Projeção de Células baseada em GPU para Visualização Interativa de Volumes

UFRJ

Dissertação submetida para a obtenção do título de

Mestre em Ciências em Engenharia de Sistemas e Computação

ao Programa de Pós-Graduação de Engenharia de Sistemas e Computação

da COPPE/UFRJ

por

André de Almeida Maximo

Novembro de 2006

PROJEÇÃO DE CÉLULAS BASEADA EM GPU PARA VISUALIZAÇÃO INTERATIVA DE VOLUMES

André de Almeida Maximo

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

Aprovada por:

Prof. Ricardo Cordeiro de Farias , Ph.D.

Prof. Claudio Esperança, Ph.D.

Prof. Waldemar Celes Filho, Ph.D.

RIO DE JANEIRO, RJ - BRASIL

NOVEMBRO DE 2006

MAXIMO, ANDRÉ DE ALMEIDA

Projeção de Células baseada em GPU para Visualização Interativa de Volumes [Rio de Janeiro] 2006

XII, 62 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2006)

Dissertação - Universidade Federal do Rio de Janeiro, COPPE

1. Visualização Volumétrica
2. Programação em Placa Gráfica
3. Projeção de Células

I. COPPE/UFRJ II. Título (série)

Aos meus pais, que sempre me apoiam

E a toda minha família.

Agradecimentos

Gostaria de agradecer a todos que, direta ou indiretamente, contribuíram para a conclusão deste trabalho.

Aos professores do LCG: Ricardo Farias, Claudio Esperança, Paulo Roma e Antônio Oliveira. Pelas dúvidas sanadas e apoio sempre presente. Principalmente ao meu orientador Ricardo Farias, amigo e mestre para todas as horas e todos os assuntos.

A todos os meus amigos(as) do LCG que acompanharam junto comigo as idas e vindas do mestrado: Álvaro, Caique, Disney, Saulo, Vitor, Ricardo, Yalmar, Guina, Karl, Okamoto, Wagner, Alexandre, Max, Daniel, Elisabete, Diego, Djeisson, Flávio, Jonas, Luis, Narciso e Pilato. Pelas conversas, discussões e conselhos necessários academicamente e pessoalmente. Principalmente ao meu amigo Ricardo Marroquim, aluno de doutorado do LCG, pela grande ajuda no decorrer deste trabalho.

Agradeço aos meus amigos(as) de infância, de muitos anos e os conhecidos recentemente: André, Anderson, Nelson, Adriana, Blay, Wilson, Alexandre, Eneida, Emilian, Danilo, Maise, Jonice, Rafael, Flávio, Gárcia, Letícia, Amanda, Targino, Grazziela, Thatiana, Rogea, Daniel, Glauco, Melissa, Ana Luisa e Ana Letícia. Pela força, amizade e incentivo no decorrer dos últimos anos.

Aos meus professores do ensino médio e fundamental: Ana Cristina, Elaine, Maida, Vitor, Kátia e Luís Sérgio. Pelos meus primeiros e importantes conhecimentos.

Agradeço aos responsáveis pelos momentos de lazer: Escravos da Mauá, Companhias Aéreas, Belmonte, Cinemark, Wizards of the Coast e escritores de bons livros. Momentos esses importantes para a continuidade do trabalho.

Aos meus familiares: Vó Nazita, Michel, Fabiano, Andréia, Cristiano, Tia Terezinha, Tia Celinha, Tio Tercílio, Tio Tuninho, Tia Eliana, Tia Rita, Paulinha,

Vanessa, Tio Olivete, Tia Iaia, Tio Solimar, Tia Zilá e Tia Maria. Por toda a ajuda e importante presença na minha vida.

Finalmente agradeço aos meus irmãos: Mário e Bárbara. E meus pais: Paulo e Magda. Pelo apoio e confiança sempre presentes. Ambos muito importantes para a conclusão desta tese.

Rio de Janeiro, Novembro de 2006

André de Almeida Maximo

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

PROJEÇÃO DE CÉLULAS BASEADA EM GPU PARA VISUALIZAÇÃO INTERATIVA DE VOLUMES

André de Almeida Maximo

Novembro/2006

Orientador: Ricardo Cordeiro de Farias

Programa: Engenharia de Sistemas e Computação

Nesta dissertação é apresentada uma abordagem prática do algoritmo de Projeção de Tetraedros (PT) para visualização interativa de dados volumétricos não-estruturados usando placas gráficas programáveis. Ao contrário de trabalhos similares apresentados recentemente, o método proposto emprega dois *shaders* de fragmentos, um para a computação das projeções de tetraedros e outro para a visualização do volume. O algoritmo proposto alcança taxas interativas por guardar o modelo em memória de textura e evitar projeções redundantes das implementações anteriores usando shaders de vértices. O algoritmo é capaz de visualizar mais de 2 milhões de tetraedros por segundo nas placas gráficas atuais, fazendo-o competitivo com abordagens recentes de traçado de raios, enquanto ocupa um espaço de memória substancialmente menor.

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

GPU-BASED CELL PROJECTION FOR INTERACTIVE VOLUME RENDERING

André de Almeida Maximo

November/2006

Advisor: Ricardo Cordeiro de Farias

Department: Systems Engineering and Computer Science

In this dissertation is presented a pratical approach of the Projected Tetrahedra's (PT) algorithm for interactive volume rendering of unstructured data using programmable graphics cards. Unlike similar works reported earlier, the proposed method employs two fragment shaders, one for computing the tetrahedra projections and another for rendering the volume. The proposed algorithm achieve interactive rates by storing the model in texture memory and avoiding redundant projections of the earlier implementations using vertex shaders. The algorithm is capable of rendering over 2 millions tetrahedra per second on current graphics hardware, making it competitive with recent ray-casting approaches, while occupying a substantially smaller memory footprint.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

"Your vision will become clear
only when you look into your heart.
Who looks outside, dreams.
Who looks inside, awakens."

– Carl Jung

Visualização volumétrica consiste de uma série de técnicas para analisar dados volumétricos e extrair do interior dos mesmos, informações significantes [1, 2]. Exemplos da visualização desses dados existem na medicina, geologia, indústria e engenharia.

Dados volumétricos contém informações tridimensionais adquiridas de diferentes tipos de fontes. Existem três tipos de fontes de dados volumétricos:

- *Dados amostrados* de objetos ou fenômenos reais;
- *Dados computados* produzidos por simulação computacional;
- *Dados modelados* gerados por um modelo geométrico.

Esta dissertação visa abordar métodos de visualização volumétrica, discutindo suas vantagens e desvantagens. Será apresentado um algoritmo que possibilite a visualização interativa de dados volumétricos.

1.1 Dados Volumétricos

Computação gráfica é usada atualmente em diferentes áreas da indústria, comércio, governo, educação e entretenimento [3]. Suas aplicações diferem nos tipos de objetos a serem representados e o tipo de imagem a ser gerada. A geração de imagens pode referir-se à superfície ou ao volume dos objetos representados, veja a Figura 1.1.

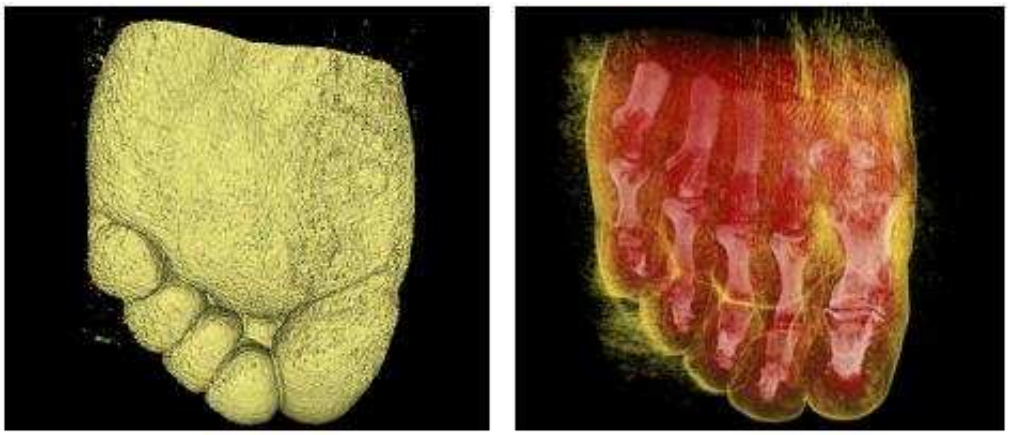

Figura 1.1: Diferença entre a visualização da superfície de um objeto (à esquerda) e do volume (à direita) [4].

A técnica de visualização volumétrica a ser utilizada depende do tipo de dado volumétrico a ser analisado. Os dados podem ser divididos pela forma como são armazenados. Existem duas definições diretas para os diferentes tipos de dados: *dados regulares* são aqueles com topologia implícita; enquanto que *dados irregulares* possuem topologia explícita.

Dados volumétricos *amostrados* são aqueles adquiridos por um processo de captura. Exemplos de equipamentos que geram dados amostrados são:

- Ressonância Magnética (MRI – *Magnetic Resonance Imaging*);
- Tomografia Computadorizada (CT – *Computed Tomography*);
- Ultra-som (*Ultrasound*);
- Microscopia Focal (CLSM – *Confocal Laser Scanning Microscopy*);
- Escaneadores industriais (*Industrial Scanners*).

Dados volumétricos *computados* são aqueles produzidos por simulação, tipicamente executadas em super-computadores. Alguns exemplos de aplicações que geram dados computados são:

- Metereologia – predição do tempo;
- Dinâmica de fluídos – simulação de fluxos;
- Engenharia mecânica – testes de novos materiais.

Existe uma abordagem chamada de gráficos volumétricos (*volume graphics*) que explora as vantagens das técnicas volumétricas não só para visualização como também para modelagem e manipulação. Dados gerados por esta abordagem são chamados de *modelados*.

Dados adquiridos por um desses três processos, são formados por uma seqüencia de imagens ou fatias (*slices*). Cada fatia é tipicamente um conjunto S de amostras (x, y, z, v), representando os valores v de alguma propriedade dos dados no ponto (x, y, z), um ponto no espaço $\Re^3$ (podendo também variar no tempo). Os dados podem representar um campo **escalar**, com v representando, por exemplo: densidade, temperatura, etc; veja a Figura 1.2. Os valores v podem ser vetores representando um campo **vetorial**, por exemplo: velocidade, fluxo, corrente marítima, etc; veja a Figura 1.3.

Nas Figuras 1.2 e 1.3 os dados estão dispostos em uma matriz tridimensional, também chamada de *volume buffer*, *3D raster* ou simplesmente *volume*, com valores escalares s_{ijk} ou vetoriais $\vec{v}_{ijk}$. Neste trabalho de pesquisa é estudado a visualização de dados escalares e, portanto, os valores são referidos como s (*scalar*) no lugar de v (*vetor*).

Em geral, as amostras podem ser obtidas de quaisquer ponto do espaço, mas na maioria dos casos S é *isotrópico*, contendo amostras obtidas em intervalos regulares do espaço ao longo dos três eixos ortogonais. Visto que S é definido em uma grade **regular** (*regular grid*), uma matriz tridimensional é tipicamente usada para armazenar os seus valores. A região de valor constante que cerca cada amostra é conhecida como célula do volume (*volume cell*).

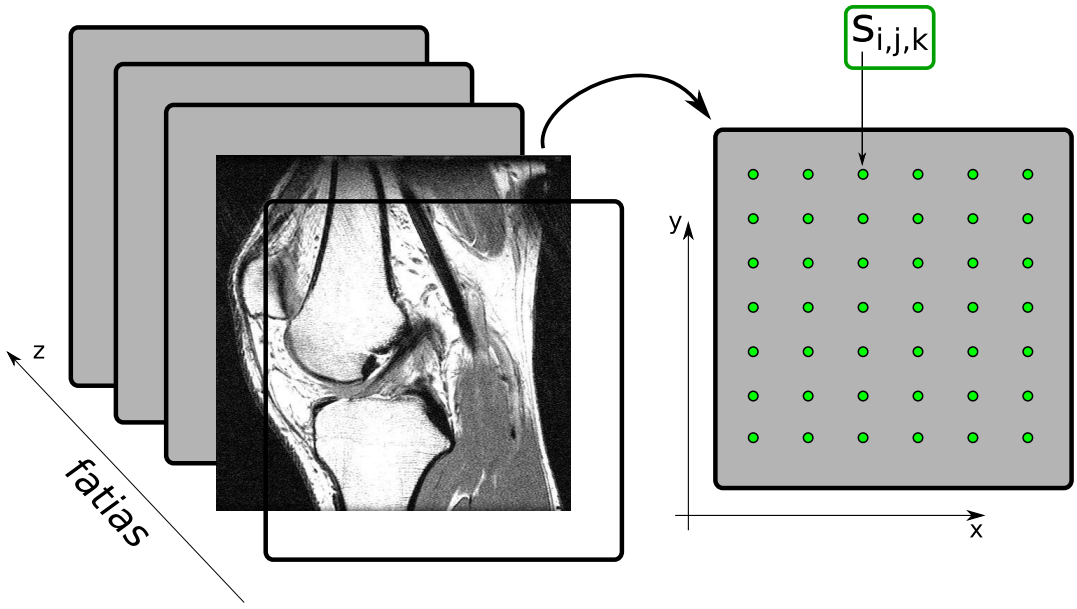

Figura 1.2: Visualização de um campo escalar. As fatias são dados amostrados da densidade de um joelho.

Em adição às grades regulares, grades **retilíneas**, **curvilíneas** e **não-estruturadas** são empregadas. Em uma grade retilínea (*rectilinear grid*) as células são alinhadas com os eixos, mas o espaçamento na grade, ao longo dos eixos, pode ser arbitrária. Quando tal grade é transformada de forma não-linear enquanto preserva a topologia, a grade torna-se curvilínea (*curvilinear grid*), também chamada de grade estruturada (*structured grid*). Usualmente, a grade retilínea definindo uma organização lógica é chamada de espaço computacional (*computational space*) e a grade curvilínea é chamada de espaço físico (*physical space*). De outra forma, a grade é chamada de não-estruturada (*unstructured grid*) ou *irregular*. Um dado volumétrico não-estruturado ou irregular é uma coleção de células cuja conectividade deve ser explicitamente fornecida. Essas células podem ter um formato arbitrário, como tetraedros, hexaedros ou prismas.

O algoritmo proposto nesta dissertação tem como objetivo visualizar dados escalares não-estruturados dispostos em tetraedros, advindos de amostras ou simulação.

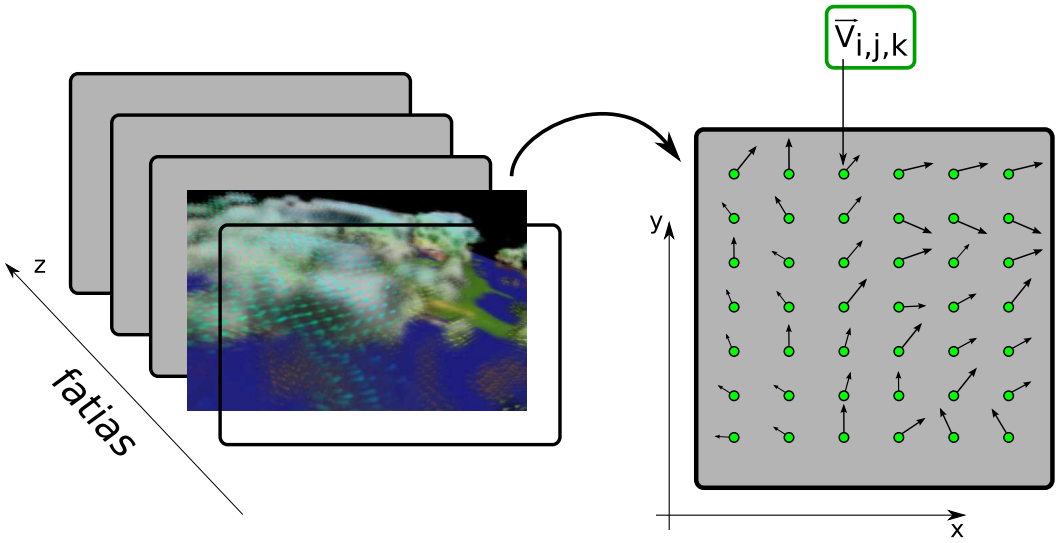

Figura 1.3: Visualização de um campo vetorial [5]. As fatias são dados computados para simulação de efeitos atmosféricos.

1.2 Visualização Volumétrica

Segundo Lichtenbelt et al. [6], o termo **visualização volumétrica** [1] se refere a: "um método para mostrar dados volumétricos como imagens bidimensionais."

A visualização volumétrica pode ser empregada em várias áreas para os mais diversos fins, tais como:

- Imagens médicas – auxiliar no diagnóstico e estudo de doenças;
- Geologia – visualização e exploração de recursos naturais;
- Paleontologia – descoberta de fósseis;
- Análise Microscópica – análise biológica e estudo de substâncias;
- Dinâmica de fluidos – pesquisa do comportamento físico de gases e líquidos;
- Indústria – inspeção interna não-destrutiva de materiais compostos ou partes mecânicas;
- Metereologia – previsão do tempo e estudo de fenômenos naturais;
- Engenharia Civil – testar e analisar a resistência de estruturas.

[1] Visualização volumétrica (*volume visualization*) é também conhecida como *volume rendering*.

Ao longo dos anos muitas técnicas foram desenvolvidas para visualizar dados volumétricos. Inicialmente os métodos envolviam aproximar o volume à superfícies contidas dentro dos dados. Visualizar superfícies facilita o processo de visualização volumétrica, pois requer apenas o uso de primitivas geométricas que são suportadas pelas placas gráficas. Quando uma visualização de superfície é empregada, uma dimensão de informação é essencialmente perdida. Em resposta a esse problema, técnicas de visualização do volume foram desenvolvidas. Essas técnicas têm como objetivo representar dados 3D em uma simples imagem 2D projetada diretamente de um dado volumétrico 3D.

