Renan Teston Inácio

Visualização interativa de volumes com GPU

Florianópolis

2009

Renan Teston Inácio

Visualização interativa de volumes com GPU

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Ciências da Computação

Orientador:
Prof. Dr. rer.nat. Aldo von Wangenheim

UNIVERSIDADE FEDERAL DE SANTA CATARINA

Florianópolis

2009

Resumo

Conjuntos de dados volumétricos estão presentes em áreas científicas e médicas como resultados de simulação ou dados capturados por dispositivos específicos. A visualização desses dados lida com o problema de representar os volumes, que são tridimensionais, em uma imagem a ser exibida em um *display* bidimensional. Para isso, algumas técnicas de renderização volumétrica presentes na literatura foram analisadas, incluindo as que exploram o poder computacional dos processadores gráficos, já que este trabalho trata de visualização interativa. A técnica de *ray-casting* foi implementada para ser executada em processador convencional e então para processador gráfico a fim de comparar os resultados visuais e de performance. Aproveitando a alta performance obtida na implementação em GPU, foram adicionados cálculos de iluminação para melhorar a qualidade visual da renderização. Este trabalho apresenta na seção de conclusões as dificuldades e limitações encontradas em cada implementação e trabalhos futuros.

Abstract

Volumetric data sets are present in scientific and medical areas as results of simulation or data captured by specific devices. The visualization of these data sets deals with the problem of representing the volumes, which are tridimensional, on an image to be viewed on a bidimensional display. For this, some volumetric rendering techniques present in the literature were analyzed, including the ones that explore the computing power of graphic processors, since this work is about interactive visualization. The ray-casting technique was implemented to be executed on conventional processors and then for graphic processors in order to compare visual and performance results. Taking advantage of the performance obtained in the GPU implementation, illumination calculations were also implemented in order to increase rendering visual quality. This work presents in the conclusions section the difficulties and limitations faced in each implementation and future works.

Lista de Algoritmos

Lista de Tabelas

Lista de Figuras

Sumário

1 Introdução

Segundo Foley et al. (1996), a computação gráfica é uma disciplina que trata da síntese gráfica de objetos reais ou imaginários a partir de modelos computacionais. A forma de representação final são informações que possam ser visualizadas em um dispositivo periférico, como imagens em um monitor.

O processo de transformação do modelo computacional para a imagem que será exibida é conhecido como renderização. Renderização em tempo-real significa produzir rapidamente essas imagens. Uma imagem aparece na tela, o usuário reage e isso afeta o que é gerado em seguida. Se este ciclo de reação e renderização acontece em um período suficientemente pequeno, de forma que o usuário não vê imagens individuais e fica imerso em um processo dinâmico, diz-se que existe interatividade. Normalmente quando se fala em renderização em tempo-real, refere-se à renderização de modelos tridimensionais (MÖLLER; HAINES; HOFFMAN, 2008).

Um dos modelos computacionais para representação de objetos é o de primitivas geométricas, através de pontos, linhas e polígonos. A Figura 1.1 mostra um objeto tridimensional representado por quadriláteros e triângulos formando a imagem de um macaco. Esta é a forma mais comum de representação em aplicações interativas, pois estas primitivas são suportadas pelas placas de aceleração gráfica. Embora não sejam necessárias, considera-se que estas placas gráficas sejam um requisito para a maior parte das aplicações que realizam renderização em tempo-real (MÖLLER; HAINES; HOFFMAN, 2008).

Por outro lado, muitos efeitos visuais são de natureza volumétrica. Fluidos, fogo, fumaça, neblina e poeira são difíceis de modelar com primitivas geométricas. Alguns desses efeitos podem ser visualizados na Figura 1.2. Além da representação desses fenômenos, visualização volumétrica é essencial para aplicações científicas e de engenharia que requerem visualização de conjuntos de dados tridimensionais. Exemplos incluem imagens médicas capturadas por dispositivos específicos e resultados de simulação de dinâmica de fluidos (FERNANDO, 2004).

De acordo com Elvins (1992), os volumes normalmente são estruturas que armazenam muita informação, aumentando o desafio de conseguir renderizá-los rapidamente. Porém, com

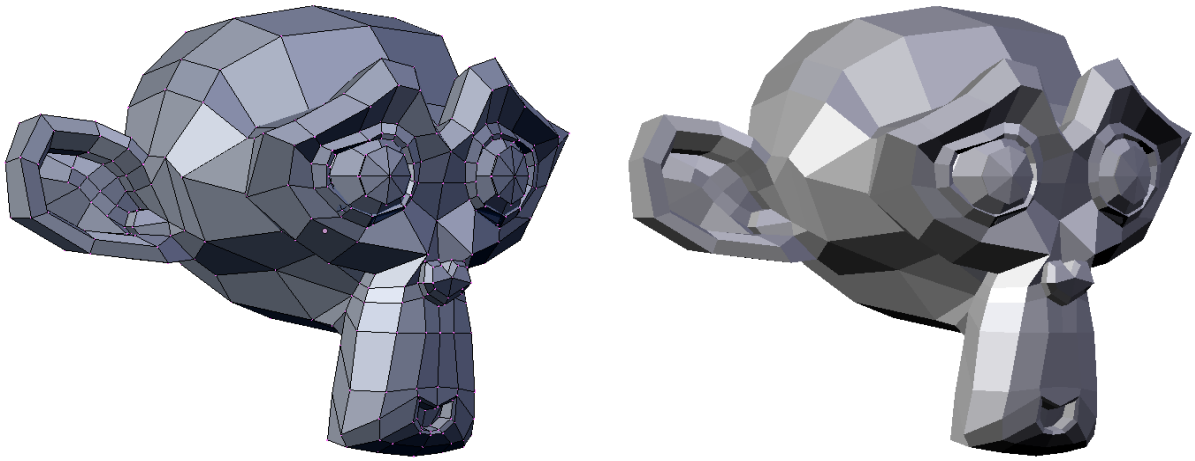


Figura 1.1: Representação através de primitivas geométricas.

