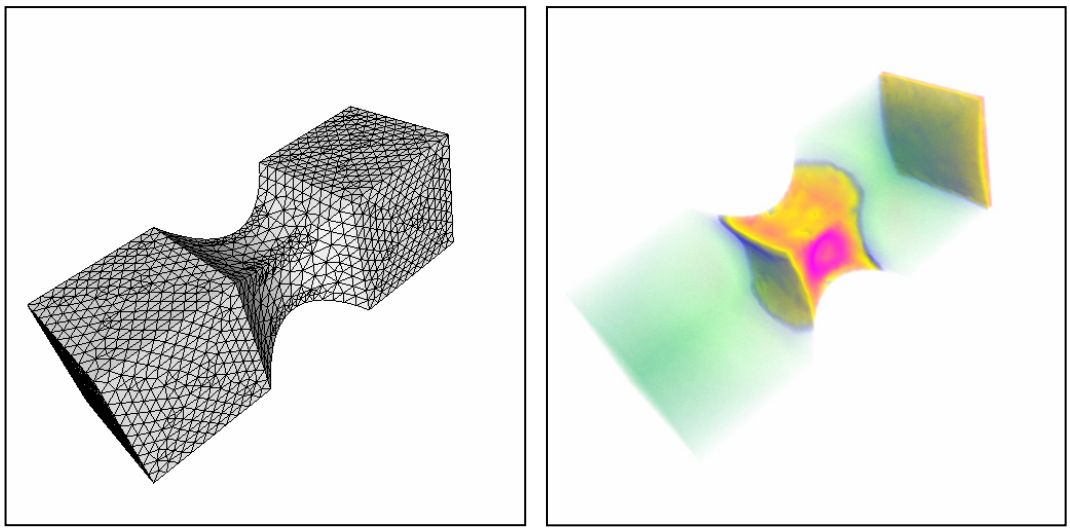
Malha	Num. de tetraedros	Num. de vértices	Propriedade visualizada
Grade 16x16x16	24.576	4.913	Escalar 1
Grade 32x32x32	196.608	35.937	Escalar 1
Barra Fixa	27.691	5.790	Tensão XX
Roda	31.725	6.855	Tensão XX
Bluntfin	224.874	40.960	Densidade
Oxygen	616.050	109.744	Momento X

As grades de 16x16x16 e 32x32x32 (Figura 29) são grades cartesianas, geradas sinteticamente e posteriormente decompostas em tetraedros, com uma propriedade escalar que varia em apenas uma direção. A malha *Barra Fixa* (Figura 30) é um modelo de elementos finitos de uma barra presa por uma das extremidades e com uma força aplicada na direção longitudinal. A *Roda* (Figura 31) também é uma malha de elementos finitos, à qual foi aplicada uma força ao longo de uma direção. *Bluntfin* (Figura 32) (Hung & Buning, 1990), que é disponibilizado pela *NASA Advanced Supercomputing Division*, é uma malha estruturada decomposta em tetraedros lineares. Essa malha apresenta uma curvatura em uma das extremidades e contém resultados da simulação do fluxo de um fluido na direção longitudinal. Finalmente, a malha *Oxygen* (ou *Liquid Oxygen Post*) (Rogers et al., 1986), também disponibilizado pela *NASA Advanced Supercomputing Division*, representa o fluxo de oxigênio líquido ao longo de um disco (Figura 33), com um "poste" cilíndrico posicionado perpendicularmente no centro do disco.



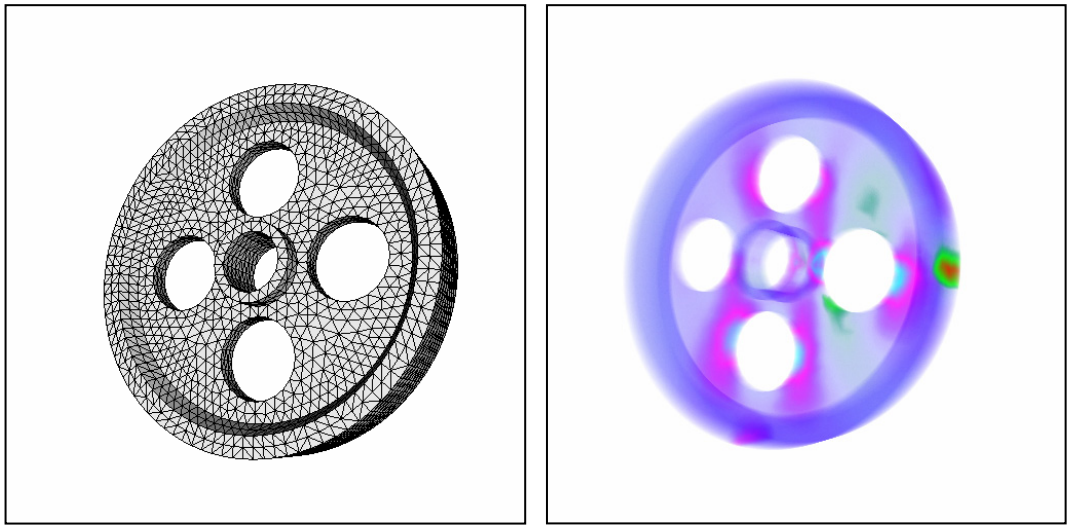
(a) **(b)**

Figura 29 – (a) Imagem da grade de 16x16x16. (b) Visualização volumétrica da grade de 16x16x16.