Uma imagem gerada utilizando-se uma técnica de visualização do volume apresenta mais informações sobre os dados do que a visualização da superfície, mas ao custo de uma maior complexidade do algoritmo e, conseqüentemente, maior tempo computacional. Em busca de maior desempenho na geração de imagens, vários algoritmos de visualização volumétrica foram desenvolvidos e otimizados, e máquinas específicas para este fim foram projetadas.

Visualização volumétrica pode ser dividida em três abordagens principais:

- Espaço do objeto;
- Espaço da imagem;
- Métodos de domínio.

Nos métodos que trabalham no **espaço do objeto** as contribuições de cada célula são calculadas e combinadas para a produção da imagem final. Um exemplo é o método de *projeção de células*, no qual cada célula é projetada no plano da imagem, sendo essas projeções combinadas. Nos métodos que agem no **espaço da imagem**, por exemplo o *traçado de raios*, raios são lançados por cada pixel, sobre o volume gerando a imagem final. As contribuições das células, ao longo do raio, são calculadas e usadas para gerar a imagem. Nos **métodos de domínio** os dados da região são transformados em um domínio alternativo, como *compressão*, *wavelet*, ou *domínio de freqüência*, no qual uma projeção é diretamente gerada. Técnicas híbridas foram propostas no intuito de se benificiar das diferentes vantagens entre os diferentes métodos.

Nesse contexto, o presente trabalho propõe um método para visualização do volume usando a técnica de projeção de células.

1.3 Avanços Tecnológicos

Uma tendência atual na evolução dos equipamentos gráficos é possibilitar um acesso flexível a seus recursos de processamento. Em particular, placas gráficas modernas dispõem de uma unidade de processamento gráfico **GPU** (*Graphics Processing Unit*) programável pelo usuário.

GPUs são dispositivos de processamento vetoriais, ou seja, permitem que um mesmo trecho de código seja aplicado a diversos dados em paralelo. Com isso, computadores comuns podem se tornar máquinas vetoriais específicas para visualização volumétrica de alto desempenho.

Os avanços tecnológicos na computação gráfica não estão somente vinculados às placas gráficas modernas. Novos dispositivos de realidade virtual são desenvolvidos a cada ano, motivando o emprego da visualização volumétrica na inspeção e interação com volumes, por exemplo tomografias e ressonâncias.

A motivação deste trabalho se baseia nesses avanços, recursos e tecnologias que possibilitam uma forma eficiente de visualização. Foi utilizada uma técnica conhecida de projeção de células, implementada no contexto de programação em GPU, objetivando melhorar o desempenho da visualização e a qualidade das imagens geradas.

1.4 A Proposta do Trabalho

A proposta deste trabalho é criar um método interativo para visualização volumétrica, usando recursos gráficos recentes de programação em GPU. Dados volumétricos dispostos em células tetraedrais são visualizados utilizando o método de projeção de células.

O restante desta dissertação é organizada da seguinte forma. No Capítulo 2 são apresentadas revisões dos trabalhos de pesquisa relacionados com visualização volumétrica. É descrito detalhadamente a base do método utilizado no Capítulo 3.

Os detalhes da implementação proposta por esta dissertação são apresentados no Capítulo 4 e os seus resultados no Capítulo 5. Finalmente, no Capítulo 6 esta dissertação é concluída.

Capítulo 2

Revisão Bibliográfica

"If I have seen farther than others,
it is because I was standing on the
shoulder of giants."

– Isaac Newton

Como foi explicado no Capítulo 1, visualização volumétrica é: a partir de um volume (representado por dados volumétricos) visa-se obter uma imagem bidimensional. Essa imagem contém elementos de figura, chamados de *pixels (picture elements)* [3], enquanto que dados volumétricos contêm elementos de volume, chamados de *voxels (volume elements)* [6], como mostrado na Figura 2.1.

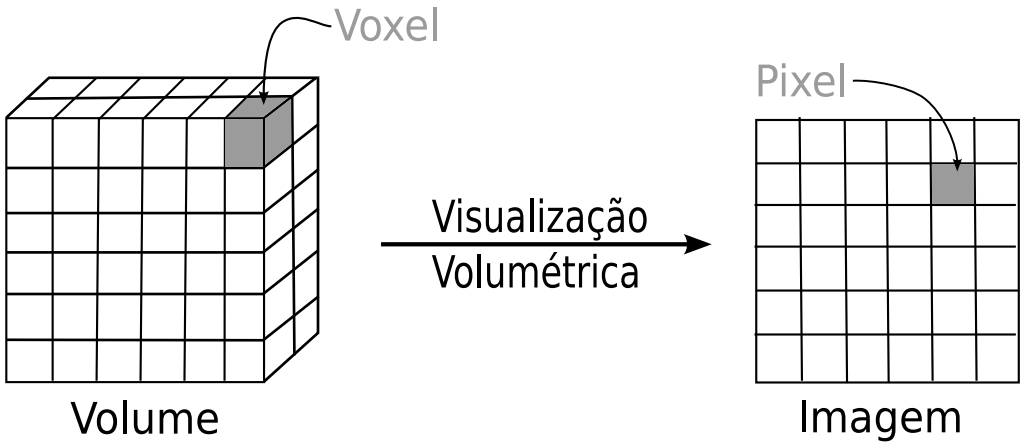

Figura 2.1: Visualização volumétrica é usada para gerar imagens a partir de informações do interior de volumes.

No decorrer deste capítulo são apresentados trabalhos de visualização volumétrica. Esses trabalhos estão divididos da seguinte forma:

- Técnicas de programação em placa gráfica (*algoritmos em GPU*);
- Métodos de visualização de superfícies (*iso-superfícies*) e de volume (*traçado de raios* e *projeção de células*);
- Trabalhos que objetivam calcular a interação física da luz com o volume (*integral de iluminação*);
- Técnicas aplicadas à visualização volumétrica em geral (*plano de varredura*).

2.1 Algoritmos em GPU

A partir de 2001, os produtores de placas gráficas disponibilizaram a funcionalidade da GPU programável. Com isso, programas restritos poderiam ser executados em GPU, no lugar da funcionalidade fixa da placa gráfica. As limitações inerentes a esses programas foram diminuindo com os avanços tecnológicos modernos. Atualmente, os programas executados em GPU, chamados de **shaders** [1], possuem algumas restrições e não alcançam ainda a versatilidade dos programas comuns em CPU.

Os shaders podem atuar em diferentes partes da linha de produção gráfica, conhecida como **pipeline** gráfico (veja a Figura 2.2). Inicialmente, os shaders eram aplicados somente na forma como a textura é lida (I), chamados de **shaders de textura** (*texture shaders*). Logo em seguida, os **shaders de pixel** (*pixel shaders*) generalizam os shaders de textura atuando na geração dos pixels (II). Os shaders de pixels, na verdade, geram os *pré-pixels*, ou *fragmentos*, pois ainda não passaram pelo processo de *composição* (b). Por este motivo, esses shaders são também chamados de **shaders de fragmento** (*fragment shaders*). Por último, o processo de computação da geometria dos vértices (III) passou a ser programável pelos **shaders de vértice** (*vertex shaders*).

[1] A palavra inglesa *shader* (usada nesta dissertação) significa *programa de computador* no contexto de programação em placa gráfica.

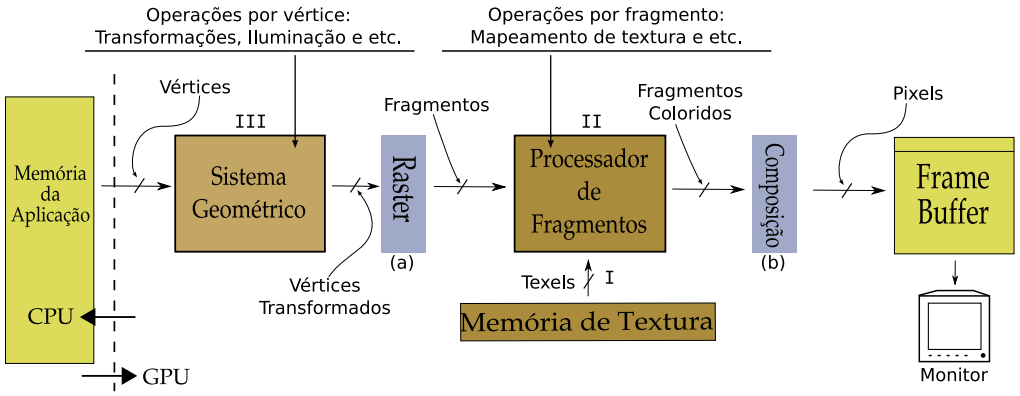

Figura 2.2: Pipeline gráfico resumido da GPU. Os processos que ocorrem em I, II e III podem ser personalizados.

A linguagem *OpenGL Shading Language* ou **GLSL** [7], desenvolvida pela *3Dlabs* [8] para a programação em GPU, é a linguagem escolhida para implementar o algoritmo proposto no presente trabalho.

O fluxo de processamento no pipeline gráfico pode ser analisado na Figura 2.2. Uma aplicação em CPU envia os vértices, de uma primitiva geométrica previamente definida (por exemplo triângulos), para o sistema geométrico (III). Em III várias operações por vértice são realizadas em paralelo. Essas operações podem ser da funcionalidade fixa (por exemplo transformações e iluminação) ou de um *shader de vértice*.

A saída de III passa pelo **raster** [2] (a), processo que preenche a primitiva pré-definida gerando fragmentos. Os elementos de textura, chamados de *texels* (*texture elements*) [3], são recuperados da memória da placa gráfica (I). Os texels são mapeados nos fragmentos pelo processo de *mapeamento de textura*, realizado pela funcionalidade fixa do processador de fragmentos (II). Em II as operações são realizadas em paralelo e podem ser substituídas por um *shader de fragmento*.

A saída de II passa pelo processo de *composição* (b) responsável por agregar os fragmentos em uma matriz de pixels, chamada de *frame buffer*. O conteúdo do frame buffer vai para o monitor, sendo mostrado na janela gráfica da aplicação.

[2] O verbo *rasterizar* (usado nesta dissertação) é um estrangeirismo derivado da palavra inglesa *rasterize* que significa *um padrão de linhas de pontos próximos que formam uma imagem.*

2.2 Iso-superfícies

Para reduzir a complexidade da tarefa de visualização volumétrica, algumas técnicas foram desenvolvidas para aproximar uma superfície contida no dado volumétrico [2]. Métodos de extração e visualização dessas superfícies, chamadas de **iso-superfícies** (*iso-surfaces*), melhoram o desempenho da visualização volumétrica, pois parte do volume é desconsiderado (veja a Figura 2.3). Apesar de ganhar desempenho na visualização, boa parte da informação contida no dado é perdida.

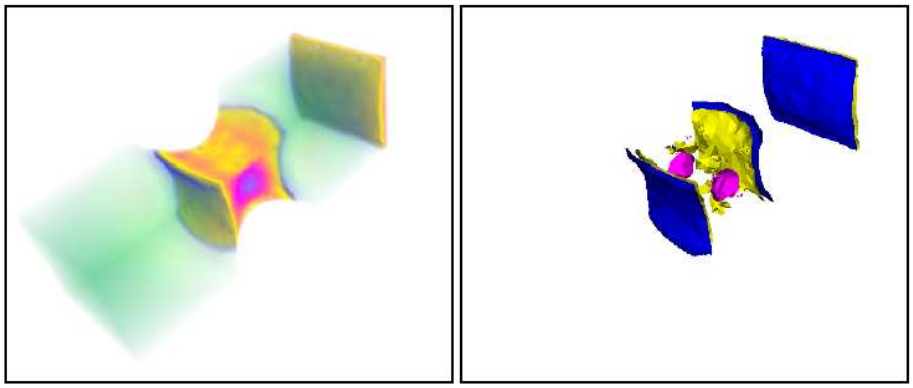

Figura 2.3: Diferença entre a visualização do volume (à esquerda) e visualização de iso-superfícies (à direita) [9].

A aproximação de iso-superfícies é feita usando primitivas geométricas, em geral triângulos, que podem ser *renderizados* [3] diretamente pela placa gráfica. Uma superfície pode ser definida por uma função de segmentação binária $f(s)$ aplicada ao dado volumétrico, onde $f(s)$ retorna 1 se o valor s é considerado parte do objeto e 0 se for parte do fundo (*background*). Existem dois tipos de regiões resultantes da aplicação de uma função $f(s)$:

- *Iso-superfície* (*iso-surface*);
- *Curvas de nível* (*iso-contours*).

Quando $f(s)$ é uma função degrau $f(s) = 1$, $\forall s \geq s_{iso}$, onde s_{iso} é chamado de **iso-valor** (*iso-value*), a região resultante é uma **iso-superfície**. Para o caso

[3] O verbo *renderizar* (usado nesta dissertação) é um estrangeirismo derivado da palavra inglesa *render* que significa desenhar.

de um intervalo $[s_1, s_2]$ em que $f(s) = 1$, $\forall s \in [s_1, s_2]$, onde $[s_1, s_2]$ é chamado de **iso-intervalo** (*iso-interval*) a região resultante é uma estrutura chamada de **curvas de nível** (*iso-contours*).

Lorensen e Cline [10] desenvolveram o algoritmo *Marching Cubes* para aproximar uma superfície de iso-valor à uma malha triangular. O algoritmo Marching Cubes trata células hexaedrais regulares, ou cubos, onde a função $f(s)$ é aplicada aos vértices da célula. Veja a Figura 2.4, dentro de cada célula, onde uma iso-superfície passa, os vértices de um triângulo da superfície são computados por interpolação nas arestas da célula.

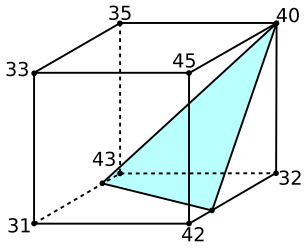

Figura 2.4: Exemplo de uma célula intersectada por uma iso-superfície (iso-valor = 40), com os valores dos voxels indicados. O algoritmo de Marching Cubes [10] interpola os vértices do triângulo nas arestas do cubo.

Max et al. [11] propõem combinar a idéia de visualização de superfícies com visualização de volume, visando melhorar a qualidade das imagens geradas. No trabalho de Max et al. é calculado apenas as curvas de nível. As curvas de nível são usadas por eles a fim de intensificar as áreas de transição dentro do volume.

O estudo e aplicação de iso-superfícies não residem no escopo do algoritmo proposto por esta dissertação. Entretanto, os trabalhos [12, 13, 14], relacionados à área de iso-superfícies, são analisados e comparados com o algoritmo proposto no Capítulo 5.

2.3 Integral de Iluminação

Em resposta ao problema de perda de informação pelo método de visualização de superfícies, técnicas mais diretas de visualização volumétrica foram desenvolvidas. Essas técnicas, chamadas de **visualização volumétrica direta** (*direct volume ren-*

dering) [15, 16] ou simplesmente **visualização volumétrica** (*volume rendering*), estudam a interação física da luz com a matéria [17, 18]. O volume é visualizado através do resultado dessa interação.

Calcular a interação física da luz exige a computação da **integral de iluminação** (*volume rendering integral*). A integral de iluminação é uma equação que computa a cor da luz que passa através do volume. Max [19] apresenta diferentes modelos para interação da luz com o volume. Neste trabalho de pesquisa é estudado o modelo de absorção mais emissão (*absortion plus emission*). Max mostra passo-a-passo a derivação da integral até chegar à:

$$I(D) = I_0 e^{-\int_0^D \tau(t)dt} + \int_0^D L(s)\tau(s)e^{-\int_s^D \tau(t)dt}ds \tag{2.1}$$

Essa é a equação da integral de iluminação. A Equação 2.1 calcula a mudança de intensidade no raio de luz I do final do volume $s = 0$ até ao observador $s = D$. Veja a Figura 2.5. O raio de luz atravessa uma distância D até o observador e o volume é visto lateralmente. O primeiro termo calcula a quantidade de luz de entrada, I_0, atenuada exponencialmente pela distância D. O segundo termo adiciona a quantidade de luz emitida por cada ponto ao longo do raio, levando em consideração a quantidade atenuada do ponto ao final do raio.

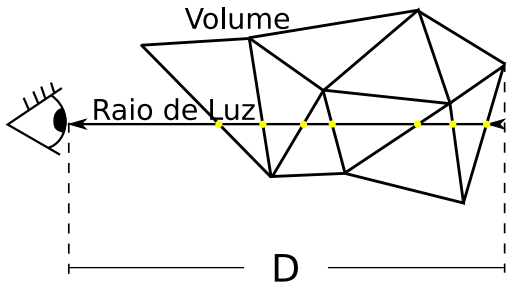

Figura 2.5: Modelo da integral de iluminação.

Computar a integral de iluminação completa quadro-a-quadro é um processo muito custoso e, portanto, evitado. O trabalho de Williams et al. [20] propõe soluções matemáticas exatas para a integral, porém com a desvantagem de baixo desempenho. Soluções exatas estão fora do escopo do presente trabalho, já que o objetivo é visualização volumétrica interativa.

Alguns trabalhos [21, 12, 22] melhoram o desempenho dos seus métodos de visualização simplificando essa integral. O modelo proposto por estes trabalhos difere da Equação 2.1, veja a Figura 2.6. Nesse modelo, o raio de visão (*viewing ray*) é considerado, ao invés do raio de luz, e a integral depende apenas da distância percorrida dentro da célula l (*length*), chamada de **espessura** (*thickness*) da célula, e os valores de entrada s_f (*scalar front*) e saída s_b (*scalar back*) do raio. O resultado da integral é então a parcela de contribuição da iluminação no pixel pela célula.