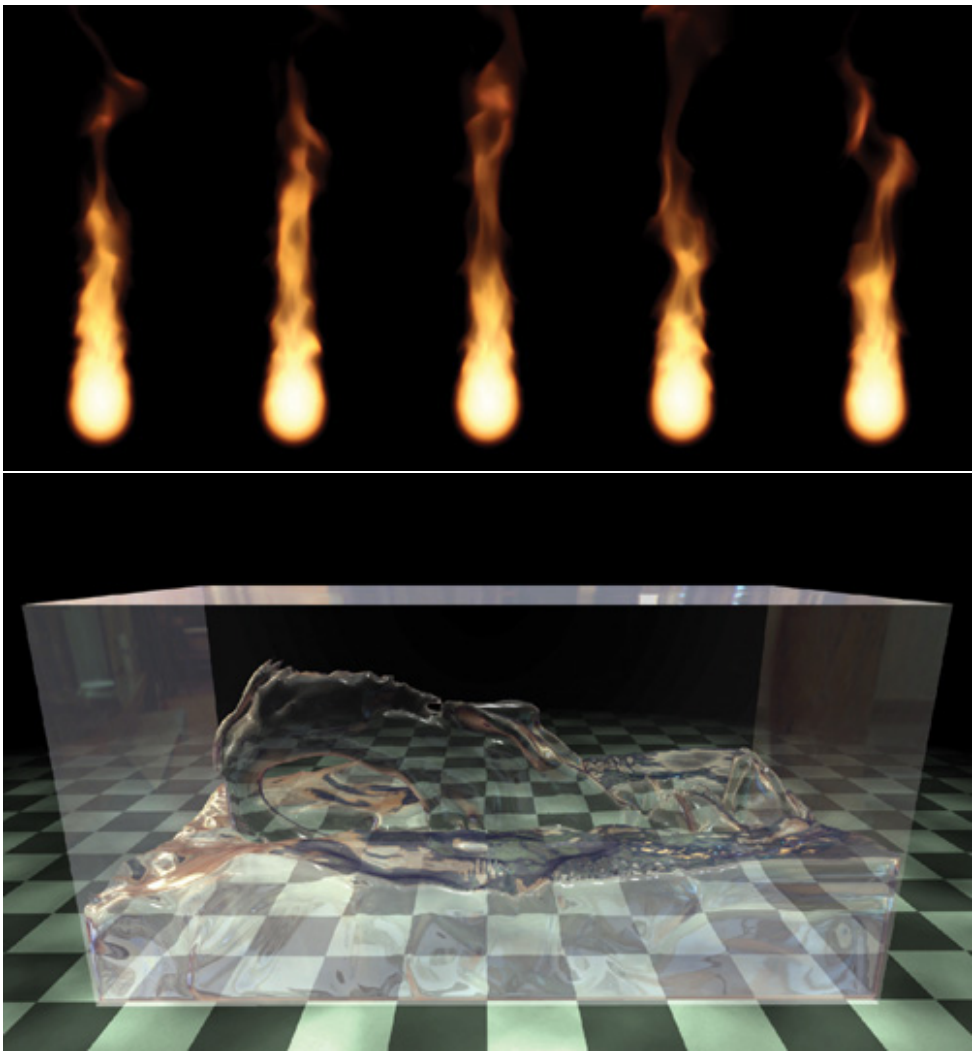


Figura 1.2: Resultados de simulações de fogo e de fluido (NGUYEN, 2007).

o avanço nas tecnologias de processadores gráficos no começo desta década, começou a ser possível realizar renderização com performance interativa (ENGEL; KRAUS; ERTL, 2001; KRUGER; WESTERMANN, 2003), sendo esta a maior motivação deste trabalho.

1.1 Objetivos gerais

Obter performance interativa e com qualidade visual aceitável em renderização volumétrica explorando a capacidade dos processadores gráficos.

1.2 Objetivos específicos

- Analisar as técnicas de renderização volumétrica na literatura, extraindo as características mais importantes dos algoritmos.
- Implementar alguma dentre as técnicas descritas em processador de uso geral e em processador gráfico e medir a performance das implementações.
- Analisar os resultados de performance obtidos, a qualidade das imagens geradas e as limitações de cada implementação.

2 *Renderização volumétrica*

A operação de transformar os modelos computacionais para uma imagem que será exibida em tela é específica para cada modelo. No caso de volumes, Elvins (1992) divide as principais técnicas classificando-as como renderização direta ou ajuste de superfície.

Os algoritmos de ajuste de superfície, também conhecidos como de extração de superfícies, tipicamente criam primitivas de superfície, como polígonos, nas regiões de contorno dentro do volume. O contorno é determinado através da especificação de um valor de referência, sendo que existe contorno entre duas regiões vizinhas se uma delas está acima deste valor de referência e a outra está abaixo. Como este contorno normalmente é formado por polígonos, são usadas as técnicas de renderização de primitivas geométricas. Segundo Dietrich et al. (2008), o algoritmo mais popular de extração de superfície é o Marching Cubes (LORENSEN; CLINE, 1987).

As técnicas de renderização direta, por outro lado, exibem os dados do volume através da avaliação de um modelo ótico que descreve como o volume emite, reflete, espalha, absorve e bloqueia a luz (MAX, 1995). Dentre os algoritmos nesta classificação, podem ser citados o *ray-casting* (LEVOY, 1988) e o *splatting* (WESTOVER, 1990).

Neste capítulo será aprofundado o conceito de volume e os conceitos comuns em renderização volumétrica, além de detalhar as técnicas de renderização.

2.1 Volume

Volume é uma estrutura de dados organizada em forma de matriz tridimensional, ou grade, onde cada elemento é denominado *voxel*, como ilustra a Figura 2.1. Este trabalho considera que o *voxel* possui apenas um valor escalar associado a ele. A semântica deste valor varia conforme a aplicação, mas normalmente é um valor de densidade, seja de um tecido humano, estruturas mecânicas ou fluidos. Dizemos que a resolução do volume está relacionada com a quantidade de *voxels* que ele possui.

Os valores do volume podem ser obtidos de diversas maneiras, dependendo da aplicação.

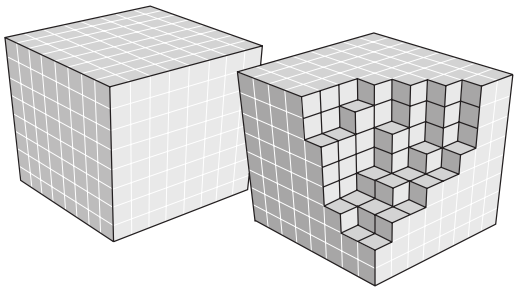
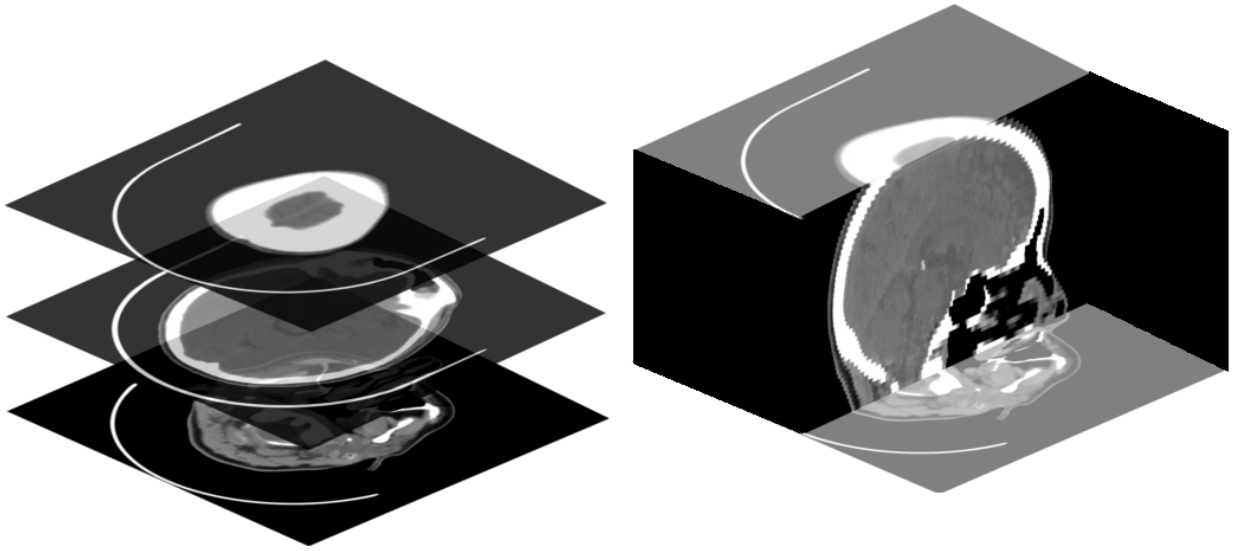


Figura 2.1: *Voxels* constituindo um volume (HADWIGER et al., 2008).