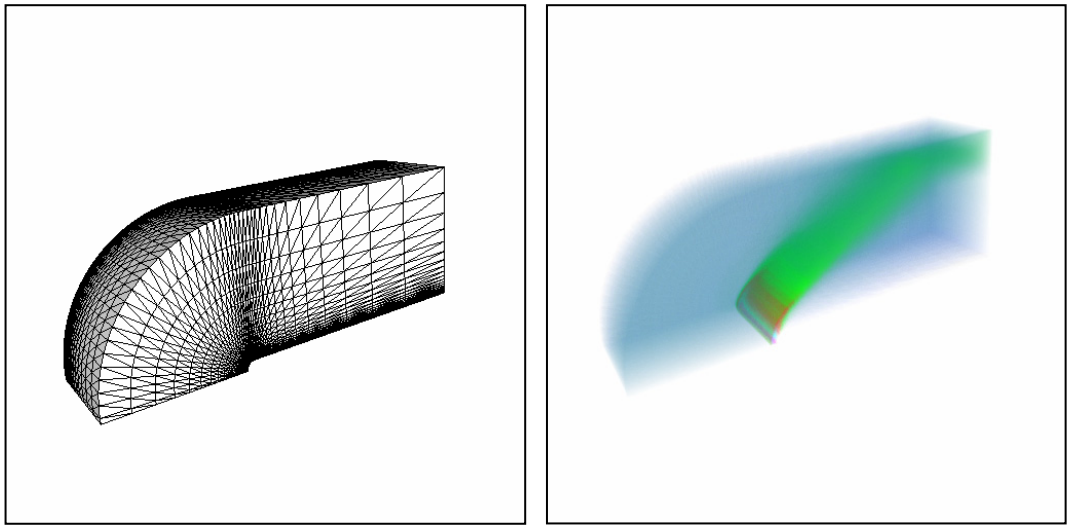
(a) **(b)**

Figura 30 – (a) Imagem da malha *Barra Fixa*. (b) Visualização volumétrica da malhas *Barra Fixa*.



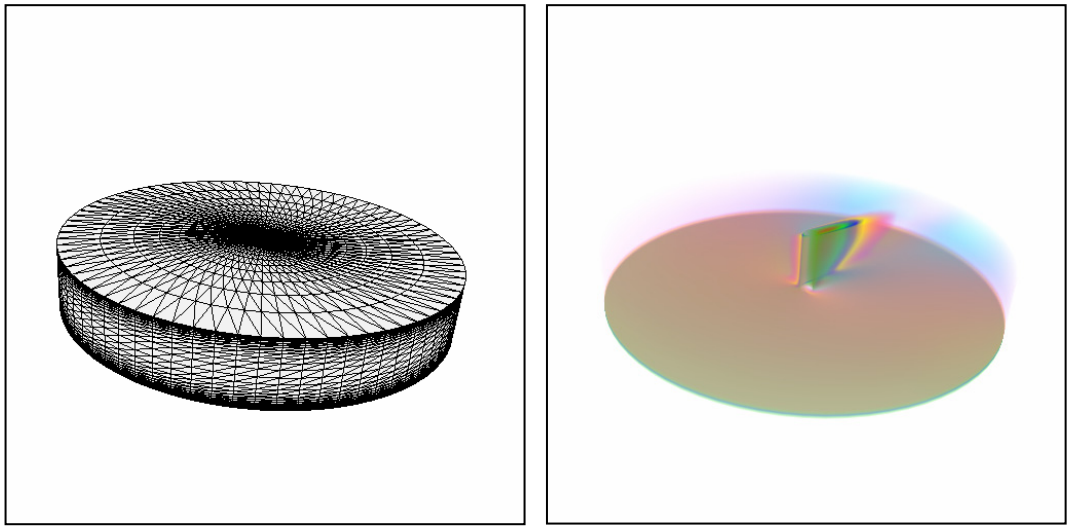
(a) **(b)**

Figura 31 – (a) Imagem da malha *Roda*. (b) Visualização volumétrica da malha *Roda*.



(a) **(b)**

Figura 32 – (a) Imagem da malha *Bluntfin*. (b) Visualização volumétrica da malha *Bluntfin*.



(a) **(b)**

Figura 33 – (a) Imagem da malha *Oxygen*. (b) Visualização volumétrica da malha *Oxygen*.

5.1. Desempenho

Os testes de desempenho foram executados em uma máquina com processador *Intel Pentium 4,* de *2,53GHz* e *512MB* de memória *RAM*. A placa gráfica utilizada foi uma *NVIDIA GeForce 6800 GT*, com *256MB* de memória e *AGP8X*, e os resultados foram medidos para uma janela de 512x512 pixels. As implementações dos programas por vértice e por fragmento foram realizadas utilizando a linguagem *Cg* (NVIDIA, 2004a), com os perfis de programação *vp40* e *fp40*.

Na Tabela 7, são apresentados os resultados para as duas implementações do VICP. Podemos observar que o desempenho do VICP (GPU) foi significativamente melhor que o VICP (CPU), mesmo sendo necessário aumentar o esforço do programa por fragmento com os acessos às texturas da estrutura de dados. Ao armazenarmos a malha na GPU, reduzimos drasticamente o custo da transferência de dados para a placa gráfica. Em nossa implementação, o novo gargalo passou a ser a rasterização. Isso foi comprovado observando-se que, conforme o tamanho da janela foi aumentado, o que implica em um número maior de fragmentos processados, houve queda de desempenho no VICP (GPU), principalmente para as malhas menores. No caso do VICP (CPU), os resultados se mantiveram estáveis. Para uma janela de 800x800, por exemplo, os valores mínimo e máximo de quadros por segundo obtidos para a Barra Fixa, com o VICP

(GPU), reduziram-se a 17,86 e 20,75, respectivamente, em comparação com os valores 24,63 e 26,67, como mostra a Tabela 7. Considerando a taxa atual de evolução da rasterização de fragmentos (Lefohn et al., 2004), acreditamos que, em pouco tempo, será possível obter resultados para janelas maiores equivalentes aos encontrados para a janela de 512x512 que foi utilizada.

Tabela 7 – Comparação dos resultados, em número de quadros por segundo (qd/s) e tetraedros por segundo (tet/s), dos algoritmos VICP (CPU) e VICP (GPU).