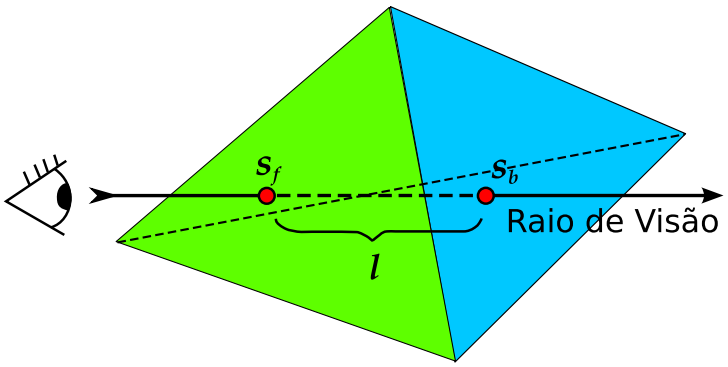

Figura 2.6: Modelo simplificado da integral de iluminação.

No modelo simplificado, a integral é reduzida a duas simples equações, como expressado por Shirley e Tuchman [21]:

$$C = \frac{C(s_f) + C(s_b)}{2} \tag{2.2}$$

$$\alpha = 1 - e^{-\frac{\tau(s_f)+\tau(s_b)}{2} l} \tag{2.3}$$

Na Equação 2.2 a cor C é computada pela média das cores de entrada $C(s_f)$ e saída $C(s_b)$. Na Equação 2.3 o cálculo da opacidade α é baseado no coeficiente de extinção (*extinction coefficient*) τ, que mede quanto de luz é absorvida pela célula. Da mesma forma que as cores, o coeficiente de extinção depende da entrada $\tau(s_f)$ e saída $\tau(s_b)$ da célula. As cores e os coeficientes de extinção são associados aos valores escalares de entrada s_f e saída s_b por uma função chamada: **função de transferência** (*transfer function*) [15].

A cor final do pixel é computada pela combinação das células. Para cada célula, além da primeira, a cor atualizada C_{i+1} é a combinação linear das cores anteriores C_i e C_{i-1}. Essa combinação é feita de acordo com a seguinte regra:

$$C_{i+1} = \alpha_i \, C_i + (1 - \alpha_i) \, C_{i-1} \tag{2.4}$$

$$\alpha_{i+1} = \alpha_i + \alpha_{i-1} \tag{2.5}$$

Note que, somente a opacidade da célula corrente α_i é usada na regra da Equação 2.4 (veja a Figura 2.7). A opacidade da célula já computada α_{i-1} é usada na Equação 2.5 para o cálculo da opacidade resultante α_{i+1}. No exemplo da Figura 2.7, o par (C_{i-1}, α_{i-1}) corresponde ao valor $RGBA$ do fragmento da célula $i-1$, que combinado ao fragmento da célula i gera a cor do pixel.

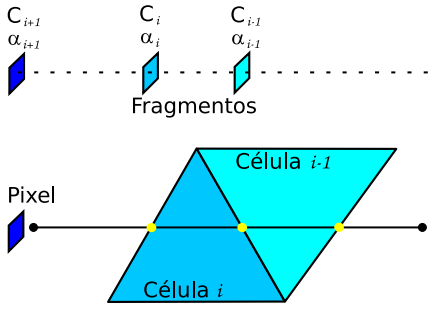

Figura 2.7: Combinação das células na computação da cor final do pixel. A ordem da combinação é determinada pelo sentido do raio, *de-frente-para-trás* (*front-to-back*) ou *de-trás-para-frente* (*back-to-front*).

Há uma outra maneira de computar a integral de iluminação, ao invés de usar o método da média dos escalares na obtenção da cor e opacidade, como visto nas Equações 2.2 e 2.3. Roettger et al. [12] empregam uma solução mais exata usando a equação:

$$s_l(x) = s_f + \frac{x}{l}(s_b - s_f) \tag{2.6}$$

Na Equação 2.6 é estimado o valor escalar de cada ponto dentro da célula. O valor escalar ao longo de l no ponto x é dado por $s_l(x)$, onde $x = 0$ é o ponto de entrada e $x = l$ o ponto de saída. Essa discretização do campo escalar, dentro da célula, permite o controle da exatidão desejada da integral. Define-se um número n de intervalos ao longo de l e, para cada célula, a média dos n escalares internos é

associada à contribuição de cor C e opacidade α da célula. Para o caso de $n = 2$, a Equação 2.6 calcula C e α como nas Equações 2.2 e 2.3.

Outra forma de contornar o cálculo quadro-a-quadro da integral de iluminação é computá-la previamente, armazenando os possíveis resultados discretizados em textura, como é exemplificado na Figura 2.8. Esta técnica, chamada de **pré-integração** (*pre-integration*), foi introduzida por Roettger et al. [12]. O trabalho de Engel et al. [13] foi pioneiro ao usá-la em GPU, usando shader de pixel.

Uma desvantagem da pré-integração é que a função de transferência deve se manter inalterada. Se a função de transferência mudar, a aplicação deve recalcular toda a tabela da integral de iluminação, e armazená-la em textura novamente. Esse procedimento é muito custoso, impedindo a alteração interativa da função de transferência. Roettger e Ertl [23] melhoram esse procedimento fazendo a recomputação dos valores pré-integrados em GPU, usando shader de pixel, reduzindo consideravelmente o tempo necessário.

Moreland e Angel [24] propõem uma solução diferente da pré-integração. Eles criaram o conceito de **pré-integração parcial**, onde apenas parte da integral de iluminação é computada. Moreland e Angel extraem matematicamente algumas funções da integral, tornando a pré-integração independente da função de transferência.

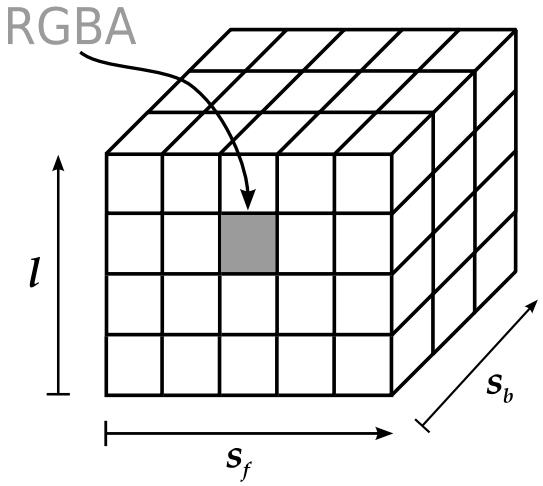

Figura 2.8: As cores (RGB) e opacidades (A) resultantes da pré-integração são armazenadas em textura, onde os parâmetros para consulta são (s_f, s_b, l).

Esta dissertação utiliza a tabela ψ [25] de pré-integração parcial criada por Moreland e Angel. Uma explicação mais detalhada da construção dessa tabela pode ser encontrada na tese de doutorado de Moreland [4]. Com a tabela ψ é possível obter qualidade na geração das imagens e alteração interativa da função de transferência.

2.4 Traçado de Raios

A técnica de **traçado de raios** (*ray tracing*) [4] é antiga em computação gráfica e seus conceitos foram inicialmente considerados em visualização volumétrica por Blinn [18]. Na concepção de Blinn, um raio de luz é lançado para cada pixel da tela, computando a absorção de luz por onde ele atravessa, veja a Figura 2.9.

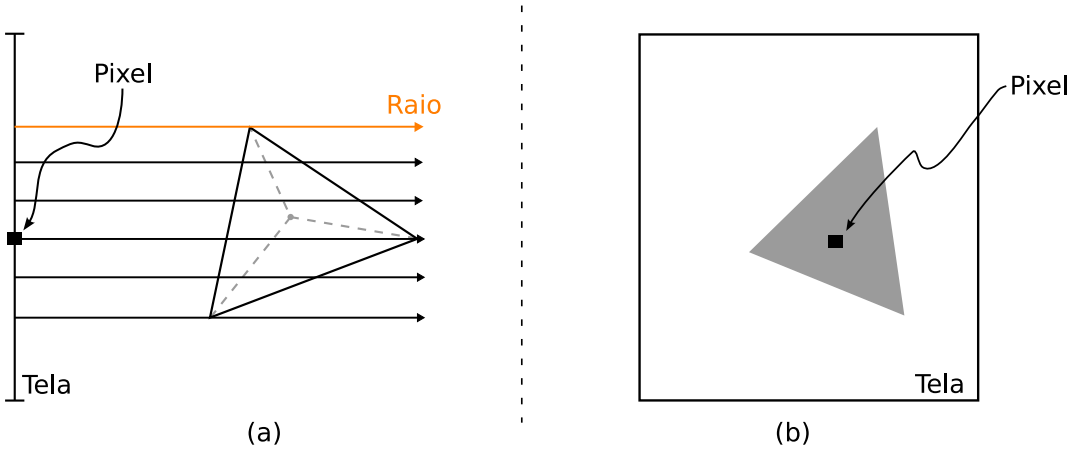

Figura 2.9: Exemplo da técnica de traçado de raios. A visão lateral dos raios atravessando uma célula do volume por pixel na tela é mostrada em (a), e o resultado em (b).

Posteriormente, o trabalho de Garrity [26] apresenta uma abordagem mais eficiente para o algoritmo de traçado de raios. Seu método traça os raios, em dados não-estruturados com transparência, considerando apenas a entrada do raio no volume por uma **face externa** (*external face*). As faces externas, também chamadas de *faces da borda* (*boundary faces*), são aquelas que pertencem à apenas uma célula. Em geral, o número de faces externas é bem menor que o total de faces. Com

[4] A técnica de traçado de raios (também chamada de *ray shooting*) é conhecida atualmente na área de visualização volumétrica como *ray casting*.

isso, Garrity testa apenas as faces externas reduzindo consideravelmente o número de testes de intersecção inicial (veja a Figura 2.10). Bunyk et al. [27] aceleram esse processo computando todas as intersecções dos raios com cada face externa da frente de uma única vez.

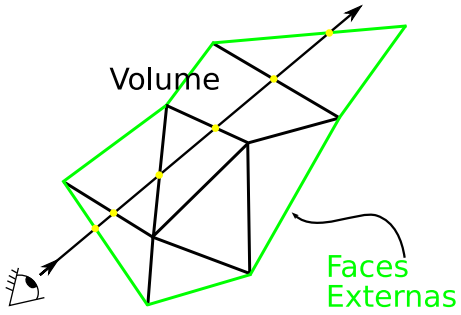

Figura 2.10: Uso das faces externas nos testes de intersecção inicial do raio.

Weiler et al. [28] desenvolveram uma forma de realizar traçado de raios em GPU, usando shader de fragmento. Weiler et al. nomearam o seu algoritmo de *Hardware-Based Ray Casting* ou **HARC**. O algoritmo HARC encontra a entrada inicial do raio renderizando as faces frontais, de forma similar a idéia de Garrity. O algoritmo atravessa o volume usando um shader de fragmento para guardar as computações das células em texturas. O algoritmo HARC é comparado ao algoritmo proposto neste trabalho no Capítulo 5.

Espinha e Celes [14] incrementam o algoritmo HARC escrevendo um novo algoritmo usando pré-integração parcial. Espinha e Celes empregam uma estrutura de dados mais eficiente que a implementação do HARC original. O algoritmo proposto por eles alcança alta qualidade e torna possível a alteração interativa da função de transferência. No Capítulo 5, os resultados do algoritmo de Espinha e Celes são analisados e comparados com o algoritmo proposto por esta dissertação.

Apesar do estudo de traçado de raios não residir no escopo deste trabalho, o conceito de unificação de algoritmos empregado por Espinha e Celes é utilizado. Neste trabalho de pesquisa a pré-integração parcial é usada com projeção de células, explicada mais detalhadamente na Seção 2.6.

2.5 Plano de varredura

Uma das limitações dos algoritmos de traçado de raios é não aproveitar a coerência entre os raios. Ou seja, para raios próximos, o método de traçado de raios recomputa todas as intersecções com as células, mesmo que duplicadas. Uma maneira de aproveitar a coerência entre os raios é com algoritmos de varredura (*sweep*). A idéia do algoritmo de varredura vem de geometria computacional, onde objetiva diminuir a dimensionalidade do problema [29].

Giertsen [30] introduz a idéia de varredura com traçado de raios em visualização volumétrica. Ele cria o **plano de varredura** (*sweep-plane*) que percorre o volume para a geração da imagem final, veja a Figura 2.11. Giertsen caminha com um plano de varredura, perpendicular à tela, em eventos específicos. Esses eventos são os vértices do volume onde a topografia dos polígonos muda e raios são lançados através do plano para determinar as cores dos pixels. Giertsen varre o modelo com um plano, mantendo um conjunto de polígonos formados pela intersecção do modelo com este plano. A vantagem do algoritmo de plano de varredura decorre da atualização incremental das intersecções, ao invés de arbitrária, aproveitando a coerência entre os raios.

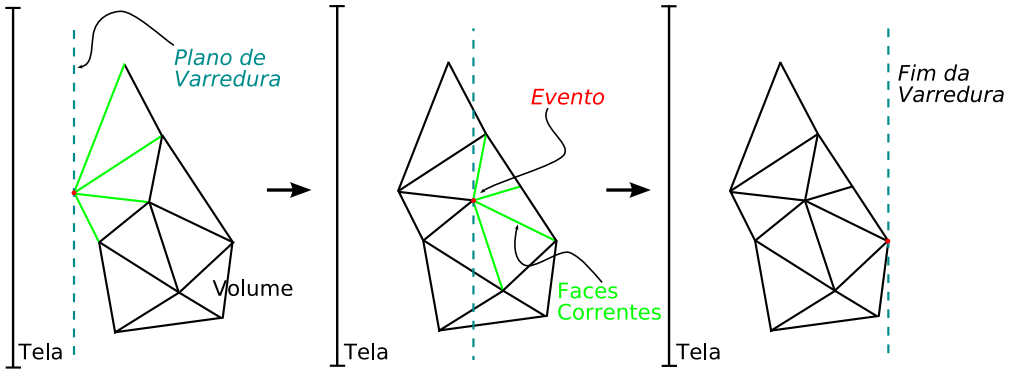

Figura 2.11: Exemplo de plano de varredura.

Posteriormente, Silva et al. [31, 32, 33] incrementam o desempenho da abordagem de Giertsen usando também uma **linha de varredura** (*sweep-line*) para determinar as intersecções dos polígonos no plano com os raios traçados.

Farias et al. [34] propõem uma nova abordagem de plano de varredura. O algoritmo **ZSWEEP**, proposto por eles, realiza a visualização do volume com o método

de projeção de células (explicada na Seção 2.6). Em cada evento do ZSWEEP, as faces ligadas ao vértice do evento são projetadas. Com as informações das projeções, o ZSWEEP extrai os parâmetros necessários para a integração do volume.

O estudo e aplicação de plano de varredura não fazem parte do algoritmo proposto por esta dissertação.

2.6 Projeção de Células

A técnica de **projeção de células** (*cell projection*), também chamada de *projeção direta* (*direct projection*), visa a geração de imagens do volume a partir da projeção de suas células na tela. A projeção é determinada transformando as células tridimensionais em primitivas geométricas bidimensionais no plano da imagem, ou **plano de visão** (*view plane*). Depois que as projeções das células são determinadas, veja a Figura 2.12, o processo de *rasterização* preenche as primitivas geométricas geradas com fragmentos, que são combinados na composição final do pixel. O algoritmo de projeção de células, por processar todas as células do volume, é considerado um método que executa no espaço do objeto.

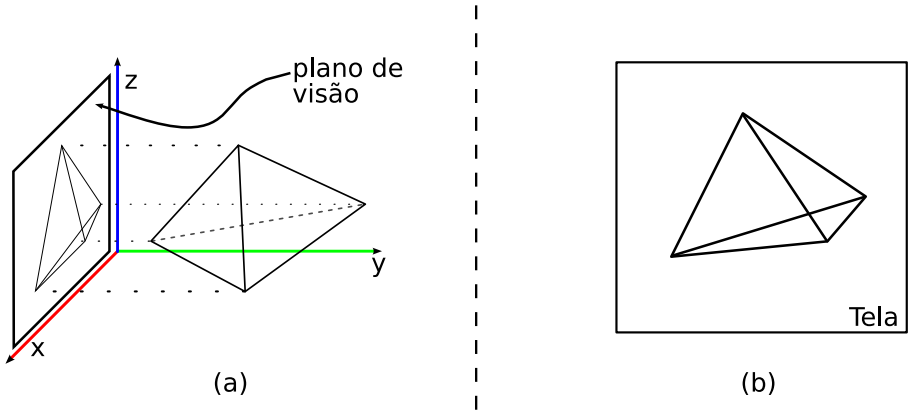

Figura 2.12: Exemplo da técnica de projeção de células. A projeção de uma célula do volume é mostrada em (a) e o resultado em (b).

A principal vantagem da projeção de células sobre o traçado de raios é que cada intersecção do raio com a célula é computada de uma única vez, quando a célula é projetada, tornando simples aproveitar a coerência entre os raios sem a necessidade de um algoritmo de varredura. O trabalho de Upson et al. [15] discute as vantagens

e desvantagens dos métodos de traçado de raios e projeção de células, antes do advento dos algoritmos em GPU.

Outra vantagem é que as placas gráficas modernas atuam também como um sistema renderizador no espaço do objeto, simplificando a implementação dos algoritmos de projeção de células em GPU e tornando-os mais eficazes que suas contrapartes em CPU. A principal desvantagem de projeção de células é a dependência de outro algoritmo para determinar a **ordem de visibilidade** (*visibility order*) das células.

O trabalho de Wittenbrink [35] visa melhorar o desempenho do algoritmo de projeção de células usando, entre outra técnicas, funções de otimização do OpenGL [5] [36]. A biblioteca OpenGL e suas funções de otimização são utilizadas na implementação proposta por esta dissertação, apresentada no Capítulo 4.

Shirley e Tuchman [21] propuseram o primeiro algoritmo de projeção de células em grades não-estruturadas. Em seu algoritmo, eles projetam exclusivamente um tipo de célula: o tetraedro. Por esse motivo, Shirley e Tuchman chamam o seu algoritmo de **projeção de tetraedros** ou **PT** (*projected tetrahedra*). O algoritmo PT forma a base do método proposto pelo presente trabalho e é detalhado no Capítulo 3.