(a) Imagens obtidas por CT (b) Corte no volume resultante formado por 50 imagens

Figura 2.2: Empilhamento de imagens para formação de um volume.

Hadwiger et al. (2006) listam algumas características dos diferentes tipos de fonte de dados: dispositivos de aquisição de imagens médicas, simulação de fenômenos físicos, voxelização e modelagem procedural.

Para aplicações de imagens médicas, os métodos mais comuns para aquisição de dados tridimensionais são por tomografia computadorizada (CT) e através de ressonância magnética (MRI) (HADWIGER et al., 2006). No caso de CT, o processo de captura é baseado em raios X. Os raios são emitidos para o corpo do paciente de um lado e a radiação que atravessa o corpo é gravada no outro lado. Através da interação entre raios X e os diferentes materiais do corpo do paciente, a radiação é atenuada. O emissor de radiação é rotacionado em volta do paciente para obter os dados de atenuação em diferentes direções. São criadas então várias imagens que quando empilhadas formam um volume, como mostra a Figura 2.2.

Volumes como resultados de simulação são outra classe de fontes de dados. No caso de simulação de dinâmica de fluidos utilizando o método de Smoothed Particles Hydrodynamics (SPH) (GINGOLD; MONAGHAN, 1977), por exemplo, as partículas do fluido são representadas por esferas que possuem um valor de densidade. O valor de densidade em cada ponto do fluido é calculado com base na densidade das esferas próximas a esse ponto, através de um *kernel* de suavização. Pode ser criado então um volume a partir da amostragem direta desses valores de densidade.

Outras fontes de dados incluem voxelização e modelagem procedural. Voxelização é transformar uma forma de representação, como o de primitivas geométricas, para representação volumétrica. A acurácia da voxelização pode ser controlada alterando a resolução do volume. Já a modelagem procedural descreve valores para qualquer ponto no espaço através de algoritmo, ou seja, o volume pode ser obtido através de amostragens do resultado dos algoritmos.

2.2 Características de visualização

Como exposto no começo deste capítulo, é possível classificar as técnicas de renderização em diretas e indiretas. Nesta seção serão abordados conceitos e etapas que essas classes possuem em comum.

2.2.1 Classificação de dados

Segundo Elvins (1992), esta é a etapa mais difícil para o usuário que deseja visualizar um volume. Os algoritmos de ajuste de superfície precisam de um valor de referência para indicar os valores de *voxel* que a superfície separa. Já os algoritmos de renderização direta necessitam de uma função de transferência que transforma valores de *voxel* em valores de opacidade e outros necessários para o cálculo de iluminação, como a cor do material e propriedades de reflexão. Esta função pode estar expressa como uma tabela.

Nesta etapa pode-se perceber a maior flexibilidade da renderização direta. A exibição dos dados não fica limitada a mostrar uma única superfície, já que é possível especificar vários níveis de opacidade. Além disso, materiais diferentes, identificados por valores de *voxel* diferentes, podem ser diferenciados por propriedades visuais diferentes.

A Figura 2.3 ilustra como o processo de classificação pode ser feito em volumes feitos a partir de imagens de CT. O histograma mostra quantos *voxels* estão presentes no volume para cada valor de densidade. A distribuição de constituição indica o intervalo de valores que re-

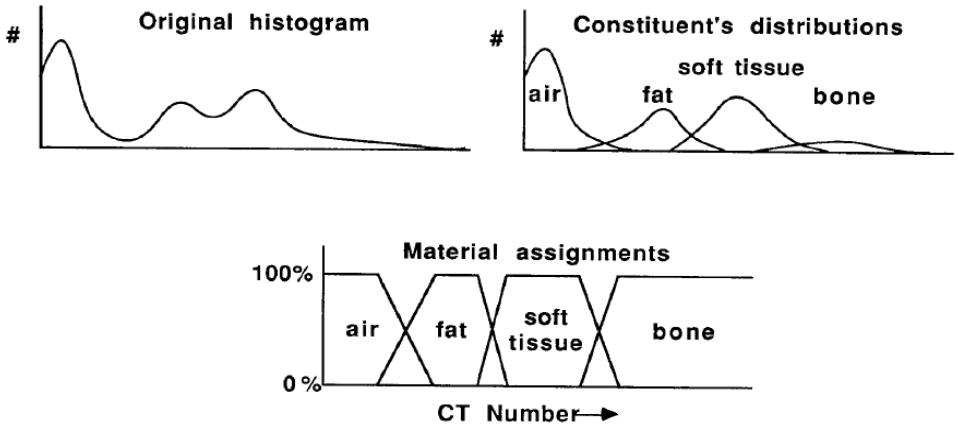


Figura 2.3: Classificação de materiais em CT (DREBIN; CARPENTER; HANRAHAN, 1988).

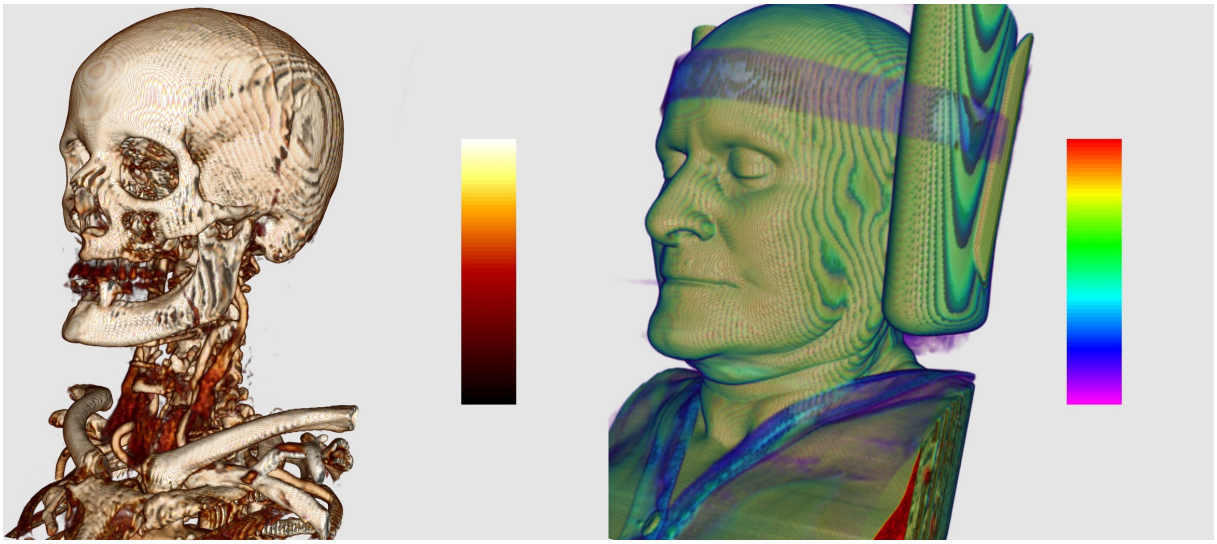


Figura 2.4: Volume obtido por CT renderizado com diferentes funções de transferência.

presenta cada material: ar, gordura, alguns tecidos conjuntivos e ossos. Drebin, Carpenter e Hanrahan (1988) notam que por causa das características da captura de imagens, alguns materiais se sobrepõem, ou seja, existem valores de *voxel* que representam mais de um material. Por esta razão, é recomendável que a função de transferência tenha transições suaves. O último gráfico mostra como seria a classificação final, assumindo valores de cor e opacidade para cada material, misturando linearmente essas propriedades nas transições.