Malhas	**VICP (CPU)**				**VICP (GPU)**			
	Min. qd/s	**Máx. qd/s**	**Min. tet/s**	**Máx. tet/s**	**Min. qd/s**	**Máx. qd/s**	**Min. tet/s**	**Máx. tet/s**
Grade 16x16x16	11,85	12,55	291K	308K	14,53	21,32	357K	524K
Grade 32x32x32	1,43	1,51	281K	297K	3,42	3,74	672K	735K
Barra Fixa	9,84	10,67	272K	295K	24,63	26,67	682K	739K
Roda	8,76	9,28	278K	294K	22,08	22,88	700K	726K
Bluntfin	1,26	1,36	283K	306K	3,09	3,50	695K	787K
Oxygen	0,46	0,47	283K	290K	0,98	1,21	604K	745K

Uma desvantagem do VICP (e, em geral, dos algoritmos de projeção de células) é a necessidade de ordenação das células, considerando o modelo óptico de emissão e absorção. Neste trabalho, a ordenação foi realizada em CPU com a heurística MPVONC (Williams, 1992), que, mesmo sendo a mais eficiente extensão do MPVO reportada até o momento (Cook et al., 2004), impõe um limite quanto ao número de células que podem ser ordenadas por segundo. A Tabela 8 mostra os tempos necessários para ordenar algumas malhas de tamanhos variados. Considerando o tempo mínimo para a ordenação da grade 16x16x16, por exemplo, o número máximo de quadros desenhados por segundo é igual a 1 / 0,015 = 66,67, o que não representa um fator limitante para o algoritmo VICP (GPU), que consegue desenhar até 20 quadros por segundo. Entretanto, conforme o tamanho da malha aumenta, o desempenho do VICP (GPU) se aproxima do limite imposto pela ordenação. No caso da malha Oxygen, o número de quadros por segundo que o algoritmo consegue desenhar (1,21) é próximo ao máximo da ordenação (1 / 0,75 = 1,33), o que sugere que a ordenação passa a ser um forte fator limitante. Mesmo que fosse utilizado o algoritmo *Projected Tetrahedra*

(Shirley & Tuchman, 1990), mais eficiente por usar os recursos convencionais das placas gráficas, porém menos flexível, não seria possível transpor o limite imposto pela ordenação.

Tabela 8 – Tempos, em segundos, necessários à ordenação de visibilidade das células de diversas malhas.

Malhas	**Tempo (s)**	
	Min.	**Máx.**
Grade 16x16x16	0,015	0,031
Grade 32x32x32	0,23	0,27
Bluntfin	0,25	0,30
Oxygen	0,75	0,82

No Traçado de Raios, a ordenação de visibilidade é realizada de forma implícita, conforme cada raio é propagado ao longo da malha de tetraedros. Dessa forma, o Traçado de Raios não sofre com o limite imposto pela ordenação das células, como o VICP. Uma comparação entre o VICP (GPU) e o Traçado de Raios (Pré-integração) é apresentada na Tabela 9. Uma otimização possível para o Traçado de Raios consiste em interromper a propagação do raio quando a opacidade acumulada atinge um determinado valor. Porém, essa otimização não foi utilizada para a medição dos resultados da Tabela 9. O Traçado de Raios se mostrou competitivo com VICP, principalmente para as malhas maiores e quase convexas, como Bluntfin e Oxygen, nas quais o limite imposto pela ordenação das células começa a ser relevante. O gargalo do Traçado de Raios se localiza na rasterização, uma vez que quase todas as operações são realizadas no programa por fragmento. Assim, esse é um algoritmo que pode se beneficiar muito com a evolução das placas gráficas.

Uma desvantagem do Traçado de Raios está relacionada ao tratamento de malhas não-convexas. A “descamação” das malhas requer que, para cada camada, sejam desenhadas todas as faces externas. Conseqüentemente, em malhas com muitas concavidades, isso pode implicar em uma significativa perda de desempenho. Porém, para a maioria das malhas reais isso não deve representar um problema.

Tabela 9 – Comparação dos resultados, em número de quadros por segundo (qd/s) e tetraedros por segundo (tet/s), do VICP (GPU) e o Traçado de Raios (Pré-integração), para diversas malhas.

Malhas	VICP (GPU)				Traçado de Raios (Pré-integração)			
	Min. qd/s	Máx. qd/s	Min. tet/s	Máx. tet/s	Min. qd/s	Máx. qd/s	Min. tet/s	Máx. tet/s
Grade 16x16x16	14,53	21,32	357K	524K	8,53	11,63	210K	286K
Grade 32x32x32	3,42	3,74	672K	735K	3,85	5,98	757K	1,18M
Barra Fixa	24,63	26,67	682K	739K	8,76	14,88	243K	412K
Roda	22,08	22,88	700K	726K	7,80	14,53	247K	461K
Bluntfin	3,09	3,50	695K	787K	6,09	8,76	1,37M	1,97M
Oxygen	0,98	1,21	604K	745K	3,40	5,91	2,09M	3,64M

Os resultados dos testes para o Traçado de Raios (Integração na GPU) são comparados na Tabela 10. Foram utilizadas funções de transferência com 10 e 255 segmentos igualmente espaçados em função do campo escalar. Pela tabela, podemos notar que o desempenho desta abordagem foi inferior ao Traçado de Raios (Pré-integração). Isto pode ser explicado pelo aumento no número de passos necessários para a propagação de um raio, que, neste caso, é proporcional ao número de iso-superfícies atravessadas. Quanto maior o número de segmentos lineares da função de transferência, maior será o número de iso-superfícies em um tetraedro e mais segmentos devem ser integrados ao longo do raio. O número de iso-superfícies também depende da variação do campo escalar no interior de cada tetraedro. Entretanto, para malhas de elementos finitos, a tendência é não haver uma grande variação do campo escalar em um tetraedro, uma vez que grandes variações indicam problemas de discretização das malhas.

Embora a diferença de desempenho do Traçado de Raios (Integração na GPU) com 255 segmentos em relação ao Traçado de Raios (Pré-integração) seja significativa, no caso de 10 segmentos a consideramos aceitável. Funções de transferência definidas por poucos segmentos lineares também podem ser muito úteis. Para a visualização de elementos finitos é comum a criação de escalas de cores a partir de poucas cores básicas (2 ou 3, por exemplo), que são interpoladas linearmente. Outros pontos de controle podem ser definidos pelo usuário para

controlar a opacidade dos dados que se deseja visualizar, mas é comum termos poucos segmentos definindo a função de transferência. É importante notar que a integração de segmentos lineares na GPU ainda tem a vantagem de oferecer uma qualidade de imagem superior (seção 5.3) à pré-integração e de não ser restrita a funções de transferência unidimensionais.