A vantagem de escolher o tetraedro, dentre outros tipos de poliedros, é que ele é um *simplex* em três dimensões. Ou seja, o tetraedro é o poliedro mais simples possível, ele possui 4 vértices, 6 arestas e 4 faces, o mínimo necessário para a construção de um poliedro. Pelo tetraedro ser um simplex é possível decompôr qualquer poliedro em tetraedros. Desta forma, o algoritmo PT pode atuar em quaisquer tipos de grades não-estruturadas, pois suas células podem ser decompostas em tetraedros.

Um algoritmo similar ao PT foi proposto por Wilhelms e van Gelder [37]. Em seu algoritmo hexaedros são projetados ao invés de tetraedros. Apesar do hexaedro não ser um simplex, a sua forma é comumente derivada dos dados regulares. Nesses casos, cada hexaedro teria de ser dividido em cinco ou seis tetraedros para aplicação do algoritmo PT. Logo, projetar os hexaedros diretamente garante um ganho substancial de desempenho. Nesta dissertação, dados regulares advindos de amostras são tetraedrizados para serem visualizados.

[5] OpenGL é uma biblioteca de programação gráfica padrão multi-plataforma.

Kraus et al. [38] melhoram a qualidade das imagens do PT aplicando uma escala logarítmica para a tabela de pré-integração. Além disso, Kraus et al. apontam o uso de texturas com maior precisão (16 bits por componente de cor) como responsável por parte da melhora da qualidade. O algoritmo apresentado nesta dissertação usa precisão dupla (32 bits por componente de cor) no uso de texturas para melhor qualidade na geração de imagens.

Weiler et al. [39, 40] desenvolveram outro método de projeção de células implementado completamente em GPU, usando shader de vértice e fragmento. O algoritmo de Weiler et al., chamado *View Independent Cell Projection* ou **VICP**, aplica o mesmo shader de vértice e fragmento independente do **ponto de vista** (*view point*). Veja o exemplo na Figura 2.13. Os vértices v_1, v_2 e v_3, da face frontal f_0, são renderizados atribuindo valores escalares do volume às suas cores. A idéia de atribuir valores escalares às cores dos vértices é usada no algoritmo proposto por esta dissertação, como será visto no Capítulo 4.

A interpolação linear dessas cores, realizada pelo rasterizador da placa gráfica (como demostrado na Figura 2.2), fornece o escalar de entrada s_f. Weiler et al. assumem uma definição paramétrica para o raio de visão $r(t) = p + t\vec{d}$, onde $\vec{d}$ é a direção normalizada do raio r e p o ponto de entrada na face. Os parâmetros t_i do raio de visão são calculados pela intersecção do raio r com os planos correspondentes às faces f_i, com normal n_i, do tetraedro. O ponto de saída é determinado pelo menor parâmetro t_i positivo (t_2 no exemplo da Figura 2.13), que é também a espessura l da célula.

Finalmente, o algoritmo VICP de Weiler et al. computa o escalar de saída s_b usando o gradiente $\vec{g}$ do campo escalar dentro do volume da seguinte forma:

$$s_b = s_f + (\vec{g} \cdot \vec{d})l \tag{2.7}$$

Na Equação 2.7, a direção do raio de visão $\vec{d}$ é computada no shader de vértice pela diferença entre as posições do vértice e do ponto de vista. Esta direção é chamada de **vetor de visão** (*view vector*). A posição do ponto de vista é passada para o shader como **variável uniforme** (*uniform variable*), ou seja, o mesmo valor é lido por todos os vértices. O escalar de entrada s_f é computado pela interpolação dos

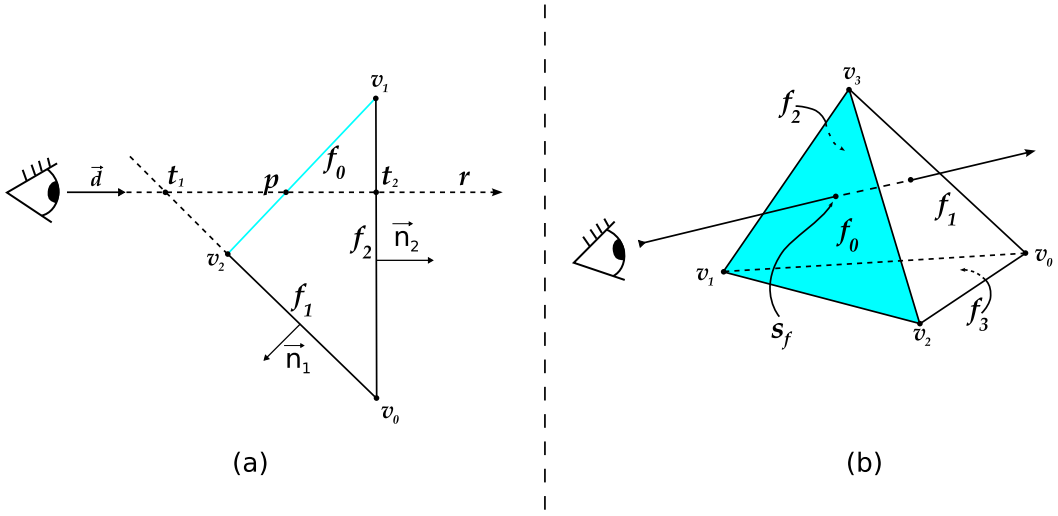

Figura 2.13: Exemplo do algoritmo VICP [40]. Em (a) uma visão lateral da intersecção tridimensional raio-célula mostrada em (b).

escalares originais, passados como **atributo do vértice** (*vertex attributes*). Essa interpolação é realizada pelo raster, entre o shader de vértice e fragmento, através das **variáveis variantes** (*varying variables*). A espessura l da célula é determinada no shader de fragmento do VICP, calculando a intersecção reta-plano. A Equação 2.7 é empregada em um dos últimos passos do shader de fragmento para computar o escalar de saída s_b.

De posse dos valores escalares de entrada s_f, de saída s_b e a espessura l da célula, o algoritmo VICP de Weiler et al. consulta a textura de pré-integração, sugerida por Roettger et al. [12], no último passo do seu shader de fragmento. A textura retorna a cor e opacidade do fragmento. A cor final do pixel é determinada pelo processo de composição dos fragmentos usando as Equações 2.4 e 2.5.

Wylie et al. [22] apresentam um novo método, chamado *GPU Accelerated Tetrahedra Renderer* ou **GATOR**, para realizar o algoritmo PT inteiramente em GPU, usando shader de vértice. O algoritmo proposto por esta dissertação usa também algumas idéias do GATOR, explicado com mais detalhes na Seção 3.1.

Capítulo 3

Algoritmo de Projeção de Tetraedros

"It is as interesting and as difficult
to say a thing well as to paint it."

– Vincent Van Gogh

O algoritmo de **projeção de tetraedros** ou **PT** de Shirley e Tuchman [21] foi descrito em 1990 e tornou-se a base de muitos trabalhos desde então. O algoritmo PT consiste em projetar tetraedros no plano da imagem e compô-los em ordem de visibilidade.

Se o volume estiver definido em uma grade retilínea ou curvilínea, então cada célula da grade deve ser particionada em tetraedros com um valor escalar por vértice da célula. Para o caso de grades não-estruturadas com células diferentes de tetraedros, cada célula deve ser particionada em tetraedros. Shirley e Tuchman [21] apresentam um método para *tetraedrizar* [1] todos os tipos de grades, porém este processo não faz parte do escopo desta dissertação.

A forma dos tetraedros projetados é classificada em uma de quatro classes, como mostrado na Figura 3.1. As classes representam as quatro possíveis silhuetas de projeção, dependendo do ponto de vista e da orientação do tetraedro. A classificação das células é realizada independente da rotação, translação ou escala da projeção do tetraedro no plano de visão. Note que as classes 3 e 4 são casos degenerados das

[1] O verbo *tetraedrizar* (usado nesta dissertação) significa transformar em tetraedros.

classes 1 e 2, pois um dos vértices é projetado sobre uma aresta (classe 3) ou sobre outro vértice (classe 4).

A classificação das células é feita baseada nas normais às faces do tetraedro original. Cada classe é distinguida examinando os vetores normais às faces e comparando-os com o ponto de vista. Essa comparação verifica apenas a direção e o sentido da normal em relação ao ponto de vista. As faces são marcadas com a notação mostrada na Figura 3.1, dependendo se:

- A normal aponta em direção ao ponto de vista: +;
- A normal aponta para o outro lado: −;
- Caso a normal seja perpendicultar ao vetor de visão: 0.

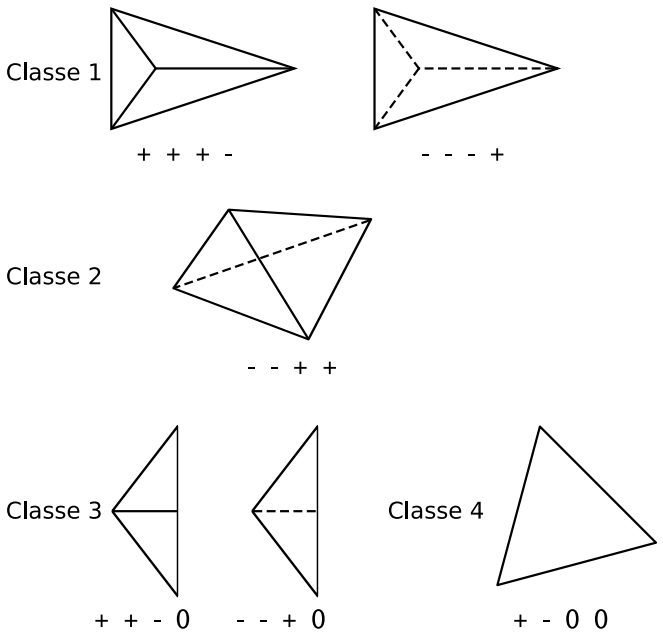

Figura 3.1: Diferentes classes do algoritmo de projeção de tetraedros e suas respectivas notações referentes as 4 faces do tetraedro [21].

Para cada tetraedro projetado, o **vértice espesso** (*thick vertex*) é definido como a projeção dos pontos de entrada e saída do segmento do raio que percorre a distância máxima dentro do tetraedro. O tamanho deste segmento é definido como **espessura da célula** (*thickness*). Todos os outros vértices projetados são chamados de **vértices finos** (*thin vertices*), pois o raio não percorre nenhuma distância.

Na Figura 3.2 é ilustrado um caso para cada classe de projeção. Os vértices v_i são as projeções dos vértices originais do tetraedro, onde s_{v_i} é o valor escalar do vértice. O vértice espesso é definido como v_t e seus atributos são: a espessura l da célula, os valores escalares de entrada s_f e saída s_b.

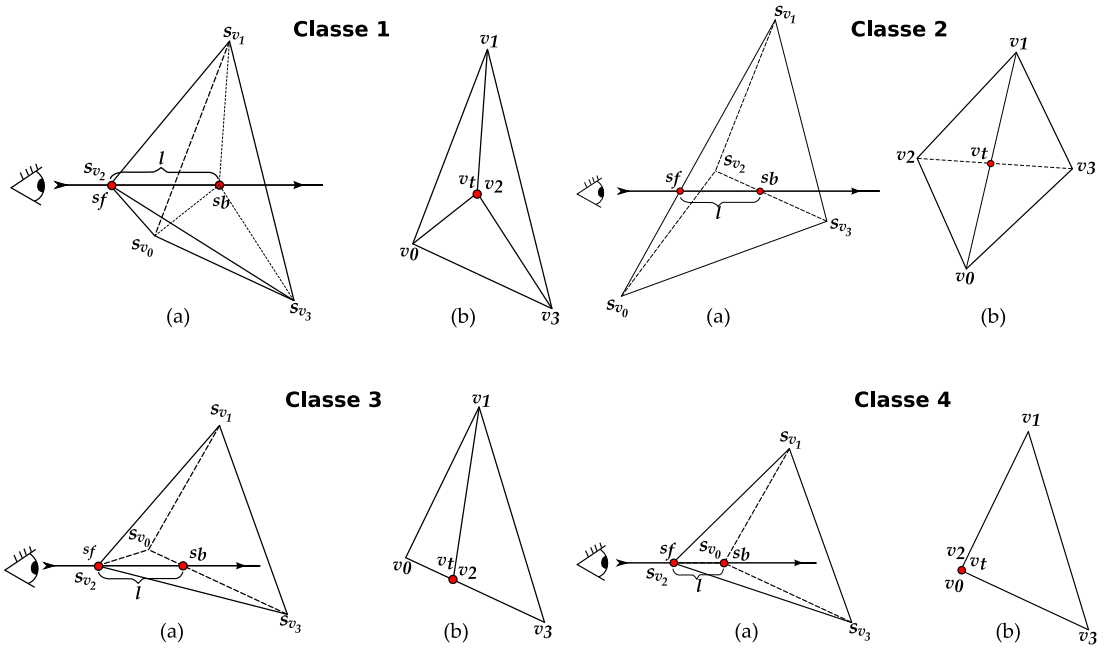

Figura 3.2: Um exemplo de cada classe de projeção. Os desenhos ilustram o tetraedro e o raio de visão (em a) e o tetraedro projetado (em b).

Analisando a Figura 3.2, para a projeção classe 4, o vértice espesso v_t é definido como v_2 ou v_0, pois esses dois vértices são colineares em relação ao vetor de visão. Em conseqüência, $s_f = s_{v_2}$, $s_b = s_{v_0}$ e l é igual ao tamanho da aresta dos vértices v_2 e v_0 no tetraedro original.

Para as outras classes é necessário realizar a intersecção das arestas dos triângulos projetados. A espessura l é computada realizando a projeção inversa de v_t, encontrando os pontos de entrada e saída, e computando a distância euclidiana desses pontos. No caso da classe 1, uma interpolação bilinear deve ser realizada:

$$v_t = v_0 + u(v_1 - v_0) + t(v_3 - v_0) \tag{3.1}$$

As coordenadas (x, y) de v_t são encontradas na projeção. Shirley e Tuchman usam a Equação 3.1 para computar a coordenada z de v_t. Finalmente, a espessura

l da projeção classe 1 é computada pela distância entre os vértices no tetraedro original (v_2 e v_t para o exemplo da Figura 3.2).

Cada tetraedro é decomposto em 1, 2, 3 ou 4 triângulos. O número de triângulos depende da classe de projeção, como mostrado na Figura 3.3.

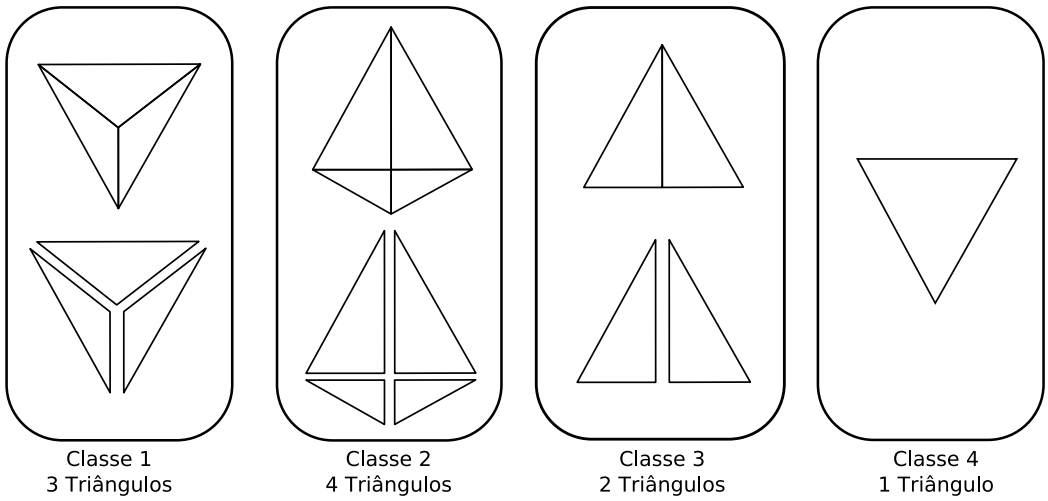

Figura 3.3: Decomposição das diferentes classes em triângulos.

Como discutido no Capítulo 2, a cor C e opacidade α dos fragmentos são interpoladas, depois de computadas a partir dos valores s_f, s_b e l dos vértices dos triângulos, de acordo com as Equações 2.2 e 2.3. A composição dos fragmentos dos triângulos renderizados é realizada seguindo a regra das Equações 2.4 e 2.5.

O algoritmo proposto nesta dissertação se baseia no algoritmo PT, porém adaptado para execução em GPU. Na próxima seção será apresentado a primeira adaptação do PT em GPU, feita por Wylie et al. [22]. Algumas de suas idéias são usadas nesta dissertação.

3.1 Algoritmo PT em GPU

O trabalho de Wylie et al. [22] propõe implementar o algoritmo PT na placa gráfica, usando shader de vértice. Para tal, o algoritmo **GATOR** (que significa *GPU Accelerated Tetrahedra Renderer*) desenvolvido por Wylie et al., classifica as projeções dos tetraedros de forma diferente do algoritmo PT de Shirley e Tuchman [21].

Os shaders de vértice não suportam a criação e nem remoção dinâmica de vértices. A topologia e o número de vértices são fixos e estritamente determinados

pela aplicação em CPU. Por este motivo, os diferentes números de triângulos por classe de projeção (mostrado na Figura 3.3) não permitem implementar o PT original usando shader de vértice. Para contornar este problema, Wylie et al. criaram uma topologia fixa, que pode ser adaptada através da manipulação da posição dos vértices. Essa topologia fixa, chamada de **grafo base** (*basis graph*), é isomorfa à projeção bidimensional do tetraedro no plano da imagem.

Observe o exemplo da Figura 3.4, as projeções classe 1 e 2 são mapeadas no grafo base. Na classe 2, o vértice resultante da intersecção (o vértice espesso) das arestas v_I (como explicado na seção anterior) é mapeado em v'_4 no grafo base. Na classe 1, o vértice v_0 é mapeado tanto em v'_3 como em v'_0.