Na Figura 2.4, o retângulo preenchido com um gradiente ao lado das imagens ilustra as cores usadas na função de transferência. Para determinar a opacidade, foi considerado um intervalo de densidades tal que todas as cores estão presentes e com opacidade total em maior parte dele. Fora deste intervalo, a opacidade é nula, existindo uma zona de transição suave.

2.2.2 Iteração

Após feita a classificação de dados, as imagens podem ser criadas de duas maneiras: ou por iteração em ordem de imagem, percorrendo os *pixels*, ou em ordem de objeto, percorrendo os elementos do volume. Segundo Elvins (1992), alguns algoritmos usam uma combinação dessas duas formas.

Iteração em ordem de imagem pode seguir uma ordem sequencial entre os *pixels*, linha a linha, ou ainda em ordem aleatória, para que o usuário possa assistir a imagem sendo refinada.

Iteração em ordem de objeto permite duas opções para percorrer os elementos do volume: de frente para trás ou trás para frente, onde definimos como o lado da frente o lado mais próximo do observador. A vantagem de percorrer de frente para trás é que os elementos atrás não precisam ser percorridos se os da frente já formaram uma imagem opaca. Já na iteração de trás para frente, o usuário pode ver a imagem progredindo e visualizar estruturas que serão ocultas pelos elementos da frente.

2.2.3 Composição

Ao tratar renderização volumétrica direta, onde pode haver sobreposição de materiais translúcidos, é necessário realizar a composição para gerar a imagem final. A operação de composição *over* foi definida por Porter e Duff (1984) através da equação 2.1, onde C_A e C_B são as cores a serem compostas e α_A é a opacidade associada ao C_A. Note que em imagens com múltiplos canais de cores esta mesma equação é aplicada em cada canal, porém usando o mesmo valor de opacidade.

$$C_A \text{ over } C_B = C_A + (1 - \alpha_A) \times C_B \tag{2.1}$$

Com este operador, pode ser feita uma composição de trás para frente (HADWIGER et al., 2008), considerando como frente o elemento mais próximo do observador. A equação 2.2 compõe o fragmento de cor c_i e opacidade a_i com o resultado da composição anterior C_{i+1}, resultando em C_i. O i é iterado de $n-1$ a 0 e $C_n = 0$.

$$C_i = c_i + (1 - a_i) \times C_{i+1} \tag{2.2}$$

Outra opção é compor de frente para trás. Para isto, a opacidade também precisa ser composta e armazenada durante a composição dos canais de cores. Nas equações 2.3 e 2.4 o i é

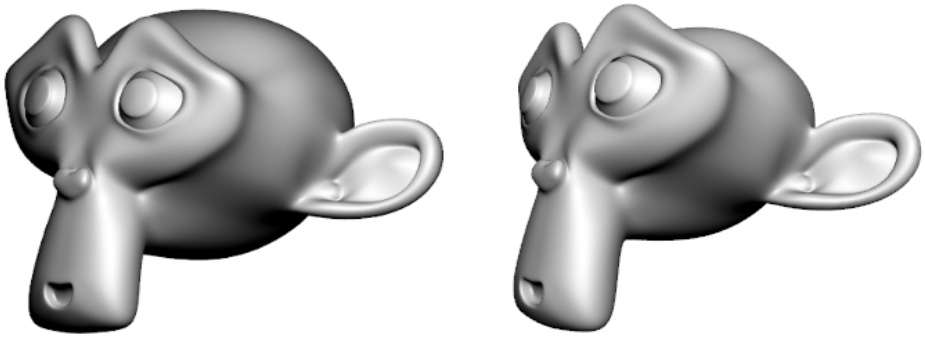


Figura 2.5: Exemplos de projeção paralela e em perspectiva.

iterado de 1 a n, com $C_0 = 0$ e $A_0 = 0$, onde A_i é a composição da opacidade no passo i.

$$C_i = C_{i-1} + (1 - A_{i-1}) \times c_i \tag{2.3}$$

$$A_i = A_{i-1} + (1 - A_{i-1}) \times a_i \tag{2.4}$$

Em todas as equações acima, a cor c_i está pré-multiplicada pela sua opacidade correspondente a_i. Esta notação, chamada de *opacity-weighted colors* é descrita por Wittenbrink, Malzbender e Goss (1998).

2.2.4 Visualização

Segundo Möller, Haines e Hoffman (2008), a projeção mais simples é a projeção paralela, embora a projeção em perspectiva seja a mais usada em aplicações de computação gráfica. A Figura 2.5 mostra exemplos dessas projeções. O autor mostra também que essas projeções são implementadas e aceleradas em placas gráficas modernas. Como visto no Capítulo 1 as placas gráficas lidam nativamente com primitivas geométricas, logo os algoritmos de ajuste de superfície não precisam lidar com a questão de projeção. A projeção perspectiva é interessante por dar maior noção de profundidade, mas Elvins (1992) nota que para visualização científica a projeção paralela assegura que não haverá confusão por causa de distorções. O autor sugere outras formas para acentuar profundidade como através de iluminação.

A operação de determinar o efeito da luz em um material é conhecido como *shading*, ou iluminação. Esta é uma etapa importante para gerar imagens compreensíveis, tanto em visualização de volumes como outras aplicações de computação gráfica (ELVINS, 1992). Segundo Max (1995), o modelo mais simples de iluminação é simplesmente tornar os *pixels* distantes mais

Figura 2.6: Modelo de *shading* dependente apenas da profundidade do *pixel*.

escuros, como mostra a Figura 2.6, porém modelos mais realistas precisam levar em consideração o vetor normal do ponto correspondente no volume, que é igual ao gradiente normalizado deste ponto.

2.3 Marching Cubes

O Marching Cubes cria uma superfície a partir dos dados do volume e um valor de referência. A idéia é que a superfície passará nos *voxels* que possuam esse valor de referência. O algoritmo considera interpolação entre os *voxels*, localizando assim a superfície mesmo quando os *voxels* não possuem exatamente o valor de referência especificado.

De forma resumida, o algoritmo analisa a intensidade entre *voxels* vizinhos e compara com o valor de referência. De acordo com o resultado dessas comparações, uma tabela é consultada para determinar os triângulos a serem criados, posiciona os vértices e calcula suas normais.

2.3.1 Detalhes do algoritmo

A cada passo do algoritmo é analisado um cubo formado por oito *voxels* vizinhos, como mostra a Figura 2.7. Cada *voxel* representa um vértice do cubo, sendo que este vértice é marcado de acordo com o valor do *voxel* correspondente. Se o valor do *voxel* está abaixo do valor de referência marca-se um valor negativo ao vértice, senão um valor positivo. Desta forma a superfície passa em uma aresta do cubo, havendo uma intersecção, se os vértices possuem marcações diferentes.