Tabela 10 – Comparação dos resultados, em número de quadros por segundo (qd/s) e tetraedros por segundo (tet/s), para o Traçado de Raios (Integração na GPU), com 10 e 255 segmentos, para diversas malhas.

Malhas	Traçado de Raios (Integração na GPU) 10 seg.				Traçado de Raios (Integração na GPU) 255 seg.			
	Min. qd/s	Máx. qd/s	Min. tet/s	Máx. tet/s	Min. qd/s	Máx. qd/s	Min. tet/s	Máx. tet/s
Grade16x16x16	6,15	9,14	151K	225K	2,02	5,76	50K	142K
Grade32x32x32	3,00	5,08	590K	999K	1,71	3,30	336K	649K
Barra Fixa	6,27	11,22	174K	311K	2,41	4,81	67K	133K
Roda	7,35	10,67	233K	339K	3,86	7,19	122K	228K
Bluntfin	3,50	7,11	787K	1,60M	2,98	5,04	670K	1,13M
Oxygen	2,32	4,21	1,43M	2,59M	1,22	2,38	752K	1,47M

5.2. Memória

Para o VICP, não é necessário armazenar dados de uma malha de tetraedros na placa gráfica. Entretanto, utilizando a estrutura de dados em GPU proposta na seção (4.1.1), que ocupa 119 *bytes* por tetraedro, foi possível melhorar o desempenho desse algoritmo de forma significativa. Embora esta seja uma alternativa atraente, ela também apresenta algumas possíveis desvantagens. No VICP (GPU), não podemos descartar a estrutura de dados na CPU, uma vez que ela é necessária para a ordenação das células. Assim, os dados são armazenados de forma redundante. Quando apenas as estruturas de dados armazenadas em CPU são utilizadas, também podem ser implementados algoritmos que alteram a malha dinamicamente, como alguns algoritmos de multi-resolução (*LOD*). No caso de

uma estrutura de dados armazenada na placa gráfica, como a da seção (4.1.1), isso pode se tornar mais difícil, pois as respectivas texturas devem ser atualizadas.

O Traçado de Raios requer uma estrutura de dados armazenada na placa gráfica, como a apresentada na seção (4.1.2), que ocupa 96 *bytes* por tetraedro. Na CPU, pode-se armazenar apenas a lista das faces externas da malha, com alguns atributos adicionais. Dessa forma, uma vantagem do Traçado de Raios é que a replicação de dados é muito menor do que no caso do VICP (GPU).

Mesmo com a estrutura de dados e o controle da renderização quase que completamente na GPU, ainda é possível implementar facilmente algoritmos como planos de corte e visualização de iso-superfícies diretamente na placa gráfica, o que representa um grande potencial dessa abordagem para diversas aplicações. Um plano de corte, por exemplo, pode ser implementado efetivamente "interrompendo" o raio, caso este intercepte o plano ainda no interior do tetraedro.

Além das estruturas de dados, outras informações, como a função de transferência pré-integrada, também ocupam espaço na memória da placa gráfica. Uma textura 3D de 4 componentes (R,G,B,A), com 8 *bits* por componente e dimensões 128x128x128 (necessária para uma boa qualidade da imagem final), ocupa 8.388.608 *bytes*. Se considerarmos a estrutura de dados proposta para o VICP (GPU) na seção (4.1.1), o espaço ocupado equivale a uma malha de $8.388.608/119 \approx 70.493$ tetraedros. Para a estrutura de dados utilizada no Traçado de Raios (seção 4.1.2), esse espaço equivale a $8.388.608/96 \approx 87.381$ tetraedros. Assim, embora a memória ocupada pela função de transferência pré-integrada não represente em si um problema de armazenamento para as placas gráficas modernas, deve-se notar que a memória da placa gráfica pode ser um fator limitante para a visualização de modelos muito grandes, quando utilizada a estruturação de dados na GPU.

5.3. Qualidade de imagem

O VICP (CPU) e o VICP (GPU) deveriam apresentar a mesma qualidade de imagem que o Traçado de Raios (Pré-integração), considerando-se uma mesma função de transferência. No entanto, o Traçado de Raios gerou imagens melhores, principalmente para malhas maiores, como ilustrado na Figura 34. A diferença

pode ser explicada pela precisão utilizada para a composição de cores. No caso do VICP, a contribuição de cada tetraedro foi composta em uma tela de projeção com formato de imagem convencional, de 8 *bits* por componente, uma vez que o driver da placa de vídeo utilizada ainda não suporta completamente formatos mais precisos. Por outro lado, no Traçado de Raios, as cores foram compostas com a precisão interna da placa gráfica (*floats* de 32 *bits* ou *halfs* de 16 *bits*).