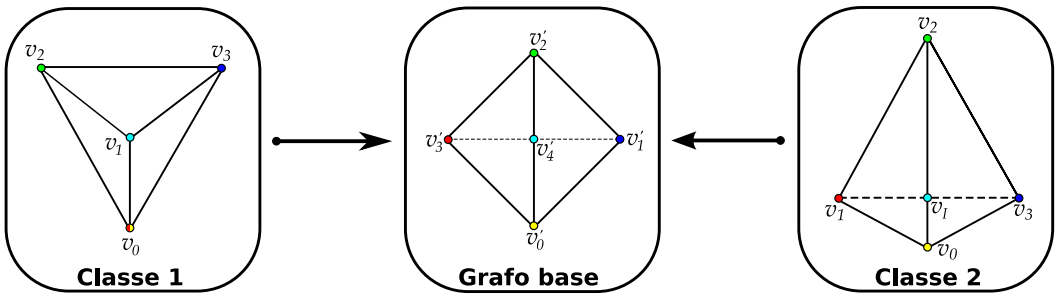

Figura 3.4: Grafo base usado no algoritmo do GATOR.

Os vértices v'_i do grafo base são alterados, no shader de vértice, para a posição do vértice correspondente da projeção do tetraedro. Os triângulos do grafo base são renderizados como um **leque de triângulos** (*triangle fan*), ou seja, seis vértices entram no pipeline gráfico na seguinte ordem: v'_4, v'_0, v'_1, v'_2, v'_3 e v'_0. Os três primeiros vértices formam um triângulo e cada vértice após o terceiro forma um novo triângulo.

No caso das classes de projeção 4, 3 e 1, onde a decomposição resulta em 1, 2 e 3 triângulos respectivamente, os triângulos degenerados são renderizados sem influenciar na imagem final. No exemplo classe 1 da Figura 3.4, o triângulo $v'_3v'_0v'_4$ tem área nula, ou seja, é degenerado, pois os vértices v'_3 e v'_0 são mapeados para o mesmo lugar.

A classificação das células feita pelo algoritmo GATOR é diferente da classificação do algoritmo original PT. Wylie et al. propõem testes de classificação independente das normais às faces e do vetor de visão. O algoritmo GATOR define 3 vetores e realiza 4 testes binários com esses vetores para classificar cada célula. Observando

a Figura 3.5, para realizar os testes no shader de vértice, o algoritmo GATOR precisa de todos todos os vértices do tetraedro como atributo. Desta forma, $vec1$, $vec2$ e $vec3$ podem ser computados e os 4 testes realizados dentro do shader para cada vértice. Note a redundância de dados imposta pela limitação do shader de vértice, para cada um dos 6 vértices do leque (grafo base), o algoritmo GATOR precisa dos 4 vértices do tetraedro original. Nesta dissertação o shader de fragmento é utilizado evitando esta redundância.

$$\begin{aligned} vec1 &= v_1 - v_0 \\ vec2 &= v_2 - v_0 \\ vec3 &= v_3 - v_0 \end{aligned}$$

$$\begin{aligned} teste1 &= (vec1 \times vec2).z > 0 \\ teste2 &= (vec1 \times vec3).z < 0 \\ teste3 &= (\text{distância de } v_0 \text{ para } v_M - \text{ distância de } v_0 \text{ para } v_I) > 0 \\ teste4 &= vec1.z > 0 \end{aligned}$$

Figura 3.5: Definições e testes usados pelo shader de vértice do GATOR.

Ao completar os testes, o algoritmo GATOR consulta uma tabela verdade para determinar o caso de projeção (veja a Tabela 3.1) em um dos últimos passos do shader de vértice. Wylie et al. mostraram 14 casos para representar as diferentes permutações das classes 1 e 2 do algoritmo PT. A partir destes casos, cada vértice é mapeado no grafo base e sua posição atualizada. A nova posição do vértice é uma das saídas do shader do GATOR e os triângulos resultantes expressam a projeção do tetraedro correspondente no plano da imagem.

O $teste3$ identifica se a projeção é classe 1 ou 2. O *vértice do meio* v_M será mapeado em v'_4 (no grafo base) se a projeção não for classe 2. Somente nos casos da classe 2, o *vértice de intersecção* v_I é mapeado em v'_4. Observe a Figura 3.6, os produtos vetoriais determinam o posicionamento relativo dos vértices projetados. O vértice do meio é definido dependendo deste posicionamento, no exemplo: $v_M = v_1$ para o caso 4; e $v_M = v_2$ para o caso 12. Nos casos da classe 2, o vértice do meio é definido desconsiderando o vértice de intersecção v_I. Desta forma, a distância entre

Caso	*teste*1	*teste*2	*teste*3	*teste*4
1	1	1	X	1
2	1	1	X	0
3	1	0	0	0
4	1	0	0	1
5	0	1	0	1
6	0	1	0	0
7	0	0	0	1
8	0	0	0	0
9	1	0	1	0
10	1	0	1	1
11	0	1	1	1
12	0	1	1	0
13	0	0	1	1
14	0	0	1	0

Tabela 3.1: Tabela verdade do algoritmo GATOR [22]. O valor 0 representa $falso$ e 1 representa *verdade*. O *teste*3 não se aplica nas entradas X da tabela.

v_0 e v_M é menor que a distância entre v_0 e v_I para os casos da classe 1 e maior para os da classe 2.

O algoritmo GATOR tem a desvantagem de desconsiderar os casos degenerados, ou seja, casos correspondentes às classes 3 e 4. O algoritmo proposto por esta dissertação trata esses casos degenerados, no intuito de gerar melhores imagens que o GATOR.

Os atributos do vértice espesso (s_f, s_b e l) são computados de forma similar ao algoritmo PT. A computação de intersecção das arestas é realizada no shader de

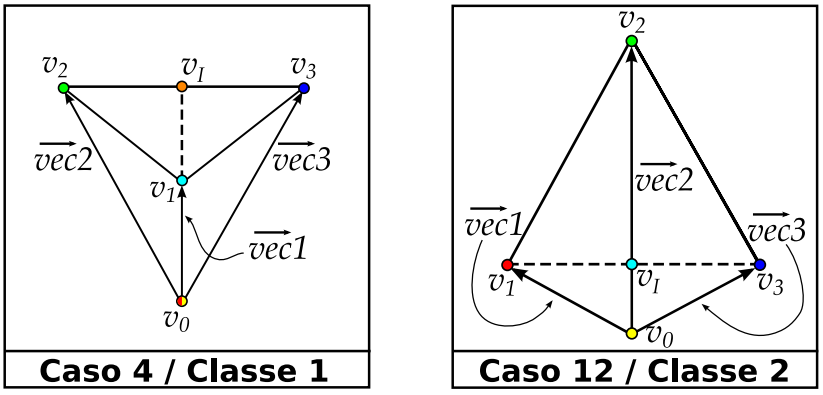

Figura 3.6: Exemplo dos casos 4 e 12 da classificação do algoritmo GATOR. O caso 4 corresponde à classe 1 e o caso 12 à classe 2.

vértice. O resultado do shader, implementado por Wylie et al., consiste em:

- Posições atualizadas dos vértices;
- Cor e opacidade $RGBA$ nos vértices.

Finalmente, o resultado do shader é rasterizado e fragmentos do leque de triângulos são gerados. Note que nenhum fragmento é gerado para triângulos degenerados. Esses fragmentos são compostos em pixels usando a mesma regra que o PT.

Esta dissertação propõe um algoritmo baseado no PT de Shirley e Tuchman [21], usando a idéia de classificação do GATOR de Wylie et al. [22]. Porém o algoritmo proposto é implementado em 2 passos no intuito de aproveitar as vantagens do shader de fragmento, ao invés do shader de vértice, como é mostrado no próximo capítulo. Além disso, o algoritmo proposto classifica também os casos degenerados desconsiderados no GATOR, produzindo imagens exatas e com qualidade superior.

Capítulo 4

Algoritmo proposto

"There is a single light of science,
and to brighten it anywhere is
to brighten it everywhere."

– Isaac Asimov

O algoritmo proposto por esta dissertação é dividido em 2 passos principais em GPU. Ambos os passos são implementados com a biblioteca OpenGL 2.0 [36], na linguagem GLSL [7] e usando shaders de vértice e fragmento. Entretanto, a maior parte da computação é realizada pelos shaders de fragmentos.

No **primeiro passo**, todos os dados necessários para implementação do algoritmo PT [21] são computados e processados por tetraedro. No **segundo passo**, esses dados são interpolados e usados na consulta à tabela ψ [25] de pré-integração parcial de Moreland e Angel [24]. Com esta tabela, a cor e opacidade $RGBA$ são determinadas para cada fragmento. A composição dos fragmentos em pixels é feita de forma similar ao PT de Shirley e Tuchman [21].

Cada quadro (*frame*), finalizado pelo algoritmo proposto, necessita de uma seqüência de processos descritos na Figura 4.1. A seguir, cada processo é descrito sucintamente para nas próximas seções serem mais detalhados.

Dados do volume que se deseja visualizar são armazenados em memória de textura. Dados dos tetraedros do volume são mapeados em um quadrado, que é renderizado, ou seja, enviado para o processador de fragmentos (I). O shader de vértice

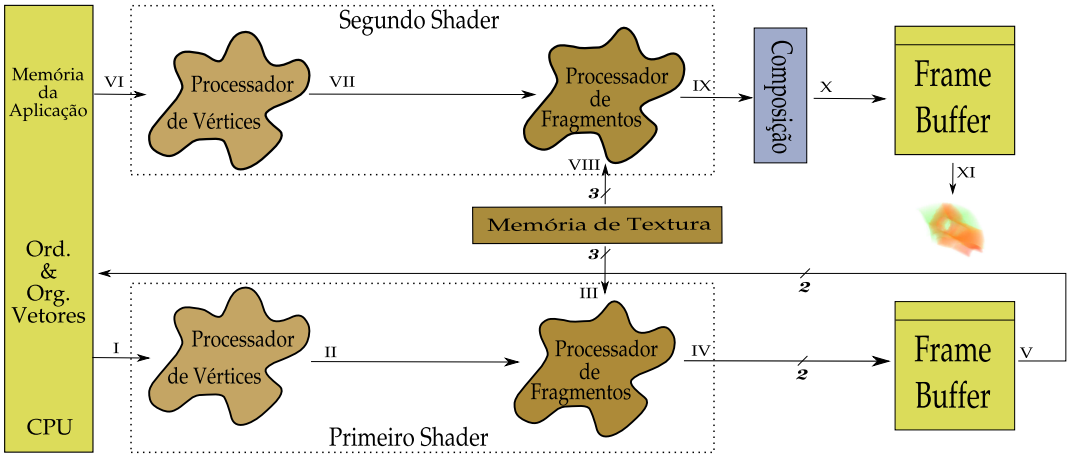

Figura 4.1: Descrição detalhada do pipeline do algoritmo proposto.

do primeiro passo simplesmente projeta o quadrado renderizado, de forma que os fragmentos gerados (II) correspondam aos texels da textura de tetraedros. Posteriormente, o shader de fragmento do primeiro passo realiza as seguintes etapas:

1- Lê os dados volumétricos consultando uma textura de tetraedros e outra de vértices (III);

2- Computa a projeção do tetraedro;

3- Classifica a projeção consultando uma terceira textura com a tabela de classificação (III);

4- Calcula a espessura da célula l, os escalares de entrada s_f e saída s_b;

5- Escreve em dois diferentes *frame buffers* (IV).

Cada fragmento de saída do shader corresponde à exatamente um pixel. Os pixels, de dois frame buffers diferentes, são lidos de volta para a memória da aplicação (V). Os dados retornados são ordenados e organizados em dois vetores: de vértices e de cores. Ambos os vetores contém dados referentes à cada tetraedro. As cores e os vértices armazenados nestes vetores são renderizados (VI) como **leques de triângulos** (*triangle fan*).

O shader de vértice do segundo passo realiza as projeções restantes, pois somente a projeção do vértice espesso é retornada pelo primeiro passo. A espessura da célula,

os escalares de entrada e saída são interpolados pela rasterização (VII) e entram no processador de fragmentos. O shader de fragmento do segundo passo realiza as seguintes etapas:

1- Determina as cores de entrada e saída consultando uma textura com a função de transferência (VIII);

2- Calcula a opacidade α consultando uma segunda textura com resultados pré-computados da função exponencial, de acordo com a Equação 2.3 (VIII);

3- Colore o fragmento consultando a tabela ψ armazenada em uma terceira textura (VIII).

Com os valores rasterizados e as texturas, a cor e opacidade $RGBA$ do fragmento são computadas (IX) e compostas (X) na imagem final da tela (XI).

Para acelerar o processo de renderização, são usados os recursos de **vetor de vértices e de cores** (*vertex and color array*) do OpenGL [36] no desenho das primitivas. Diferente do GATOR [22], que usa o grafo base para cobrir todos os casos, o algoritmo proposto não renderiza triângulos degenerados desnecessariamente. Depois de determinada a classificação no primeiro passo, 1, 2, 3 ou 4 triângulos serão enviados para o segundo passo por tetraedro, dependendo se a projeção for classe 4, 3, 1 ou 2 respectivamente.

4.1 Primeiro passo

O primeiro passo é implementado com shaders de vértice e fragmento em GPU e é responsável pela projeção e classificação dos tetraedros. Como a classe de projeção depende do ponto de vista, este primeiro passo só é executado se o volume for manipulado. Há duas manipulações permitidas pela implementação do algoritmo proposto: *rotação* e *zoom*. Quando nenhuma das duas manipulações está sendo feita, o pipeline começa do segundo passo (processo VI em diante na Figura 4.1). Por este motivo, o primeiro passo pode ser referido como *passo de atualização*.

Os dados volumétricos a serem visualizados são lidos de um arquivo e armazenados em memória da placa gráfica (memória de textura). Duas texturas são criadas

para armazenar os dados volumétricos: **Textura de Vértices** e **Textura de Tetraedros**. Na Textura de Vértices são armazenadas as coordenadas do vértice e o seu valor escalar (x, y, z, s) em seus texels $RGBA$. Na Textura de Tetraedros são armazenados os índices com a conectividade dos vértices do volume. Para cada texel $RGBA$ da Textura de Tetraedros, quatro índices apontam para os 4 vértices do tetraedro correspondente àquele texel.

Na Figura 4.2 é apresentado o esquema de conectividade do volume. Este esquema de descrição reduzido do volume permite a alocação de dados volumétricos com uma grande quantidade de células (milhões de tetraedros em uma placa gráfica com 256 MB). Cada textura tem 32 bits por componente de cor, consumindo 16 Bytes por tetraedro na Textura de Tetraedros, e 16 Bytes por vértice na Textura de Vértices. As texturas são passadas para os shaders como *variáveis uniformes*.

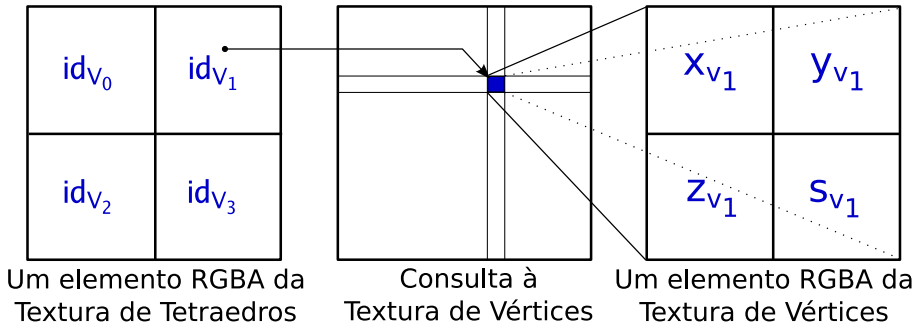

Figura 4.2: Cada texel da Textura de Tetraedros contém quatro índices para a Textura de Vértices.

Para executar o shader de fragmento uma vez por tetraedro, a Textura de Tetraedros é mapeada à um quadrado com o tamanho da tela. Assim o número de texels será igual ao número de pixels (aproximadamente o número de tetraedros). Este método é comumente usado nos algoritmos de *GPU para Propósito Geral* (*General Purpose GPU – GPGPU*) [41], onde o objetivo é empregar a GPU para computação genérica ao invés de processamento gráfico. Ou seja, a imagem resultante dos shaders é lida como resultado de computação ao invés de imagem à ser visualizada.

Uma terceira textura é carregada na GPU para descobrir a classificação da projeção. Essa textura é chamada de **Textura de Classificação** e contém uma *tabela verdade ternária* (*Ternary Truth Table – TTT*), descrita detalhadamente a seguir

(veja a Tabela 4.1). Esta tabela compreende as diferentes permutações das classes de projeção, chamadas de **casos**. Em cima dos 14 casos do Wylie et. al. [22], foram adicionados 24 casos da classe três e 12 casos da classe quatro. No total, 50 casos descrevem todas as permutações das 4 classes de projeção (veja alguns exemplos na Figura 4.4). Cada texel $RGBA$ representa um dos casos e contém a ordem correta para computar o vértice de intersecção.

Com o uso dessas 3 texturas não há necessidade de passar atributos de vértices, reduzindo assim a transferência de dados pelo barramento entre CPU e GPU. A sobrecarga neste barramento (*bus transfer overhead*) impossibilita a visualização interativa.

No **primeiro passo**, a Textura de Tetraedros é mapeada à um quadrado que é enviado à GPU. O shader de vértice substitui a funcionalidade fixa no intuito de garantir que o quadrado seja renderizado no tamanho da tela. Para isso, apenas a matriz de projeção (*projection matrix*) é aplicada aos quatro vértices do quadrado. Enquanto que, a matriz de transformações (*modelview matrix*) não é usada. Desta forma, o algoritmo garante que cada fragmento corresponderá à um tetraedro, e o shader de fragmento será executado por tetraedro.

O shader de fragmento do primeiro passo do algoritmo proposto é responsável por realizar as seguintes etapas:

1- Ler os 4 vértices da Textura de Vértices (5 acessos);

2- Calcular o baricentro do tetraedro;

3- Projetar os vértices do tetraedro;

4- Classificar a projeção dos vértices (1 acesso);

5- Computar o vértice espesso v_t;

6- Computar a espessura da célula l, os escalares de entrada s_f e saída s_b;

7- Escrever os resultados nos 2 frame buffers de saída.