Como o cubo é formado por 8 vértices e cada vértice pode assumir 2 estados, existem $2^8 =$

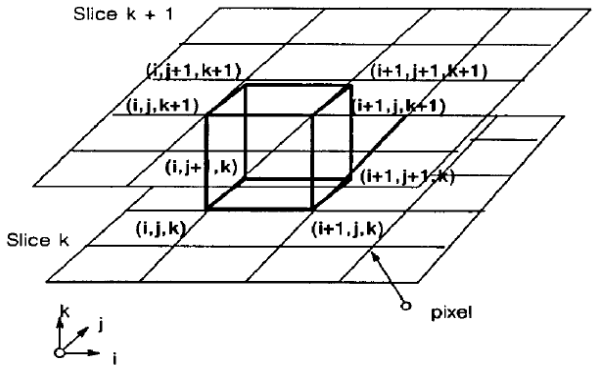


Figura 2.7: Cubo posicionado entre duas fatias do volume (LORENSEN; CLINE, 1987).

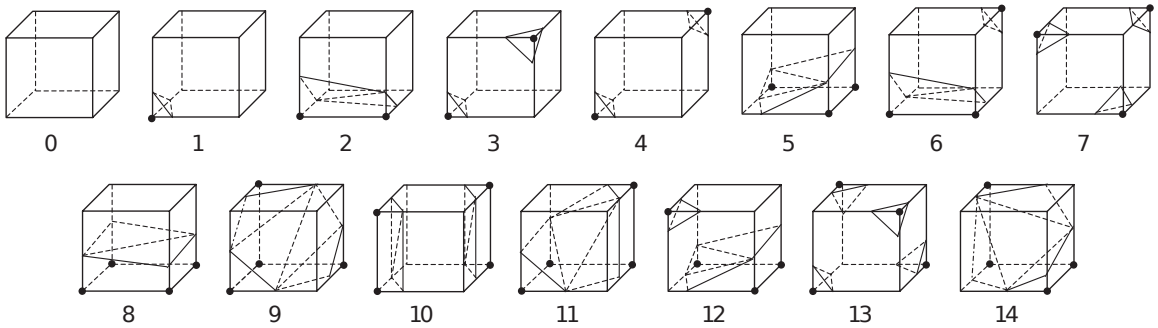


Figura 2.8: Casos de configuração para o Marching Cubes original (NEWMAN; YI, 2006).

256 possíveis configurações de vértices para um cubo. Pode ser então criada uma tabela que diga as intersecções da superfície com o cubo dada a configuração. Note que para determinação da superfície, o valor de marcação dos vértices é usado apenas para comparar com o de outro vértice, logo só importa se eles são diferentes ou iguais. Essa simetria permite uma redução para 128 configurações. Além disso, existe simetria também quanto à rotação do cubo. Levando isso em conta, Lorensen e Cline (1987) reduziram o número de casos para apenas 15, ilustrados na Figura 2.8.

Após obtidas as intersecções da superfície com as arestas, são criados os vértices cujas posições são determinadas através da interpolação linear pelos valores dos *voxels*. As normais destes vértices são obtidas por interpolação dos valores de gradiente dos mesmos *voxels*.

2.3.2 Análise

Segundo Chernyaev (1995), o Marching Cubes pode gerar resultados topologicamente incorretos em algumas situações e mostra que existem 33 configurações topologicamente diferentes ao invés de apenas 15. O autor propõe as modificações necessárias e chama o algoritmo de Marching Cubes 33.

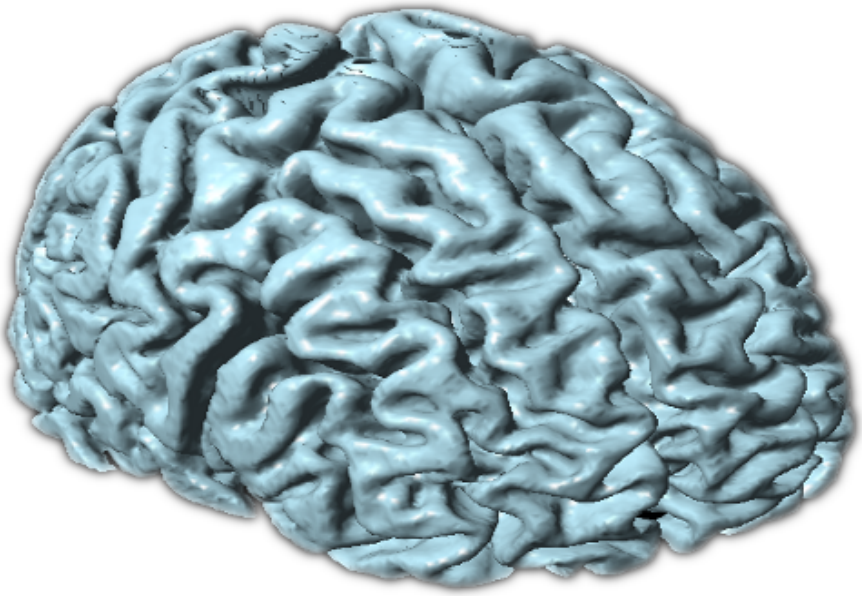


Figura 2.9: Superfície obtida por Marching Cubes.

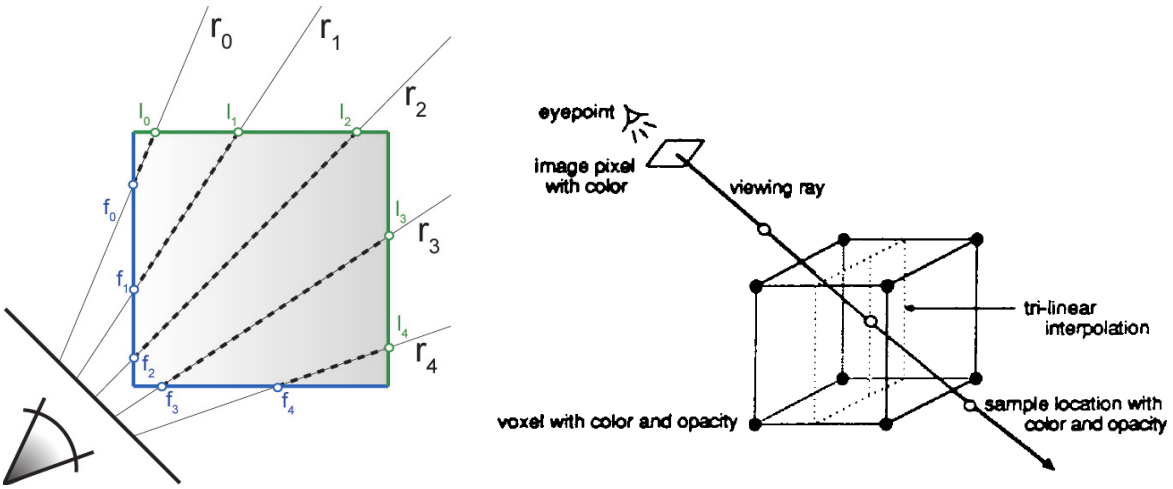
Após extraída a superfície na forma de malha de triângulos, pode-se ignorar as informações volumétricas e aplicar algoritmos apenas nesta malha de triângulos. Por outro lado, a transformação em superfície pode não ser adequada dependendo das características do volume, como por exemplo ao tentar representar fumaça.

O algoritmo exibe um grau de paralelismo intrínseco e oferece potencial para ganhos de performance neste sentido. Newman e Yi (2006) listam diversas abordagens.