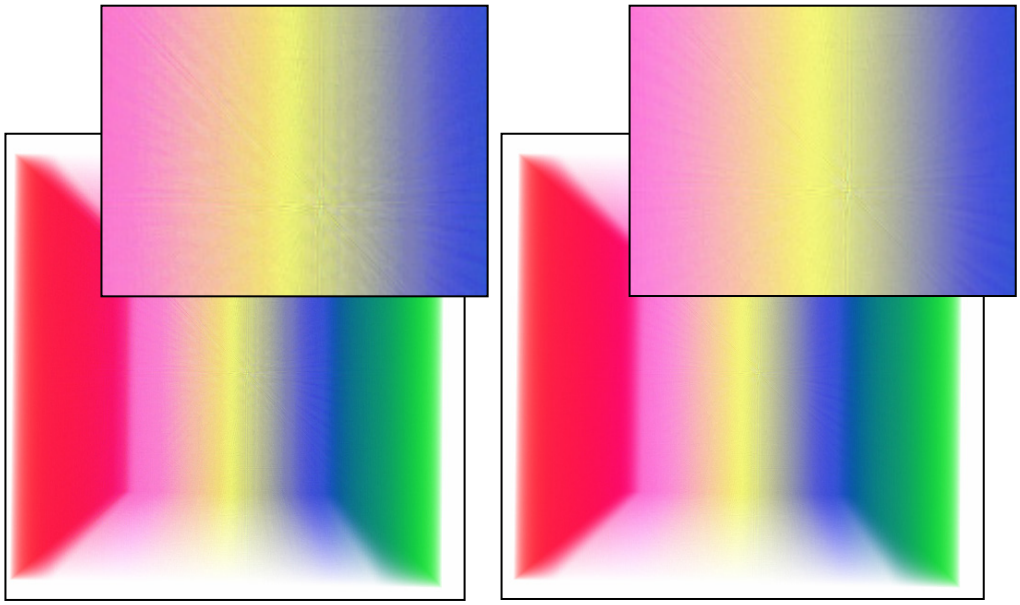
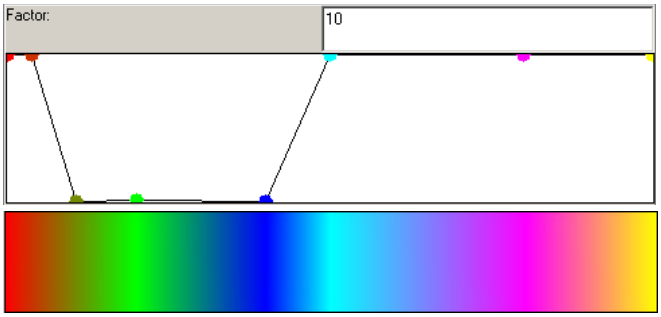
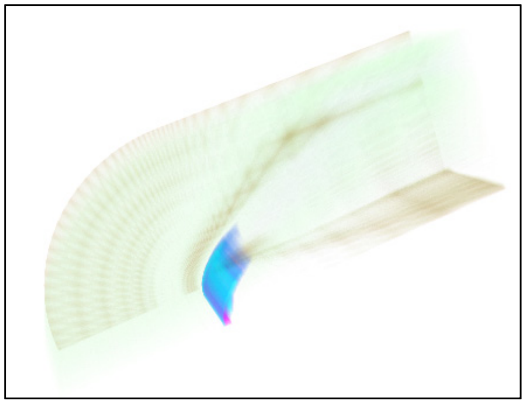


Figura 34 – Diferença entre a qualidade da imagem gerada pelo VICP (a) e Traçado de Raios (Pré-integração) (b), para grade 32x32x32, com a mesma função de transferência, armazenada em uma textura 3D de 128x128x128.

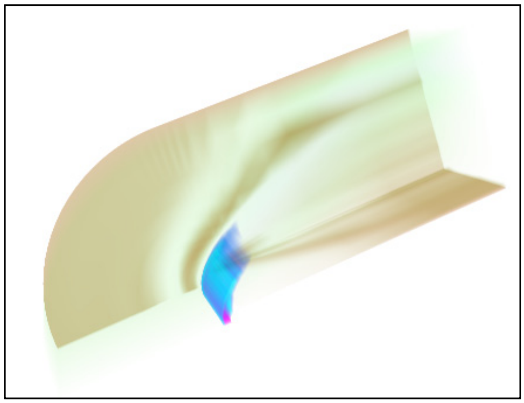
O Traçado de Raios (Integração na GPU) gerou imagens com maior precisão e qualidade superior ao Traçado de Raios (Pré-integração). Em muitas situações, como a ilustrada na Figura 35, a diferença entre as imagens torna-se significativa. Neste caso, a malha Bluntfin, cujo tamanho das células varia muito, apresenta falhas visuais que podem ser notadas em algumas situações. A diferença pode ser atribuída à amostragem finita da função de transferência pré-integrada. Como as coordenadas de textura variam no intervalo [0,1), a distância que o raio percorre em um tetraedro é normalizada em função do comprimento da maior aresta da malha. Tetraedros muito pequenos passam a corresponder a posições muito próximas na textura 3D, enquanto que os maiores correspondem a posições distantes.



(a)



(b)



(c)

Figura 35 – Diferença entre a qualidade da imagem do Traçado de Raios (Pré-integração) e Traçado de Raios (Integração na GPU): (a) função de transferência utilizada; (b) Traçado de Raios (Pré-integração), com a função de transferência pré-integrada e armazenada em uma textura 3D de 128x128x128; (c) Traçado de Raios (Integração na GPU).

No Traçado de Raios também é importante manter o parâmetro da equação de cada raio atualizado a cada passo do algoritmo, com a maior precisão possível. Ao compactarmos esse parâmetro em um tipo *half*, de 16 *bits*, a perda de precisão resultou no problema de amostragem conhecido como *aliasing*, para alguns casos do algoritmo de Traçado de Raios. Esse problema ocorreu para malhas maiores, como Bluntfin e Oxygen, como ilustrado na Figura 36. Com a precisão de 32 *bits*, não observamos nenhuma falha visual relativa ao fenômeno de *aliasing*. Entretanto, esse é um problema que poderá ocorrer em malhas muito maiores do que as que foram visualizadas.

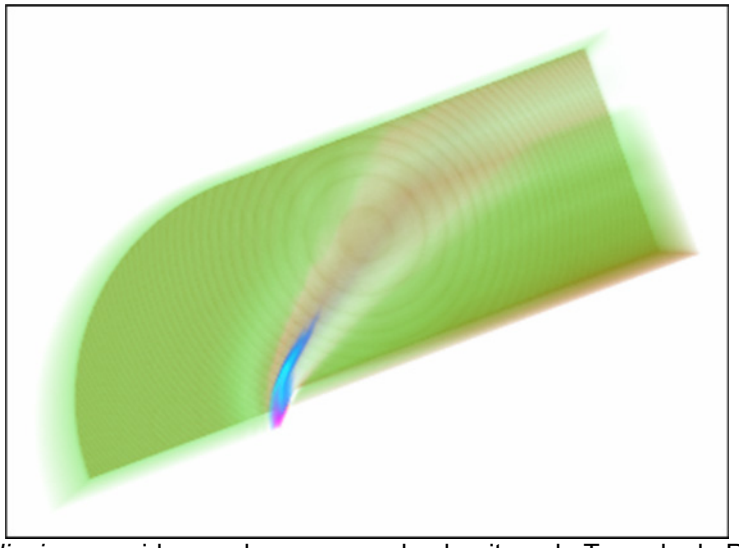


Figura 36 – *Aliasing* ocorrido em alguns casos do algoritmo de Traçado de Raios, para a malha Bluntfin, quando a precisão de 16 *bits* é utilizada para o parâmetro da equação do raio.