Na primeira etapa, 1 acesso à memória de textura é realizado consultando a Textura de Tetraedros para retornar quatro índices dos vértices do tetraedro. Em seguida, mais 4 acessos são realizados para ler os 4 vértices da Textura de Vértices.

Na segunda etapa, o baricentro do tetraedro é calculado pela média das coordenadas dos 4 vértices. A coordenada z do baricentro é uma das saídas deste shader e é usado pelo algoritmo de ordenação de células, antes do segundo passo ser executado, como descrito na próxima seção. Entretanto, a ordenação por baricentro é aproximada e é utilizada apenas por questão de eficiência.

Na terceira etapa, os 4 vértices são transformados e projetados no plano da imagem usando *variáveis uniformes embutidas* (*built-in uniform variables*). A classificação da projeção é determinada na quarta etapa computando 4 diferentes testes. Este processo de classificação é similar ao algoritmo GATOR do Wylie [22], exceto que também são tratados os casos degenerados. Além disso, o algoritmo proposto evita redundância computacional realizando os testes uma única vez por tetraedro ao invés de uma vez por vértice como feito pelo GATOR.

$$
\begin{aligned}
vec_1 &= v_1 - v_0 \\
vec_2 &= v_2 - v_0 \\
vec_3 &= v_3 - v_0 \\
vec_4 &= v_1 - v_2 \\
vec_5 &= v_1 - v_3
\end{aligned}
$$

$$
\begin{aligned}
teste_1 &= sign((vec_1 \times vec_2).z) + 1 \\
teste_2 &= sign((vec_1 \times vec_3).z) + 1 \\
teste_3 &= sign((vec_2 \times vec_3).z) + 1 \\
teste_4 &= sign((vec_4 \times vec_5).z) + 1
\end{aligned}
$$

Figura 4.3: Testes realizados no shader de fragmento para classificação da projeção. A função *sign* do GLSL [7] retorna -1, 0 ou 1 dependendo se o argumento é menor que, igual a ou maior que zero, respectivamente.

A classificação realizada pela quinta etapa respeita o modelo da Figura 4.3. Cada $teste_i$ representa a posição de um vetor em relação ao outro e pode ter 3 possíveis resultados: 0, 1 ou 2. Por este motivo, a tabela de classificação usada é dita ternária. Por exemplo, o $teste_3$ terá valor 0 se o vetor vec_2 estiver à esquerda de vec_3; ou valor 1 caso os vetores sejam paralelos (classe 3 ou 4); ou valor 2 se vec_2 estiver à direita de

vec_3. Os vértices v_i são os vértices do tetraedro original projetados. Os 4 testes são usados para determinar em que caso das classes de projeção o tetraedro se encontra.

Os resultados dos testes são usados em uma operação de consulta à Textura de Classificação. Essa textura armazena uma tabela verdade ternária, que é indexada por todos os possíveis resultados dos 4 testes. Observe a Tabela 4.1, cada resultado $teste_{1234}$ corresponde à um caso de classificação da projeção. O texel $RGBA$ contém a ordem correta em que os vértices projetados devem estar para que a intersecção seja realizada.

id_{ttt}	$teste_{1234}$	$Caso$	$RGBA$
0	0 0 0 0	12	0-3-2-1
1	0 0 0 1	18	0-3-2-1
2	0 0 0 2	6	0-3-2-1
3	0 0 1 0	25	1-0-3-2
4	0 0 1 1	40	2-3-1-0
.	.	.	.
.	.	.	.
.	.	.	.
80	2 2 2 2	11	0-1-2-3

Tabela 4.1: Tabela verdade ternária do algoritmo proposto.

Note que a Tabela 4.1 possui entradas sem caso e ordem de vértices. Isto ocorre porquê há combinações de testes que não resultam em nenhum caso. Por exemplo, o $teste_{1234} = 0111$ ($id_{ttt} = 13$) representa um tetraedro em que os 4 vértices são colineares, ou seja, nenhum triângulo é gerado. No total de 81 entradas da tabela verdade ternária, apenas 50 são casos válidos.

Na Figura 4.4 são apresentados 4 exemplos de casos, um para cada classe de projeção. O caso 6 corresponde ao resultado $teste_{1234} = 0002$, ou seja, vec_1 está à esquerda de vec_2 e vec_3 ($teste_1 = 0$ e $teste_2 = 0$), vec_2 está à esquerda de vec_3 ($teste_3 = 0$) e vec_4 está à direita de vec_5 ($teste_4 = 2$). De forma equivalente, o caso 12 tem como resultado $teste_{1234} = 0000$, o caso 25 corresponde ao $teste_{1234} = 0010$ e o caso 40 ao $teste_{1234} = 0011$.

O identificador da tabela verdade ternária id_{TTT} é computado, na quarta etapa do shader de fragmento, com a seguinte regra:

$$id_{TTT} = teste_1 * 3^3 + teste_2 * 3^2 + teste_3 * 3^1 + teste_4 * 3^0 \qquad (4.1)$$

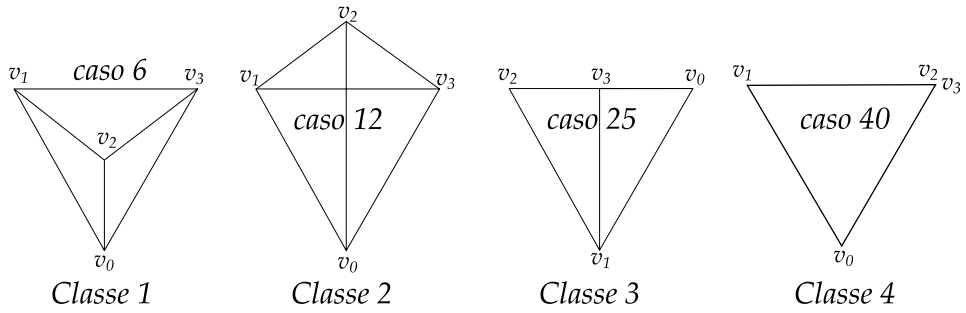

Figura 4.4: Exemplos de quatro casos, um para cada classe do algoritmo PT.

Os casos degenerados são descobertos quando um ou mais testes resultam em 1. Se um teste retorna 1, então é um caso da classe 3, se dois testes retornarem 1, então é um caso da classe 4. Para esses casos a maior parte das computações nas próximas etapas podem ser evitadas ou trocadas por operações mais simples. Observe a Figura 4.4, o caso 40 (da classe 4) possui os vetores vec_2 e vec_3 paralelos e os vetores vec_4 e vec_5 também paralelos.

Se mais de dois testes retornarem 1 todos os pontos foram projetados em uma linha, logo o modelo original contém um tetraedro com volume nulo e essa projeção é descartada. Identificar os casos degenerados é importante para que artefatos não sejam gerados na imagem final.

Na quinta etapa, o vértice espesso v_t é encontrado realizando a intersecção entre $v_0 - v_2$ e $v_1 - v_3$. Note que, os vértices v_i são os vértices do tetraedro original projetados, porém em uma ordem determinada pelo *RGBA* da Textura de Classificação (Tabela 4.1). Com a ordem dos vértices definida, as propriedades do vértice espesso (s_f, s_b, l) são computadas na sexta etapa.

Na última etapa, os dados do fragmento são retornados usando *renderização em múltiplos alvos* (*multiple render targets – MRT*) com dois *objetos de frame buffers* (*frame buffer objects – FBO*) anexados à texturas (*color attachments*). Cada um anexado à uma textura *RGBA* com 32 bits por componente. A primeira textura de saída contém as coordenadas do vértice de intersecção v_{I_x} e v_{I_y} (usado somente para as projeções classe 2), a coordenada z do baricentro do tetraedro z_c (*centroid* z) e o índice para a tabela verdade ternária id_{TTT} (*ternary truth table identifier*). A segunda textura de saída contém os escalares de entrada s_f e saída s_b, a espessura l e o número de vértices do leque de triângulos *cnt* (*triangle fan count*). Esse esquema

é descrito na Figura 4.5.

É importante notar que toda a computação realizada em GPU é processada em paralelo. No shader de fragmento do primeiro passo, a atualização de todos os tetraedros do volume é realizada em paralelo, pois cada tetraedro corresponde a um fragmento. A quantidade de tetraedros processados em paralelo depende do número de processadores de fragmentos da placa gráfica.

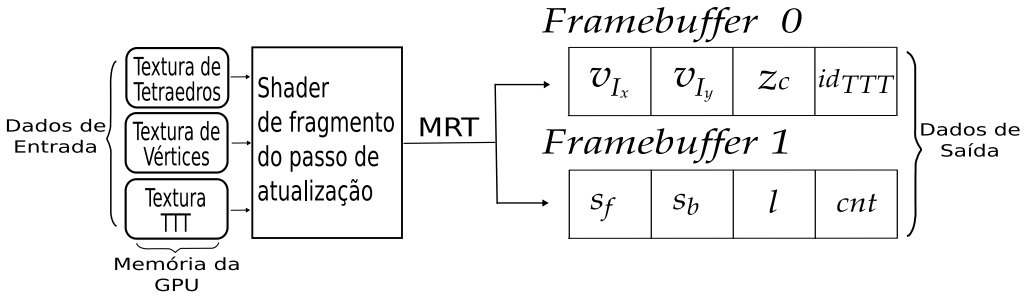

Figura 4.5: Esquema de entrada/saída do shader de fragmento do primeiro passo.

4.2 Ordenação das células e organização dos vetores

Um dos problemas dos algoritmos de projeção de células é a necessidade de **ordenar as células** em ordem de visibilidade antes de renderizar. Como mencionado na Seção 4.1, o primeiro shader de fragmento retorna a coordenada z dos baricentros dos tetraedros para que um algoritmo em CPU utilize-os no intuito de ordenar os tetraedros em ordem de profundidade.

Dois algoritmos de ordenação são usados na implementação proposta: uma rápida mas imprecisa ordenação (*bucket sort*) é usada sempre que o ponto de vista está mudando, enquanto que um algoritmo padrão de ordenação (*merge sort*) é empregado para quadros parados.

O bucket sort é baseado em fatias perpendiculares ao vetor de visão, isto é, os tetraedros são divididos em k grupos (20 deles) de forma que todos os tetraedros em um dado grupo têm aproximadamente a mesma profundidade em relação ao observador. Isto significa que todos os tetraedros dentro de um grupo são renderizados em quaisquer ordem com um pequeno impacto na exatidão da imagem final. As fatias são visitadas de-trás-para-frente (*back-to-front*).

Por outro lado, quando o usuário pára de manipular o modelo, uma ordenação merge sort padrão é usada para obter uma ordem de profundidade mais apurada dos baricentros. Para um volume com n tetraedros a complexidade do merge sort é $O(n\log n)$. Desta forma a imagem final, para quadros parados, possui uma exatidão maior que imagens produzidas durante a manipulação do volume.

Deve ser mencionado, entretanto, que a ordenação por baricentros não garante resultados 100% corretos em todos os casos. Se este nível de precisão é requerido, uma abordagem mais sofisticada deve ser utilizada para ordenar os tetraedros, como o MPVO de Williams [42] ou algoritmos baseado no trabalho de Williams [43, 44] ou ainda o k-buffer de Callahan et al. [45].

Para o segundo passo do algoritmo ser executado é necessário **organizar os vetores**: de vértices e de cores. Tanto a ordenação das células como a organização dos vetores são realizadas em CPU e preparam a GPU para executar o segundo passo do algoritmo.

O algoritmo proposto renderiza os leques de triângulos com a função otimizada do OpenGL [36]: *glMultiDrawElements*. Essa função necessita de um vetor de índices e um de contadores como argumento e referencia os vetores de vértice e de cores (*vertex and color arrays*) que guardam as coordenadas dos vértices e informações para o cálculo de suas cores (veja a Figura 4.6).

Os vetores de vértices e cores são agrupados em cinco elementos por tetraedro: o vértice espesso mais os quatro vértices originais. Esses dois arrays são quase constantes, pois para cada mudança na direção de visão somente a posição e a cor do vértice espesso são atualizadas. A atualização dos quatro vértices originais não é retornada pelo primeiro passo e, logo, não é atualizada no processo de organização de vetores.

Cada elemento do vetor de vértices contém as coordenadas (x, y, z) do vértice. O vetor de cores contém valores (s_f, s_b, l) para cada vértice, ao invés das cores normais RGB, que serão computadas no final do segundo passo do algoritmo. Note que, para os vértices *finos*, $s_f = s_b$ e $l = 0$ (como mostrado na Figura 4.7).

Para renderizar os leques de triângulos, dois vetores adicionais são necessários. O vetor de índices ($indice_i$) é dividido em grupos, cada um com seis elementos que

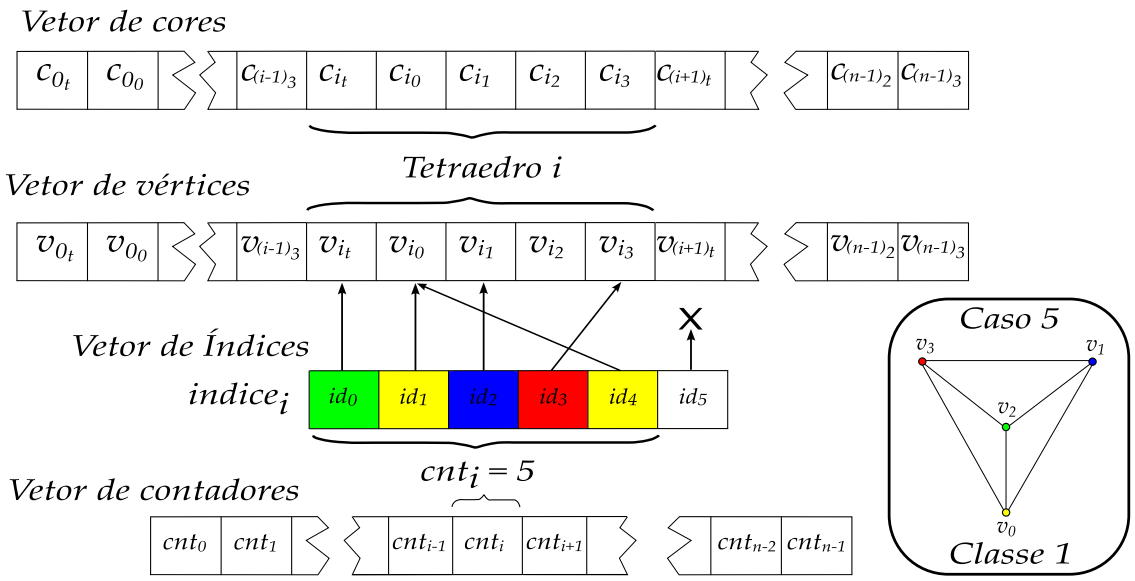

Figura 4.6: Estrutura de dados para a função *glMultiDrawElements*. Os índices ilustram o caso 5 (da classe 1), onde a ordem correta para renderizar o tetraedro i é $v_{i_t} - v_{i_0} - v_{i_1} - v_{i_3} - v_{i_0}$. Note que v_{i_2} é o vértice espesso e suas coordenadas são copiadas para v_{i_t}.

guardam a ordem correta dos vértices no leque usado para renderizar o tetraedro. O vetor de contadores (cnt_i) contém o número de vértices em cada leque. Este valor é retornado pelo primeiro passo. Note que o número máximo de vértices no leque é seis (casos de classe 2). Todos os vetores usados são atualizados em CPU usando os dados retornados pelo primeiro shader de fragmento. O uso do grupo $indices_i$ é limitado pelo número cnt_i, ou seja, o tetraedro i terá $cnt_i - 2$ triângulos renderizados correspondentes à sua classe de projeção. Veja a Figura 4.6 para mais detalhes.

É importante notar que a ordenação das células e organização dos vetores só é realizada caso o modelo tiver sido manipulado, ou seja, o primeiro passo foi executado. Caso contrário, apenas o segundo passo é executado por quadro.

4.3 Segundo passo

O segundo passo é implementado em GPU, como no primeiro passo, e é responsável pela visualização volumétrica propriamente dita. Por este motivo, o segundo passo pode ser referido como *passo de renderização*.

No primeiro passo, somente as propriedades do vértice espesso v_t são retornadas do shader de fragmento, deixando os quatro vértices originais do tetraedro sem alteração. O shader de vértice do segundo passo computa as atualizações dos vértices restantes. Ele substitui a funcionalidade fixa para aplicar as matrizes de projeção e transformações (*Modelview–Projection Matrix*) a todos os vértices do tetraedro exceto o vértice espesso.

A saída do shader de vértice é rasterizada e fragmentos são gerado pela interpolação das cores dos vértices. Note que, as cores RGB dos vértices correspondem aos valores (s_f, s_b, l). Por este motivo, a espessura da célula l, os escalares de entrada s_f e saída s_b são interpolados. Na Figura 4.7, um exemplo da interpolação é mostrada com os valores do fragmento interpolado (s'_f, s'_b, l').

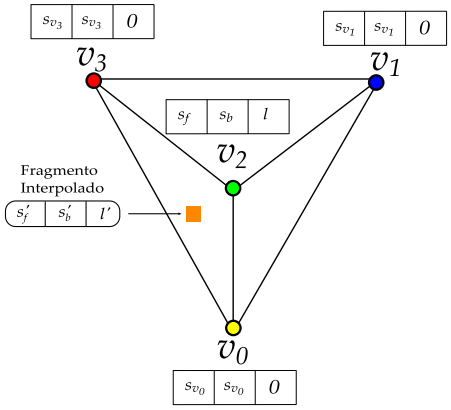

Figura 4.7: Valores interpolados para o exemplo classe 1 mostrado na Figura 4.6. Note que, exceto pelo vértice espesso, todos os outros são renderizados com os valores originais do volume.