A Figura 2.9 mostra o resultado da aplicação do Marching Cubes em um volume cujos dados foram obtidos por MRI. Esta imagem foi gerada pelo algoritmo implementado nas bibliotecas do LAPIX (SILVA et al., 2009).

2.4 Ray-casting

O algoritmo de *ray-casting*, ou de lançamento de raios, renderiza o conteúdo do volume diretamente para uma imagem. Um raio é disparado em direção ao volume a partir de cada *pixel* da imagem. A cor de um determinado *pixel* é calculada a partir dos valores dos *voxels* que o raio disparado intersectar.



(a) Disparo dos raios em direção ao volume (HADWIGER et al., 2008).

(b) Raio passando por um *voxel* (LEVOY, 1988).

Figura 2.10: Visão geral do algoritmo de *ray-casting*.

2.4.1 Detalhes do algoritmo

São disparados raios em direção ao volume a fim de coletar amostras. A origem desses raios é o plano de visão do observador, sendo que cada raio corresponde a um pixel na imagem a ser renderizada. As amostras são coletadas com um espaçamento constante ao longo do raio e o valor é obtido a partir da interpolação trilinear dos valores dos *voxels* vizinhos. A Figura 2.10 esquematiza a visão geral do algoritmo.

Para cada raio disparado, são realizadas 3 etapas para determinar a cor final do *pixel*: *shading*, classificação e composição. As etapas de *shading* e classificação são independentes e aplicadas em cada amostra obtida pelo raio. A etapa de composição une os resultados das etapas anteriores e define a cor final do *pixel*.

A etapa de *shading* consiste em obter um valor de cor para cada amostra através de um modelo de iluminação, além dos atributos necessários pelo modelo que podem ser obtidos pela função de transferência.

A etapa de classificação tem como propósito definir a contribuição das amostras para a imagem final através do valor de opacidade obtido pela função de transferência.

Por fim, tendo a cor e a opacidade de cada amostra, é realizada a etapa de composição. Nesta etapa, Levoy (1988) faz uma composição de trás para frente, ou seja, define uma cor para o plano de fundo e compõe as cores começando pela amostra mais distante até a mais próxima do observador. Também é possível fazer a composição na ordem de frente para trás facilmente (DREBIN; CARPENTER; HANRAHAN, 1988).

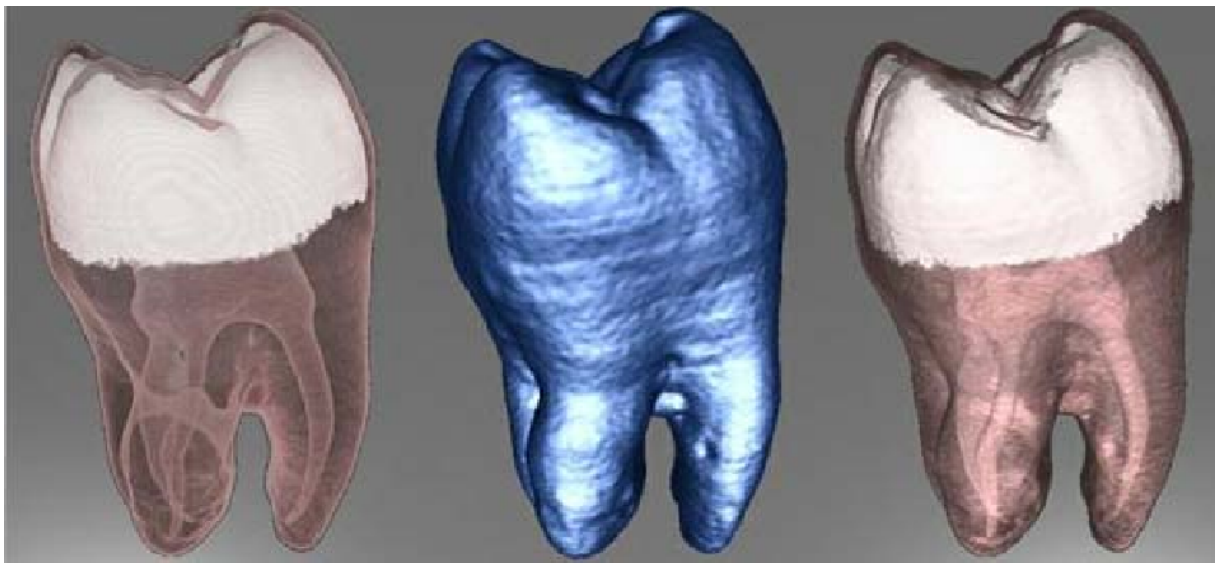


Figura 2.11: Renderização realizada por *ray-casting* com diferentes funções de transferência (KRUGER; WESTERMANN, 2003).

2.4.2 Análise

A Figura 2.11 exibe resultados criados por *ray-casting*. Note a maior flexibilidade de visualização que a função de transferência proporciona comparado ao Marching Cubes.

De acordo com Westover (1989), a interpolação trilinear realizada entre os *voxels* é um *kern*el de reconstrução pobre e que causa *aliasing*. Como a qualidade da imagem depende da quantidade de raios, é possível realizar *supersampling* para evitar artefatos (LEVOY, 1988).

Levoy (1990) descreve algumas otimizações que podem ser feitas, como término antecipado de raio e salto de espaços vazios no volume. Essa primeira otimização consiste em amostrar de frente para trás e parar de amostrar quando a opacidade acumulada for igual ou muito próximo de 1, pois neste caso as amostras mais atrás não contribuirão para a imagem final. A otimização de salto de espaços vazios tenta reduzir o número de amostragens nas partes do volume onde os *voxels* seriam classificados como totalmente transparentes.

Elvins (1992) nota que como o cálculo de cada raio é feito independentemente, o algoritmo pode ser paralelizável no nível de *pixel*.

2.5 Splatting

Pode ser feita uma analogia desta técnica com o ato de jogar bolas de neve em um plano de vidro. As bolas de neves seriam os *voxels* e o plano de vidro seria a imagem. A neve se espalha

quando acerta o plano de vidro, assim como o *voxel* espalha valores de *pixel* para uma região da imagem.

2.5.1 Detalhes do algoritmo

O algoritmo consiste nas seguintes etapas aplicadas para cada *voxel* em uma ordem pré-determinada. Esta ordem pode ser de frente para trás, processando primeiro os *voxels* mais próximos do observador, ou de trás para frente.

Primeiramente define-se a cor e opacidade do *voxel* de acordo com a função de transferência.

Em seguida, o *voxel* é projetado na imagem e determina-se a sua região de contribuição na imagem. A extensão desta região é calculada através de um *kernel*, como por exemplo o *kernel* Gaussiano. Note porém que estas extensões podem ser pré-calculadas e no caso de projeções ortogonais, usadas para todos os *voxels*. Westover (1990) chama esta etapa de Splatting.

Por último é realizada a composição. A região de contribuição obtida anteriormente é composta com a imagem, sendo a única etapa dependente da ordem de travessia de *voxels*.

2.5.2 Análise

O objetivo principal de Westover (1989) ao criar esta técnica foi encontrar um algoritmo para renderização interativa, portanto que fosse paralelizável facilmente e fosse o máximo possível orientado a consulta em tabelas ao invés de realizar cálculos.

Splatting requer que apenas um *voxel* precise ser acessado a cada iteração, ou seja, em sistemas paralelos que não compartilham memória, não é necessário duplicar o volume em cada ponta. Um cuidado em implementações paralelas é que a etapa de composição deve ser feita na ordem correta.