5.4. Modificação interativa da função de transferência

Um inconveniente da pré-integração está no tempo necessário para a integração da função de transferência. Quanto maior a textura 3D, maior o tempo de integração e atualização dessa função. Na literatura são encontradas algumas técnicas aproximativas (Roettger & Ertl, 2002; Weiler et al., 2003a) que aceleram dramaticamente este processo. Entretanto, cada uma apresenta vantagens e desvantagens. Entre as possíveis desvantagens, encontra-se a perda de qualidade de imagem para algumas funções de transferência, devido a erros de aproximação. Mas, em geral, apresentam resultados satisfatórios.

Neste trabalho, realizamos a pré-integração da função de transferência com o auxílio da placa gráfica, como proposto por (Roettger & Ertl, 2002). Para isso, foram desenhados n_l retângulos de dimensões n_{sf} e n_{sb}, sendo n_l a dimensão ao longo do comprimento l de um raio que atravessa um tetraedro, e n_{sf} e n_{sb} as dimensões dos valores dos escalares de entrada e saída do raio no tetraedro. Para a integração de cada valor em função de (s_f, s_b, l) foi utilizado o método de integração de segmentos lineares na GPU proposto na seção (4.2). Os resultados de cada retângulo desenhado foram armazenados em uma textura 3D. Com uma textura de dimensões 128x128x128 e 32 segmentos lineares, a pré-integração pôde ser realizada em aproximadamente 1,3s.

Para o algoritmo de Traçado de Raios (Integração na GPU), a atualização da função de transferência é quase instantânea, pois o custo da pré-integração não existe, o que permite a realização de modificações interativas.

6
Conclusão

Neste trabalho, foi estudado o problema da visualização volumétrica interativa de malhas não-estruturadas utilizando renderização direta de volumes. Para isso, foram consideradas duas classes de algoritmos: projeção de células e traçado de raios. As abordagens utilizadas exploraram a programação em placas gráficas, e foi possível observar o seu potencial de aplicação para a área de visualização científica.

Ambos os algoritmos estudados, VICP e Traçado de Raios, aplicados a tetraedros lineares, atenderam com sucesso aos requisitos propostos para o programa Pos3D, que motivou esta pesquisa. Pelos resultados encontrados, os dois algoritmos se mostraram robustos e eficientes para as malhas testadas. Entretanto, o Traçado de Raios superou o VICP em desempenho quando foram utilizadas malhas maiores. Nestes casos, mesmo com a estruturação dos dados na placa gráfica, o que melhorou significativamente o desempenho do VICP, o algoritmo passou a ser limitado pela ordenação das células, realizada em CPU. Quanto à qualidade de imagem, a integração de segmentos lineares na GPU, aplicada ao Traçado de Raios, resultou em imagens mais precisas e de melhor qualidade em comparação com a pré-integração, ao custo de alguma perda de desempenho.

O Traçado de Raios parece ser o algoritmo que mais se beneficiará com a evolução das placas gráficas programáveis. Dessa forma, concluímos que o Traçado de Raios é o principal candidato para ser utilizado futuramente em um programa como o Pos3D.

Com os algoritmos estudados, pudemos explorar um pouco da programação de placas gráficas aplicada à visualização volumétrica de malhas não-estruturadas. Com o desenvolvimento dessas placas, muitas outras formas de visualização de dados poderão ser exploradas.

6.1. Trabalhos futuros

Uma extensão deste trabalho seria na direção da visualização direta de outros tipos de células, isto é, sem decompô-las em tetraedros lineares. Hexaedros lineares e tetraedros quadráticos seriam candidatos naturais a uma continuação da pesquisa realizada.

Como mencionado, os algoritmos utilizados neste trabalho podem ser facilmente estendidos para hexaedros de faces planas e campo escalar constante. As técnicas discutidas nesta dissertação poderiam ser especializadas e aplicadas à visualização volumétrica de malhas de diferenças finitas, como as utilizadas na simulação de reservatórios naturais de petróleo.

A adaptação de algoritmos de multi-resolução (*LOD*) para serem utilizados em conjunto com as técnicas estudadas poderia também ser investigada, de forma a permitir a visualização de malhas muito maiores, com maior interatividade.

Finalmente, outra direção a ser pesquisada seria a utilização dos algoritmos de visualização volumétrica em GPU para a visualização paralela e distribuída de grandes malhas, utilizando agrupamentos de PCs.

7
Referências bibliográficas

BEALL, M. W.; SHEPHARD, M. S. **A general topology-based mesh data structure**, International Journal for Numerical Methods in Engineering, 40(9), p. 1573-1596, 1997.

BERNARDON, F. F.; PAGOT, C. A.; COMBA, J. L. D.; SILVA, C. T.; **GPU-Tiled Ray Casting using Depth Peeling**. Technical Report, SCI Institute, University of Utah, 2004.

BLINN, J. **Light Reflection Functions for Simulation of Clouds and Dusty Surfaces**. ACM SIGGRAPH, Computer Graphics, 16(3), p. 21-29, julho de 1982.

BLINN, J. Jim Blinn's Corner - **Compositing, Part I: Theory**. IEEE Computer Graphics and Applications, 14(5), p. 83-87, 1994.

BUNYK, P.; KAUFMAN, A. E.; SILVA, C. T. **Simple, fast, and robust ray casting of irregular grids**. In: Proceedings of Dagstuhl '97, p. 30-36, 1997.

CARPENTER, L. **The A-Buffer, an Antialiased Hidden Surface Method**. In: Computer Graphics, Proceedings of SIGGRAPH '84, p. 103-108, julho de 1984.