O shader de fragmento do segundo passo do algoritmo proposto é responsável por realizar as seguintes etapas:

1- Determinar cores correspondentes aos escalares de entrada e saída (2 acessos);

2- Computar parâmetros da tabela ψ;

3- Calcular opacidade α (1 acesso);

4- Consultar a tabela ψ (1 acesso);

5- Colorir o fragmento.

Na primeira etapa, as cores RGB e o coeficiente de extinção τ do escalar de entrada e saída são determinados. Para isso são necessários 2 acessos, um para cada escalar, à uma textura 1D que armazena a função de transferência corrente usada para o volume.

Na segunda etapa, parâmetros de consulta à tabela de pré-integração parcial são computados. Estes parâmetros dependem apenas do coeficiente de extinção τ e da espessura l da célula, como descrito no trabalho de Moreland e Angel [24].

Na terceira etapa, a opacidade α do fragmento é calculada. Para isso é necessário 1 acesso à uma segunda textura que armazena valores pré-computados de opacidade exponencial (Equação 2.3). Testes experimentais indicam que o uso dessa textura 1D é ligeiramente mais rápido que computar a função exponencial no shader de fragmento. A precisão utilizada foi de 512 valores de e^{-u}, com u variando no intervalo de $[0, 1]$.

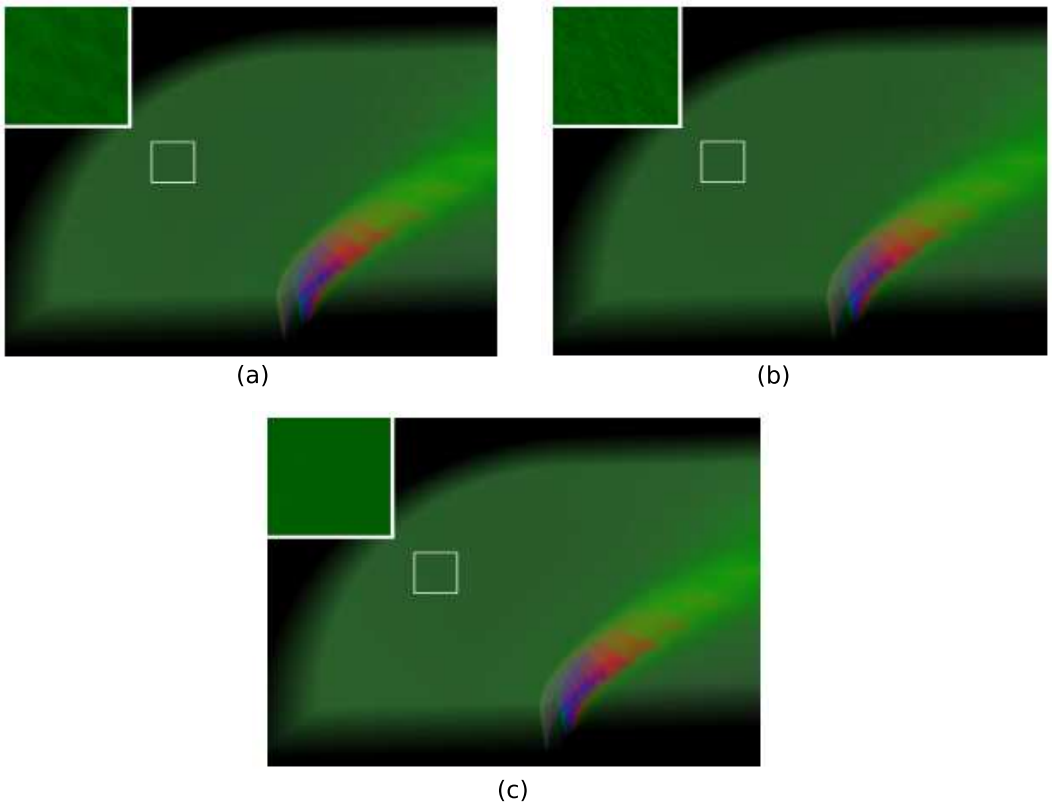

Figura 4.8: Exemplos de renderização de artefatos [38]: (a) sem consulta HILO à textura; (b) frame buffer com 8 bits por componente de cor; (c) usando consulta HILO e frame buffer com precisão ponto-flutuante.

Na quarta etapa, uma terceira textura 2D com a tabela ψ é utilizada. O valor retornado pela textura é empregado para encontrar uma cor intermediária às cores de entrada e saída, determinadas na primeira etapa. Esta cor intermediária corresponde a cor final do fragmento. Na Figura 4.8 são analisados diferentes fragmentos dependendo da estratégia de renderização.

Os fragmentos retornados pelo segundo shader são compostos e enviados para o frame buffer de saída que resultará na imagem final. O passo de renderização é obrigatório para todos os quadros. É neste passo que o volume é visualizado e a qualidade das imagens de cada quadro pode ser medida.

A resolução em bits da textura de pré-integração e do frame buffer de saída influenciam na qualidade da imagem final, como estudado por Kraus et al. [38] (veja a Figura 4.8). Consulta a texturas com 16 bits por componente de cor (texturas do tipo HILO [46]) ou mais produzem melhores resultados, assim como o uso de frame buffer com precisão ponto-flutuante.

Na implementação proposta por este trabalho de pesquisa, os artefatos (erros nas imagens geradas) foram evitados utilizando precisão dupla na consulta à textura e no frame buffer de saída.

Capítulo 5

Resultados

"History will be kind to me
for I intend to write it."

– Winston Churchill

No capítulo anterior foi descrito uma implementação do algoritmo PT [21] em GPU com pré-integração parcial [24]. A implementação foi escrita em C++ usando OpenGL 2.0 [36, 47, 48] com GLSL [7] em Linux. Medidas de desempenho foram realizadas em um Intel Pentium IV 3.6 GHz com 2 GB RAM. A placa gráfica utilizada foi a nVidia GeForce 6800 com 256 MB e barramento PCI Express 16x.

Os dados volumétricos utilizados nos testes do algoritmo proposto foram: *Blunt Fin* e *Liquid Oxygen Post* do site NAS da NASA [49], ambos dispostos em grades curvilíneas e convertidos em tetraedros; *SPX* [50] e *Combustion Chamber* do laboratório LLNL [51], o SPX subdividido em um número maior de tetraedros; *Fuel Injection* e *Brain Tomography* do site Volvis [52], dispostos em grades retilíneas convertidas em tetraedros.

Na Tabela 5.1 são apresentadas informações e resultados obtidos sobre esses dados volumétricos. O tamanho de cada volume, o número de vértices e tetraedros são listados. O desempenho do algoritmo proposto é descrito em quadros por segundo (*frames per second – fps*) e em tetraedros por segundo (*tetrahedra per second – tets/s*), tendo sido executado com bucket sort. Valores indicados com K representam

milhares e com M milhões. O plano de visão (*viewport*) utilizado para visualização dos volumes é de 512x512 pixels e os tempos são dados considerando que o volume está constantemente rotacionando.

Dado	Vértices	Tetraedros	fps	M tets/s
Blunt Fin	40 K	187 K	11,30	2,1197
Oxygen Post	110 K	513 K	4,49	2,3844
SPX	150 K	828 K	3,04	2,5269
Combustion Chamber	47 K	215 K	9,32	2,0054
Fuel Injection	262 K	1,5 M	1,49	2,2460
Brain Tomography	950 K	5,5 M	0,46	2,5608

Tabela 5.1: Tamanho e tempo médio dos diferentes volumes.

Na Figura 5.1 é apresentada a interface da implementação do algoritmo proposto. O volume visualizado foi o SPX e nas laterais são apresentadas informações sobre o volume e sua visualização. Note que os tempos estão inferiores dos apresentados na Tabela 5.1. Isto se deve ao fato da ordenação merge sort estar ativada para todos os quadros e não somente para os parados.

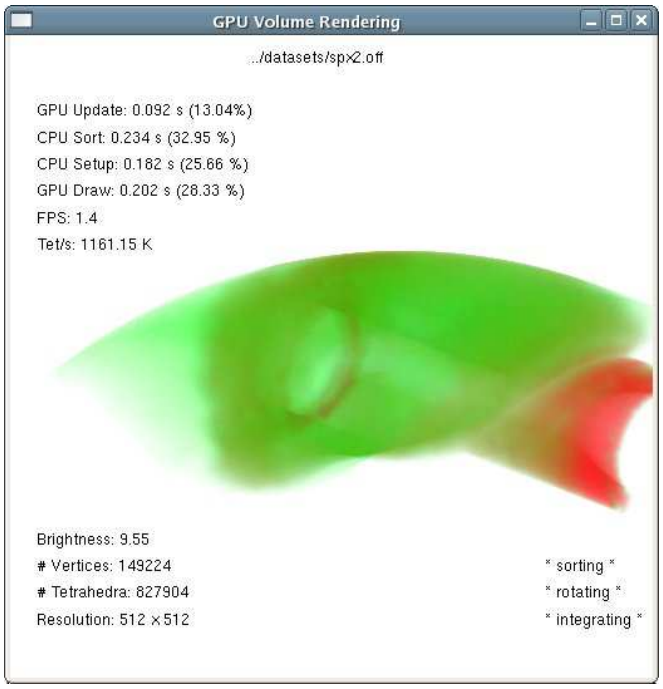

Figura 5.1: Exemplo da janela de interface do algoritmo proposto.

Os tempos e porcentagens das seguintes partes do algoritmo são mostrados no canto superior esquerdo:

- GPU Update – passo de atualização em GPU;
- CPU Sort – ordenação dos vetores em CPU;
- CPU Setup – organização dos vetores em CPU;
- GPU Draw – passo de renderização em GPU.

Informações sobre o plano de visão e o volume visualizado são mostrados no canto inferior esquerdo:

- Brightness – brilho da imagem gerada;
- # Vertices – número de vértices do volume;
- # Tetrahedra – número de tetraedros do volume;
- Resolution – resolução da imagem gerada.

No canto inferior direito, da Figura 5.1, aparecem as características da visualização corrente:

- Sorting – a ordenação merge sort está ativada para todos os quadros e não só para os quadros parados;
- Rotating – o volume está constante rotacionando independente de interação;
- Integrating – a pré-integração parcial está ativada.

Na Figura 5.2 uma outra janela disponibiliza uma interface para alteração interativa da função de transferência. Ao passo que o gráfico é manipulado, a janela com o volume reflete a alteração imediatamente.

Na Tabela 5.2, o algoritmo proposto é comparado com outros algoritmos de visualização volumétrica. A tabela apresenta o desempenho médio em 128 segundos de animação (volume em constante rotação).

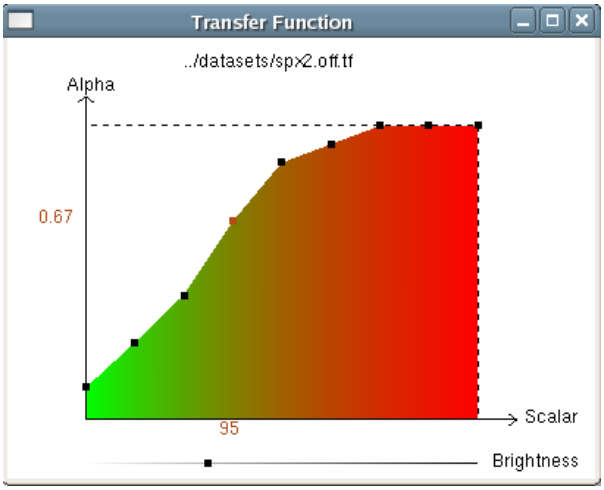

Figura 5.2: Exemplo de outra janela para edição da função de transferência.

Os seguintes algoritmos de projeção de células e de traçado de raios foram testados:

- PTINT – Algoritmo proposto por esta dissertação (PT com pré-integração parcial);
- GATOR – GPU Accelerated Tetrahedra Renderer [22];
- VICP – View-Independent Cell Projection (implementado em GPU e CPU) [40];
- VICP (Balanced) – VICP balanceado [24];
- HARC – Hardware-Based Ray Casting [28];
- HARC (INT) – HARC com pré-integração parcial [14];
- HAVIS – Hardware-Accelerated Volume and Isosurface Rendering Based on Cell-Projection [12].

É importante ressaltar que apesar do algoritmo proposto (PTINT) ganhar em desempenho no *Blunt Fin* (11,30 fps), ele perde para os algoritmos de traçado de raios (HARC e HARC(INT)) no *Oxygen Post* (4,49 fps). Isto ocorre devido às diferentes características dos volumes, ou seja, o *Oxygen Post* (veja a Figura 5.4 (b)), em determinados pontos de vista, possui uma quantidade pequena de pixels para a computação do algoritmo de traçado de raios. Enquanto que o *Blunt Fin* (veja

Algoritmo / Dado	Blunt Fin	Oxygen Post
PTINT	11,30 fps	4,49 fps
GATOR	4,07 fps	1,51 fps
VICP (GPU)	5,20 fps	1,93 fps
VICP (CPU)	1,82 fps	0,57 fps
VICP (Balanced)	4,10 fps	0,11 fps
HARC	4,47 fps	8,63 fps
HARC (INT)	4,94 fps	5,93 fps
HAVIS	2,36 fps	0,79 fps

Tabela 5.2: Comparação de tempo entre os diferentes algoritmos de visualização volumétrica.

a Figura 5.4 (a)) não tem uma grande variação na quantidade de pixels na imagem final. Para os algoritmos de projeção de células, como o PTINT, esta variação é irrelevante.

Uma outra análise dessa diferença de desempenho reside no espaço de abordagem de cada método. Os algoritmos de traçado de raios trabalham no espaço da imagem e, portanto, seu desempenho é diretamente proporcional ao número de pixels na imagem final. Enquanto que para os algoritmos de projeção de células, que trabalham no espaço do objeto, o desempenho depende da complexidade do modelo. Desta forma, quanto maior a imagem melhor o desempenho dos algoritmos de projeção de células sobre os de traçado de raios. Por outro lado, quanto maior o modelo melhor o desempenho dos algoritmos de traçado de raios comparado aos de projeção de células.

Observe a Figura 5.3, o desempenho do algoritmo proposto é mostrado para diferentes resoluções. Note uma característica importante dos algoritmos de projeção de células realizado em grandes resoluções. O aumento da imagem não prejudica o desempenho do algoritmo proposto, pois este só depende do número de células do modelo.

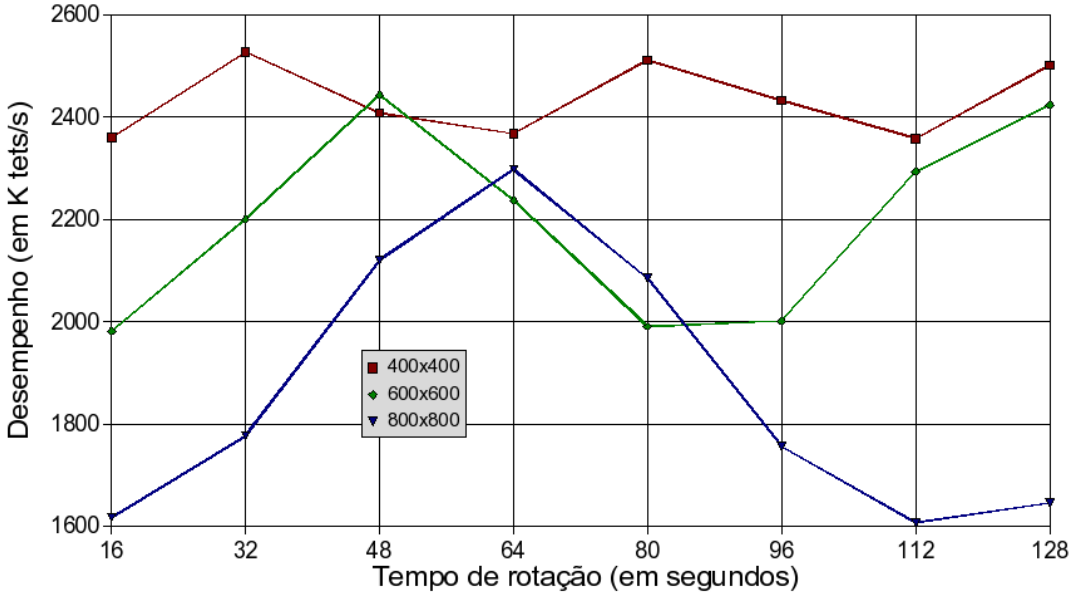

Figura 5.3: Desempenho na renderização (em K tets/s) por tempo de rotação (em segundos) do *Oxygen Post* para diferentes resoluções.

Além disso, a implementação do algoritmo proposto requer uma quantidade menor de Bytes por tetraedro (*Bytes/tet*) que os algoritmos de traçado de raios comparados, que requerem guardar estruturas de dados complexas em memória de GPU.

A abordagem proposta por esta dissertação requer 16 Bytes por elemento de textura dos tetraedros e vértices. Considerando uma média de 4 tetraedros por vértice (por análise dos volumes testados), a implementação necessita de apenas 20 Bytes/tet. Isso significa que um milhão de células ocupa por volta de 20 MB de memória de GPU. Em contraste, HARC (INT) [14] requer 96 Bytes/tet, enquanto a implementação original do HARC precisa de 144 Bytes/tet.

Um exemplo da visualização de cada um dos seis volumes pode ser observado na Figura 5.4. O *Blunt Fin* (a) e o *Oxygen Post* (b) são experiências da interação do oxigênio em determinado ambiente. O *SPX* (c) apresenta áreas de possíveis vazamentos em um reator. O *Combustion Chamber* (d) mede áreas com diferentes temperaturas em uma câmara de combustão. O *Fuel Injection* (e) simula a injeção de combustível em uma câmara de combustão. O *Brain Tomography* (f) é o resultado de uma tomografia do cérebro.