Diferente do *ray-casting*, não é necessário realizar *supersampling* para evitar *aliasing* na imagem, pois a reconstrução no espaço de imagem é feita usando uma função contínua e precisa ser apenas amostrada (WESTOVER, 1989).

van Baar et al. (2001) apresenta uma técnica chamada EWA Volume Splatting, que utiliza uma combinação de *kernel* elíptico Gaussiano com filtro passa-baixo Gaussiano. Segundo o autor, esta técnica fornece imagens de melhor qualidade e facilita projeção em perspectiva eficiente.

Figura 2.12: Renderização realizada por *splatting* com diferentes funções de transferência (NEOPHYTOU; MUELLER, 2005).

A Figura 2.12 mostra resultados de renderização via *splatting*.

3 *Aceleração gráfica*

Möller, Haines e Hoffman (2008) explicam que pode-se falar em renderização de tempo real se existe interatividade e modelos tridimensionais. Embora o uso de dispositivos de aceleração gráfica não seja um requisito absoluto, o autor nota que eles têm se tornado necessários para a maior parte das aplicações de tempo-real.

As placas de vídeo modernas possuem um processador dedicado chamado de Graphics Processing Unit (GPU), que segundo Fernando (2004), tem um poder computacional maior do que processadores de propósito geral, ou Central Processing Units (CPU). O autor justifica essa afirmação com a comparação abaixo:

- Intel Pentium 4 de 3 GHz, dados teóricos: 6 GFLOPS e pico de 5.96 GB/s em largura de banda de memória.
- Nvidia GeForce FX 5900, dados observados: 20 GFLOPS e pico de 25.3 GB/s em largura de banda de memória.

A Figura 3.1 mostra um comparativo mais recente entre as capacidades de processamento das GPUs e CPUs.

A justificativa para isso é que a natureza especializada das GPUs permite que transistores adicionais sejam usados em unidades aritméticas ao invés de memória *cache*, além do mercado multibilionário de jogos pressionar a evolução destes processadores (LUEBKE et al., 2004).

Para demonstrar o ganho de performance que as aplicações que utilizam o *pipeline* com aceleração por *hardware* podem obter, a Figura 3.2 mostra resultados de um *benchmark* feito no jogo Unreal Tournament 2004. O gráfico mostra o tempo que cada imagem levou para ser renderizada ao longo da simulação, sem considerar o tempo de outros processamentos, como física. O *benchmark* foi realizado em uma máquina com processador Intel Core2 Duo de 2.26 GHz e placa de vídeo NVIDIA GeForce 9800M GS. O próprio jogo possui um mecanismo de *benchmark,* que faz a simulação de uma partida entre jogadores controlados por inteligência artificial. O modo de renderização via *software*, ou seja, sem utilizar dispositivos de aceleração

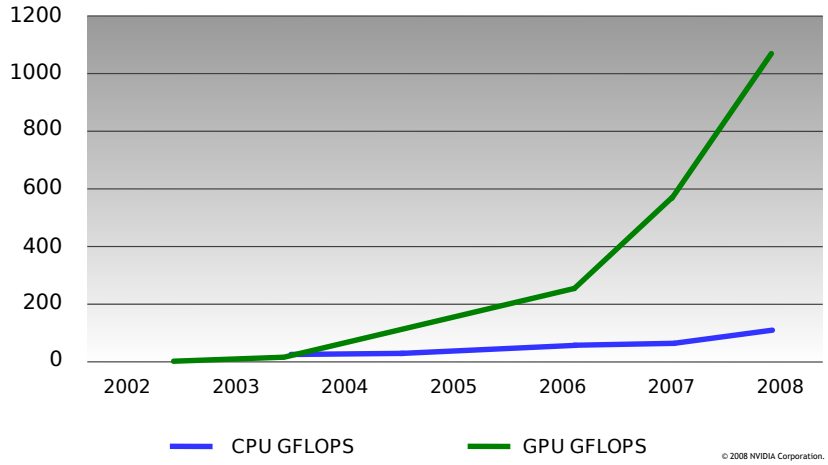


Figura 3.1: Comparação da capacidade de processamento entre GPUs e CPUs (TAMASI, 2008).

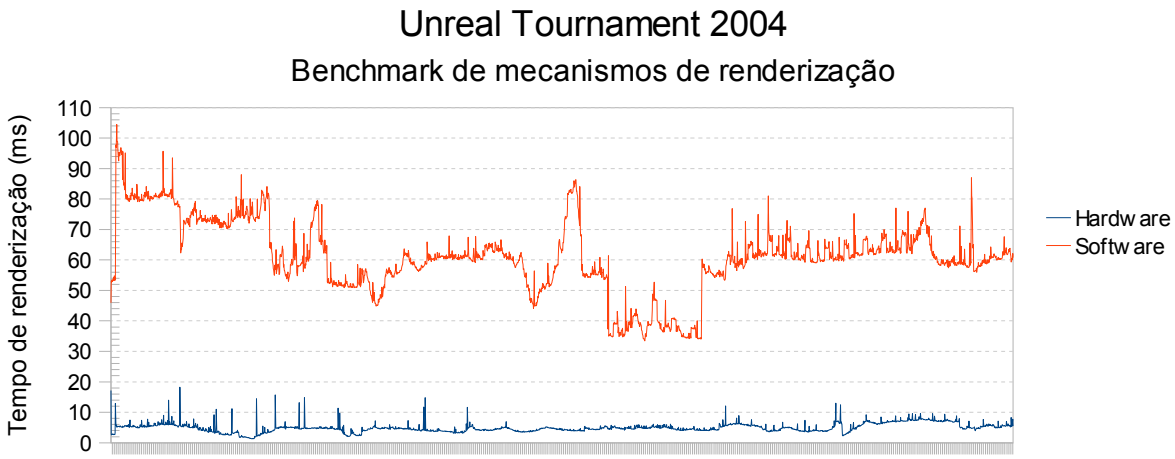


Figura 3.2: *Benchmark* de mecanismos de renderização do Unreal Tournament 2004.

gráfica, utiliza o renderizador Pixomatic desenvolvido pela RAD Game Tools. O modo de renderização via *hardware* utiliza o OpenGL. Para o *benchmark* foram habilitados todos os efeitos gráficos disponíveis, sem realização de *anti-aliasing* e com resolução de 1024x768.

As GPUs evoluíram para a forma de processadores poderosos e flexíveis, oferecendo grande largura de banda de memória e unidades de processamento de vértices e *pixels* totalmente programáveis (LUEBKE et al., 2004). O autor diz que não é surpreendente que esses processadores sejam capazes de realizar computação de propósito geral além das aplicações gráficas para o qual foram projetados. Esta técnica ficou conhecida como General-Purpose computation on Graphics Processing Units (GPGPU).