CELES, W.; MARTHA, L. F.; GATTASS, M. **Pós-processador Genérico de Elementos Finitos**. XI Congresso Íbero Latino Americano sobre Mét. Comp. para Engenharia, p. 569-577, 1990

CELES, W.; PAULINO, G. H.; ESPINHA, R. **A compact adjacency-based topological data structure for finite element mesh representation**. Aceito para publicação no International Journal for Numerical Methods in Engineering, julho de 2004.

CHOPRA, P.; MEYER, J. **Incremental Slicing Revisited: Accelerated Volume Rendering Of Unstructured Meshes**. In: Proceedings of IASTED Visualization, Imaging and Image Processing 2002, Malaga, Espanha, setembro de 2002.

CIGNONI, P.; DE FLORIANI, L. **Power Diagram Depth Sorting**. In: Proceedings of the 10th Canadian Conference on Computational Geometry, 1998.

COMBA, J.; DIETRICH, C.; PAGOT, C.; SCHEIDEGGER, C. **Computation on GPUs: from a programmable pipeline to an efficient stream processor**. Revista de Informática Teórica e Aplicada, v. 10, n. 1, p. 41-70, 2003.

COMBA, J., KLOSOWSKI, J.; MAX, N.; MITCHELL, J. S. B., SILVA, C.; WILLIAMS, P. **Fast Polyhedral Cell Sorting for Interactive Rendering of Unstructured Grids**. Computer Graphics Forum. Vol. 18, p. 367-376, 1999.

COOK, R.; MAX, N.; SILVA, C. T.; WILLIAMS, P. L. **Image-Space Visibility Ordering for Cell Projection Volume Rendering of Unstructured Data**. In: IEEE Transactions on Visualization and Computer Graphics. Vol. 10, No. 6, novembro de 2004.

DE BERG, M.; VAN KREVELD, M.; OVERMARS, M.; SCHWARZKOPF, O. **Computational Geometry**. 2ª edição, Springer-Verlag, Berlim: 2000.

DREBIN, R. A.; CARPENTER L.; HANRAHAN P. **Volume rendering**. In: Proceedings of SIGGRAPH '88, 1988. p. 65-74.

ENGEL, K.; KRAUS, M.; ERTL, T. **High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading**. Proceedings of Eurographics/SIGGRAPH Workshop on Graphics Hardware '01, pp.9-16, 2001.

EVERITT, C.; **Interactive Order-Independent Transparency**. Technical Report. http://www.nvidia.com/object/Interactive_Order_Transparency.html. 2001. Acesso em: 18 de fevereiro de 2005.

FARIAS, R.; MITCHELL, J.; SILVA, C. ZSWEEP: **An Efficient and Exact Projection Algorithm for Unstructured Volume Rendering**. In: Proceedings 2000 Volume Visualization Symposium, pp. 91-99, outubro de 2000.

FOLEY, J. D.; DAM, A. van; FEINER, S. K.; HUGHES, J. F. **Computer Graphics: Principles and Practice**. Addison-Wesley, Reading, MA, second edition, 1997.

GARIMELLA, R. V., **Mesh data structure selection for mesh generation and FEA applications**. In: International Journal for Numerical Methods in Engineering, 55:451-478, 2002.

GARRITY, M. P. **Raytracing Irregular Volume Data**. In: Proceedings of the 1990 Workshop on Volume Visualization, ACM Press, p. 35-40, 1990.

GUTHE, S., ROETTGER, S., SCHIEBER, A., STRASSER, W., ERTL, T. **High-quality unstructured volume rendering on the PC platform**. In: Proceedings of ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, Eurographics Association, p. 119-125, 2002.

HADWIGER, M.; KNISS, J. M.; ENGEL K.; REZK-SALAMA C. **High-Quality Volume Graphics on Consumer Graphics Hardware**. Course Notes 42, SIGGRAPH'2002, San Antonio: 2002.

HUNG, C. M.; BUNING, P. G. **Blunt Fin data set**. 1990. http://www.nas.nasa.gov/Research/Datasets/Hung/index.shtml. Acesso em: 17 de fevereiro de 2005.

KINDLMANN, G.; DURKIN, J. W. **Semi-automatic generation of transfer functions for direct volume rendering**. In: Proceedings of the 1998 IEE Symposium on Volume Visualization, p. 79-86, outubro de 1998.

KNISS, J.; KINDLMANN, G.;HANSEN, C. **Multidimensional Transfer Functions for Interactive Volume Rendering**. In: IEEE Transactions on Visualization and Computer Graphics. Vol. 8, no. 3, p. 270-285, julho de 2002.

KRAUS, M.; ERTL, T.; **Cell Projection of Cyclic Meshes**. In Proceedings of IEEE Visualization 2001. p. 215-222, 2001.

KRÜGER, W. **The Application of Transport Theory to Visualization of 3D Scalar Data Fields**. In: Computational Physics, Vol. 5, No. 4, p. 397-406, 1991.

LEFOHN, A.; BUCK, I.; OWENS, J. D.; STRZODKA, R. **GPGPU: General-Purpose Computation on Graphics Processors**. In: Proceedings of IEEE Visualization 2004. Tutorial no. 3. Outubro de 2004.

MALTA, I.; PESCO, S.; LOPES, H. **Cálculo a uma variável – Volume II – Derivada e Integral**, Rio de Janeiro: Editora PUC-Rio, 2002.

MAX, N. **Optical models for direct volume rendering**. In: IEEE Transactions on Visualization and Computer Graphics, 1(2):99-108, 1995.

MORELAND, K.; ANGEL, E. **A Fast High Accuracy Volume Renderer for Unstructured Data**. Proceedings of IEEE Symposium on Volume Visualization and Graphics 2004, p. 9-16, Austin, Texas, USA, outubro de 2004.

NAS – NASA Advanced Supercomputing Division. http://www.nas.nasa.gov. Acesso em 17 de fevereiro de 2005.

NOVINS, K.; ARVO, J. **Controlled Precision Volume Integration**. 1992 Workshop on Volume Visualization, ACM, 1992.