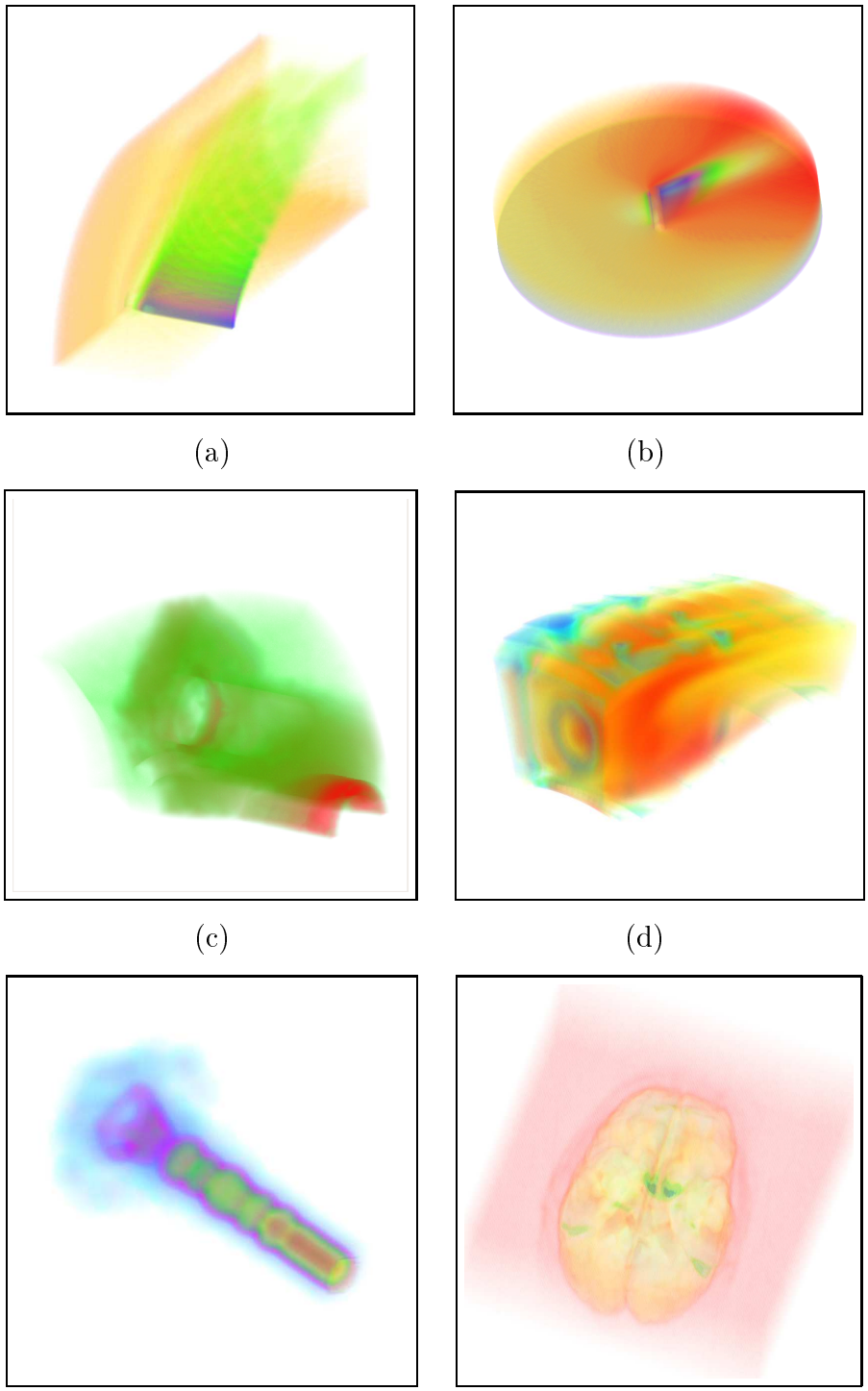

Figura 5.4: Visualização dos seguintes volumes: *Blunt Fin* (a), *Oxygen Post* (b), *SPX* (c), *Combustion Chamber* (d), *Fuel Injection* (e), *Brain Tomography* (f).

Capítulo 6

Conclusões

"Computers are useless.
They can only give you answers."

– Pablo Picasso

Nesta dissertação foi apresentado um método de projeção de células para visualização volumétrica que consegue usufruir completamente das vantagens das placas gráficas modernas. A implementação proposta alcança taxas de renderização acima de 2 milhões de tetraedros por segundo (2 M tets/s) com imagens de alta qualidade e, ao mesmo tempo, oferece controle interativo da função de transferência.

O algoritmo apresentado foi publicado no Simpósio Brasileiro de Computação Gráfica, Processamento de Imagens e Visão Computacional, SIBGRAPI [53], no ano de 2006 [54]. O artigo foi avaliado entre os 6 melhores artigos deste simpósio, que é o mais importante encontro brasileiro na área de Computação Gráfica.

O trabalho também foi aceito [55] em um *workshop* do Simpósio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho, SBAC-PAD [56].

Diferentemente dos algoritmos de projeção de tetraedros anteriores, o método proposto não sobrecarrega o barramento de transferência CPU – GPU pois guarda todo o volume em memória de textura na placa gráfica. Este armazenamento é menos custoso quando comparado aos algoritmos de traçado de raios, pois não mantém nenhuma estrutura auxiliar em memória. Como por exemplo, foi possível visualizar

um volume com 5.5 milhões de tetraedros usando uma placa gráfica com 256 MB de memória de textura.

A ordem de visibilidade continua sendo a verdadeira desvantagem do algoritmo. Por usar um método de ordenação aproximada durante a interação, artefatos são perceptíveis visualmente, como discutido no Capítulo 4.

Deve ser notado que as arquiteturas das placas gráficas atuais não possibilitam a manipulação da memória de GPU diretamente (escrita e leitura). Tais meios estão sendo planejados para a próxima geração de placas gráficas. Com isso, os dois passos do algoritmo proposto poderia ser reduzido em um único passo, melhorando substancialmente o desempenho.

Recentemente novas funcionalidades foram acrescidas às GPUs. A extensão dos VBOs (*Vertex Buffer Objects*) permite renderizar diretamente no vetor de vértices (*Vertex Array*) ao invés de renderizar em texturas com os FBOs (*Frame Buffer Objects*). Esta nova funcionalidade pode permitir a renderização dos dados dos tetraedros (no primeiro passo) direto nos vetores de vértices e cores, eliminando a organização de vetores realizada em CPU.

Algoritmos em GPU de ordenação, como o HAVS (*Hardware-Assisted Visibility Sorting*) de Callahan et al. [45], abrem uma possível extensão do algoritmo proposto trocando a ordenação de células em CPU por um passo intermediário também em GPU. Desta forma o algoritmo estendido teria 3 passos em GPU.

A computação e interpolação dos escalares são uma aproximação da variação do campo escalar dentro do tetraedro. O gradiente do campo pode ser utilizado para melhorar a qualidade na geração das imagens.

Além da visualização volumétrica, o realce das iso-superfícies aumenta a qualidade das imagens geradas. Como no algoritmo HAVIS (*Hardware-Accelerated Volume and Isosurface Rendering*) de Roettger et al. [12], uma solução híbrida pode ser empregada para estender o algoritmo proposto.

O gradiente do campo combinado com a visualização híbrida de iso-superfícies possibilitam o emprego de um modelo de iluminação mais complexo. Em tal modelo, novas funcionalidades podem ser acrescentadas na implementação do algoritmo proposto. Como por exemplo, a alteração da posição e intensidade da fonte de luz.

Em resumo, há diversas extensões e melhorias possíveis. Trabalhos decorrentes desta dissertação podem melhorar tanto em desempenho, como em qualidade e interface.

A franca expansão das funcionalidades da GPU demonstram o nascimento de um novo paradigma em Computação Gráfica. O estudo e exploração deste paradigma formam a força de evolução das placas gráficas. Em um futuro próximo e excitante pode ser possível que qualquer algoritmo de Computação Gráfica possa ser implementado inteiramente em GPU.

Referências Bibliográficas

[1] Kaufman, A. E. Volume visualization. *ACM Comput. Surv. 28*, 1 (1996), 165–167.

[2] Kaufman, A., and Mueller, K. Overview of volume rendering. *Chapter for The Visualization Handbook* (2005).

[3] Foley, van Dam, Feiner, and Hughes. *Computer Graphics Principles and Practice*, second ed. Addison-Wesley, 1996.

[4] Moreland, K. D. *Fast High Accuracy Volume Rendering*. Tese de Doutorado, The University of New Mexico, Albuquerque, New Mexico, July 2004.

[5] Crawfis, R., Max, N., Becker, B., and Cabral, B. Volume rendering of 3d scalar and vector fields at llnl. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing* (New York, NY, USA, 1993), ACM Press, pp. 570–576.

[6] Lichtenbelt, B., Crane, R., and Naqvi, S. *Introduction to Volume Rendering*, first ed. Hewlett-Packard, 1998.

[7] 3Dlabs. OpenGL Shading Language, July 2002. http://developer.3dlabs.com/openGL2/index.htm/.

[8] 3dlabs, April 1994. http://www.3dlabs.com/.

[9] Espinha, R. Visualização interativa de malhas não-estruturadas utilizando placas gráficas programáveis. Dissertação de Mestrado, Pontifícia Universidade Católica do Rio de Janeiro, Março 2005.

[10] Lorensen, W. E., and Cline, H. E. Marching cubes: A high resolution 3d surface construction algorithm. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1987), ACM Press, pp. 163–169.

[11] Max, N., Hanrahan, P., and Crawfis, R. Area and volume coherence for efficient visualization of 3d scalar functions. In *VVS '90: Proceedings of the 1990 workshop on Volume visualization* (New York, NY, USA, 1990), ACM Press, pp. 27–33.

[12] Roettger, S., Kraus, M., and Ertl, T. Hardware-accelerated volume and isosurface rendering based on cell-projection. In *VIS '00: Proceedings of the conference on Visualization '00* (Los Alamitos, CA, USA, 2000), IEEE Computer Society Press, pp. 109–116.

[13] Engel, K., Kraus, M., and Ertl, T. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (New York, NY, USA, 2001), ACM Press, pp. 9–16.

[14] Espinha, R., and Celes, W. High-quality hardware-based ray-casting volume rendering using partial pre-integration. In *SIBGRAPI '05: Proceedings of the XVIII Brazilian Symposium on Computer Graphics and Image Processing* (2005), IEEE Computer Society, p. 273.

[15] Upson, C., and Keeler, M. V-buffer: visible volume rendering. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1988), ACM Press, pp. 59–64.

[16] Sabella, P. A rendering algorithm for visualizing 3d scalar fields. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1988), ACM Press, pp. 51–58.

[17] Max, N. L. Vectorized procedural models for natural terrain: Waves and islands in the sunset. In *SIGGRAPH '81: Proceedings of the 8th annual confe-*

rence on Computer graphics and interactive techniques (New York, NY, USA, 1981), ACM Press, pp. 317–324.

[18] Blinn, J. F. Light reflection functions for simulation of clouds and dusty surfaces. In *SIGGRAPH '82: Proceedings of the 9th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1982), ACM Press, pp. 21–29.

[19] Max, N. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics 1*, 2 (1995), 99–108.

[20] Williams, P. L., Max, N. L., and Stein, C. M. A high accuracy volume renderer for unstructured data. *IEEE Transactions on Visualization and Computer Graphics 4*, 1 (1998), 37–54.

[21] Shirley, P., and Tuchman, A. A. Polygonal approximation to direct scalar volume rendering. In *Proceedings San Diego Workshop on Volume Visualization, Computer Graphics* (1990), vol. 24(5), pp. 63–70.

[22] Wylie, B., Moreland, K., Fisk, L. A., and Crossno, P. Tetrahedral projection using vertex shaders. In *VVS '02: Proceedings of the 2002 IEEE Symposium on Volume visualization and graphics* (Piscataway, NJ, USA, 2002), IEEE Press, pp. 7–12.

[23] Roettger, S., and Ertl, T. A two-step approach for interactive pre-integrated volume rendering of unstructured grids. In *VVS '02: Proceedings of the 2002 IEEE Symposium on Volume visualization and graphics* (Piscataway, NJ, USA, 2002), IEEE Press, pp. 23–28.

[24] Moreland, K., and Angel, E. A fast high accuracy volume renderer for unstructured data. In *VVS '04: Proceedings of the 2004 IEEE Symposium on Volume visualization and graphics* (Piscataway, NJ, USA, 2004), IEEE Press, pp. 13–22.

[25] Moreland, K., and Angel, E. ψ table, 2004. http://www.cs.unm.edu/ kmorel/documents/volvis2004/.

[26] Garrity, M. P. Raytracing irregular volume data. In *VVS '90: Proceedings of the 1990 workshop on Volume visualization* (New York, NY, USA, 1990), ACM Press, pp. 35–40.

[27] Bunyk, P., Kaufman, A. E., and Silva, C. T. Simple, fast, and robust ray casting of irregular grids. In *Dagstuhl '97, Scientific Visualization* (Washington, DC, USA, 1997), IEEE Computer Society, pp. 30–36.

[28] Weiler, M., Kraus, M., Merz, M., and Ertl, T. Hardware-based ray casting for tetrahedral meshes. In *VIS '03: Proceedings of the 14th IEEE conference on Visualization '03* (2003), pp. 333–340.

[29] de Berg, M., van Kreveld, M., Overmars, M., and Schwarzkopf, O. *Computational Geometry: Algorithms and Applications*, second ed. Springer, 2000.

[30] Giertsen, C. Volume visualization of sparse irregular meshes. *IEEE Comput. Graph. Appl. 12*, 2 (1992), 40–48.

[31] Silva, C. T. *Parallel Volume Rendering of Irregular Grids*. Tese de Doutorado, State University of New York, Stony Brook, November 1996.

[32] Silva, C., Mitchell, J. S. B., and Kaufman, A. E. Fast rendering of irregular grids. In *VVS '96: Proceedings of the 1996 symposium on Volume visualization* (Piscataway, NJ, USA, 1996), IEEE Press, pp. 15–ff.

[33] Silva, C. T., and Mitchell, J. S. B. The lazy sweep ray casting algorithm for rendering irregular grids. *IEEE Transactions on Visualization and Computer Graphics 3*, 2 (1997), 142–157.

[34] Farias, R., Mitchell, J. S. B., and Silva, C. T. Zsweep: an efficient and exact projection algorithm for unstructured volume rendering. In *VVS '00: Proceedings of the 2000 IEEE Symposium on Volume visualization* (New York, NY, USA, 2000), ACM Press, pp. 91–99.

[35] Wittenbrink, C. M. Cellfast: Interactive unstructured volume rendering. In *HP White Paper* (1999), Hewlett-Packard Labs.

[36] SGI. OpenGL, 1992. http://www.opengl.org/.

[37] Wilhelms, J., and Gelder, A. V. A coherent projection approach for direct volume rendering. In *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1991), ACM Press, pp. 275–284.

[38] Kraus, M., Qiao, W., and Ebert, D. S. Projecting tetrahedra without rendering artifacts. In *VIS '04: Proceedings of the conference on Visualization '04* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 27–34.

[39] Weiler, M., Kraus, M., , and Ertl, T. Hardware-based view-independent cell projection. In *VVS '02: Proceedings of the 2002 IEEE Symposium on Volume visualization and graphics* (Piscataway, NJ, USA, 2002), IEEE Press, pp. 13–22.

[40] Weiler, M., Kraus, M., Merz, M., and Ertl, T. Hardware-based view-independent cell projection. *IEEE Transactions on Visualization and Computer Graphics 9*, 2 (2003), 163–175.

[41] Harris, M. General-Purpose computation using Graphics Hardware, 2004. http://www.gpgpu.org/.

[42] Williams, P. L. Visibility-ordering meshed polyhedra. *ACM Trans. Graph. 11*, 2 (1992), 103–126.

[43] Stein, C., Becker, B., and Max, N. Sorting and hardware assisted rendering for volume visualization. In *1994 Symposium on Volume Visualization* (1994), A. Kaufman and W. Krueger, Eds., pp. 83–90.

[44] Comba, J., Klosowski, J. T., Max, N. L., Mitchell, J. S. B., Silva, C. T., and Williams, P. L. Fast polyhedral cell sorting for interactive rendering of unstructured grids. *Computer Graphics Forum 18*, 3 (1999), 369–376.

[45] Callahan, S. P., and Comba, J. L. D. Hardware-assisted visibility sorting for unstructured volume rendering. *IEEE Transactions on Visualization and*

Computer Graphics 11, 3 (2005), 285–295. Student Member-Milan Ikits and Member-Claudio T. Silva.

[46] Kilgard, M. J. *OpenGL Extension Specification*. NVIDIA Corporation, 2001.

[47] Woo, M., Neider, J., Davis, T., and Shreiner, D. *OpenGL Programming Guide*, third ed. Addison Wesley, 1999.

[48] Rost, R. J. *OpenGL Shading Language*, second ed. Addison Wesley, 2006.

[49] NASA. NASA Advanced Supercomputing (NAS) Division, 2006. http://www.nas.nasa.gov/.

[50] Kitware. The Visualization Toolkit VTK, 2006. http://public.kitware.com/VTK/.

[51] LLNL. Lawrence Livermore National Laboratory, 2006. http://www.llnl.gov/.

[52] VolVis. Volume Visualization, 2006. http://www.volvis.org/.

[53] UFAM. Simpósio Brasileiro de Computação Gráfica, Processamento de Imagens e Visão Computacional, 2006. http://www.sibgrapi.ufam.edu.br/index.php?lang=pt_BR.

[54] Marroquim, R., Maximo, A., Farias, R., and Esperanca, C. GPU-Based Cell Projection for Interactive Volume Rendering. In *SIBGRAPI '06: Proceedings of the XIX Brazilian Symposium on Computer Graphics and Image Processing* (Los Alamitos, CA, USA, 2006), IEEE Computer Society, pp. 147–154.

[55] Maximo, A., Marroquim, R., Esperanca, C., dos Santos, R. W., Bentes, C., and Farias, R. High-Performance Volume Rendering for 3D Heart Visualization. In *SBAC-PAD '06: Workshop on High Performance Computing in the Life Science* (2006).

[56] UFMG. Simpósio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho, 2006. http://www.sbc.org.br/sbac/2006/index.htm.