NVIDIA CORPORATION. **Cg Toolkit User's Manual: A Developer's Guide to Programmable Graphics**. Release 1.2, janeiro de 2004. Disponível em: http://www.nvidia.com/object/cg_toolkit.html. Acesso em 24 jan. 2005.

NVIDIA CORPORATION. **NVIDIA GPU Programming Guide Version 2.2.1**. NVidia Corporation, novembro de 2004. Disponível em: http://www.nvidia.com/object/gpu_programming_guide.html. Acesso em: 20 jan. 2005.

OPENGL ARB. **OpenGL® - The Industry Standard for High Performance Graphics**. http://www.opengl.org. Acesso em 24 jan. 2005.

PAIVA, A. C.; SEIXAS, R. B.; GATTASS, M. **Introdução à Visualização Volumétrica**. Departamento de Informática, PUC-Rio, Inf. MCC03/99, Rio de Janeiro, 1999.

PORTER T.; DUFF T. **Compositing Digital Images**. Computer Graphics (Proceedings of SIGGRAPH, Col. 18, No. 3, p. 253-259, julho de 1984.

REMACLE, J. F.; SHEPHARD, M. S. **An algorithm oriented mesh database**. In: International Journal for Numerical Methods in Engineering, 58, p. 349-374, 2003.

ROETTGER, S.; KRAUS, M.; ERTL, T. **Hardware-accelerated volume and isosurface rendering**. In: Proceedings of Visualization '00, p. 109-116, 2000.

ROETTGER, S.; ERTL, T. **A two-step approach for interactive pre-integrated volume rendering of unstructured grids**. In: Proceedings of the 2002 IEEE Symposium on Volume Visualization and Graphics, p. 23-28, 2002.

ROGERS, S. E.; KWAK, D.; KAUL, U. **Liquid Oxygen Post**. Data set. 1986. http://www.nas.nasa.gov/Research/Datasets/Rogers/index.shtml. Acesso em: 17 de fevereiro de 2005.

SHIRLEY P.; TUCHMAN A. **A Polygonal Approximation to Direct Scalar Volume Rendering**. Computer Graphics, 24(5), p. 63-70, dezembro de 1990.

SILVA, P. M. **Visualização Volumétrica de Horizontes em Dados Sísmicos 3D**. Tese de Doutorado em Informática, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, julho de 2004.

SILVA, C.; MITCHELL, J. S. B.; WILLIAMS, P. **An Exact Interactive Time Visibility Ordering Algorithm for Polyhedral Cell Complexes**. In: Proceedings of ACM Symposium on Volume Visualization, p. 87-94, outubro de 1998.

SPERAY, D.; KENNON S. **Volume Probes: Interactive Data Exploration on Abitrary Grids**. Proceedings of San Diego Workshop on Volume Visualization, Computer Graphics, 24(5), p. 5-12, dezembro de 1990.

STEIN, C.; BECKER, B.; MAX, N. **Sorting and Hardware Assisted Rendering for Volume Visualization**. In: Proceedings of SIGGRAPH Symposium on Volume Visualization, p. 83-90, outubro de 1994.

WEILER, K. **The Radial Edge Structure: A Topological Representation for Non-Manifold Geometric Boundary Modeling**. Geometric Modeling for CAD Applications, p. 3-36, 1988.

WEILER, M.; KRAUS, M.; ERTL, T. **Hardware-Based View-Independent Cell Projection**. In: Proceedings of IEEE Symposium on Volume Visualization 2002, p. 13-22, 2002.

WEILER, M.; KRAUS, M.; MERZ, M.; ERTL, T. **Hardware-Based Ray Casting for Tetrahedral Meshes**. In: Proceedings of IEEE Visualization ’03, p. 333-340. IEEE, 2003.

WEILER, M.; KRAUS, M.; MERZ, M.; ERTL, T. **Hardware-Based View-Independent Cell Projection**. In: IEEE Transactions on Visualization and Computer Graphics (Special Issue on IEEE Visualization 2002), 9(2), p. 163-175, junho de 2003.

WEILER, M.; MALLÓN, P. N.; KRAUS, M.; ERTL, T. **Texture-Encoded Tetrahedral Strips**. In: Proceedings Symposium on Volume Visualization 2004, p. 71-78, 2004.

WILHELMS, J.; VAN GELDER, A.; **A Coherent Projection Approach for Direct Volume Rendering**. IEEE Computer Graphics, vol. 25, no. 4, p. 275-284, julho de 1991.

WILLIAMS, P. **Visibility Ordering Meshed Polyhedra**. ACM Transactions on Graphics, 11(2), p. 103-125, abril de 1992.

WILLIAMS, P.; MAX, N. **A volume density optical model**. Proceedings of the 1992 Workshop on Volume Visualization, p. 61-68. Boston, outubro de 1992.

WILLIAMS, P. L.; MAX, N. L.; STEIN, C. M. **A High Accuracy Volume Renderer for Unstructured Data**. IEEE Transactions on Visualization and Computer Graphics, Vol. 4, No. 1, p. 37-54, março de 1998.

WITTENBRINK, C. **R-Buffer: A Pointerless A-Buffer Hardware Architecture**. In: Proceedings of ACM-Eurographics Workshop on Graphics Hardware, p. 73-80, 2001.

WITTENBRINK, C. M.; MALZBENDER, T; GOSS, M. E. **Opacity-Weighted Color Interpolation for Volume Sampling**. In: Proceeding of the 1998 Symposium on Volume Visualization, p. 135-142, outubro de 1998.

WYLIE, B.; MORELAND, M.; FISK, L. A.; CROSSNO, P. **Tetrahedral Projection using Vertex Shaders**. In: Proceedings of the IEEE Symposium of Volume Visualization 2002, p. 13-22, 2002.

YAGEL, R.; REED, D. M.; LAW, A.; SHIH, P.; SHAREEF, N. **Hardware Assisted Volume Rendering of Unstructured Grids by Incremental Slicing**. In: ACM Symposium of Volume Visualization ’96. p. 55-63, 1996.