Neste capítulo será apresentado o *pipeline* gráfico utilizado em aplicações que realizam renderização em tempo-real, qual é a relação entre ele e as GPUs e quais são os recursos de

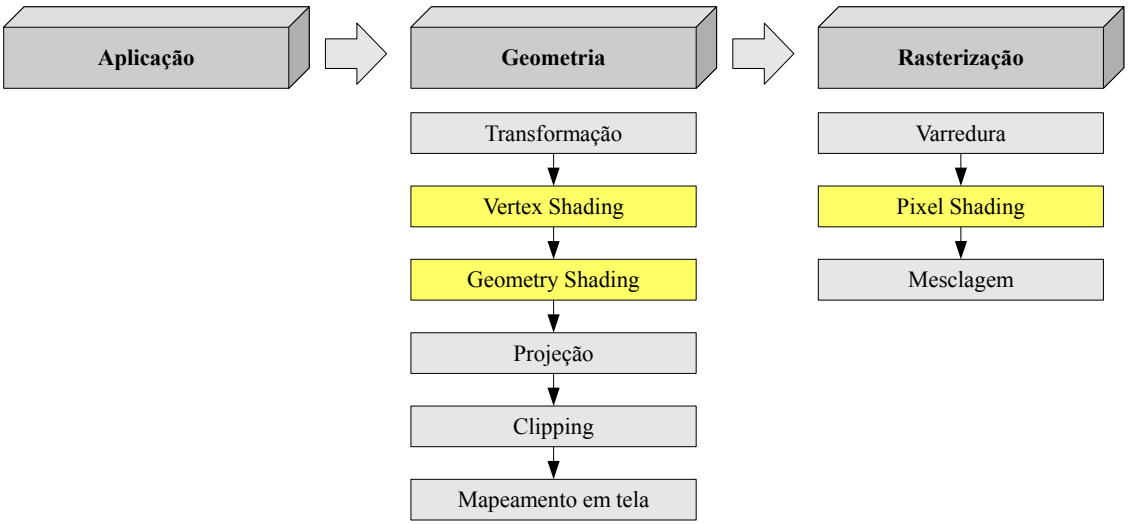


Figura 3.3: Visão geral do *pipeline* de renderização gráfica.

programação disponíveis para utilizar o *pipeline* acelerado. Por fim serão apresentadas técnicas de renderização volumétrica adaptadas para serem aceleradas por GPU.

3.1 Pipeline de renderização

O *pipeline* de renderização gráfica, ou apenas *pipeline*, é considerado o núcleo gráfico de aplicações gráficas de tempo real (MÖLLER; HAINES; HOFFMAN, 2008). A função principal é gerar, ou renderizar, uma imagem bidimensional a partir de uma descrição tridimensional de objetos, câmera, luzes, etc. O pipeline aqui apresentado é o que é acelerado pelas placas modernas de aceleração gráfica e trabalha apenas com o modelo de primitivas geométricas. A Figura 3.3 mostra a visão geral do *pipeline*, adaptado de Möller, Haines e Hoffman (2008).

O primeiro estágio do pipeline é a aplicação. Este estágio é controlado completamente pelo programador e sua única função, do ponto de vista da renderização, é alimentar o estágio seguinte com as primitivas geométricas a serem utilizadas na renderização.

O estágio de geometria é responsável pela maioria das operações sobre as primitivas geométricas. Ele é dividido em etapas, onde o resultado de cada uma é disponibilizado para as seguintes:

- Transformação: aplica as matrizes de transformação e conversão entre sistemas de coordenadas diferentes, como de coordenadas de objeto para coordenadas de mundo e então para coordenadas de câmera;

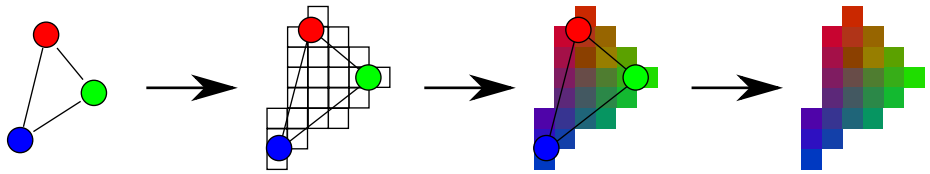


Figura 3.4: Conversão de uma primitiva geométrica em fragmentos.

- Vertex Shading: realiza os cálculos relativos aos vértices das primitivas, como iluminação;
- Geometry Shading: descarta e gera primitivas baseado nas primitivas originalmente recebidas da aplicação;
- Projeção: realiza os cálculos de projeção;
- Clipping: corta as primitivas que estejam apenas parcialmente dentro do espaço de visualização;
- Mapeamento em tela: transforma as coordenadas das primitivas visíveis em coordenadas de tela.

O estágio de rasterização recebe as primitivas geométricas em coordenadas de tela, tendo como objetivo transformar essa geometria em um conjunto de *pixels* que definirá a imagem final da renderização. Ele é divido nas seguintes etapas:

- Varredura: converte a primitiva em *pixels*. As propriedades destes *pixels* vêm da interpolação das propriedades dos vértices que formam a primitiva. Essas propriedades incluem informação de profundidade e informações de iluminação do estágio de geometria. Esta etapa está ilustrada na Figura 3.4;
- Pixel Shading: realiza os cálculos relativos a cada *pixel* para definir a cor, como por exemplo cálculos de iluminação e texturização;
- Mesclagem: mescla os *pixels* rasterizados com a imagem. O *pixel* pode ter sido gerado para um objeto que está posicionado atrás de outro previamente rasterizado, portanto esta etapa também deve verificar se o *pixel* vai ser realmente aproveitado ou descartado.

A maior parte deste processo é realizado nas GPUs, já que o propósito dela é justamente acelerar um *pipeline* como este. Os algoritmos de muitas destas etapas podem ser executados

em paralelo, já que elas são aplicadas em vários elementos e de forma independente. No Pixel Shading, por exemplo, os cálculos de todos os *pixels* podem ser feitos ao mesmo tempo se houver recursos de *hardware* suficientes.

Muitas dessas etapas, ou funcionalidades, são implementadas diretamente nas GPUs e não podem ser alteradas. Elas são chamadas de funcionalidade fixa. Outras funcionalidades podem ser alteradas em placas de aceleração mais modernas através de programação: Vertex Shading, Geometry Shading e Pixel Shading. Note que a etapa de Geometry Shading é interessante caso possa ser programada, mas não influencia o *pipeline* em sua funcionalidade padrão. O programa que substitui a funcionalidade dessas etapas é chamado de programa *shader* ou simplesmente *shader*.

Embora normalmente a renderização seja feita para uma imagem que será exibida em tela, existem técnicas de renderização de múltiplos passos onde são renderizadas imagens intermediárias para serem usadas nos passos seguintes até finalmente formar a imagem que será exibida. Um dos algoritmos que utiliza esta técnica é o conhecido como Shadow Mapping (WILLIAMS, 1978), que tem o propósito de gerar sombras. Este algoritmo consiste em primeiro renderizar a cena do ponto de vista da fonte de luz, para no segundo passo renderizar do ponto de vista do observador utilizando o resultado do primeiro passo.

3.2 Recursos de programação

Para ter acesso aos recursos computacionais da GPU, existem algumas linguagens específicas, além das interfaces de programação, ou Application Programming Interface (API). Nas subseções seguintes serão descritos os recursos escolhidos para este trabalho.

3.2.1 OpenGL

O OpenGL, ou Open Graphics Library, é uma interface de software para um dispositivo de hardware. Esta interface consiste de comandos usados para especificar objetos e operações necessárias para produzir aplicações tridimensionais interativas (OPENGL et al., 2005). Embora normalmente o OpenGL seja usado para ter acesso à aceleração gráfica, nada impede que as funcionalidades não sejam aceleradas em determinada implementação.

A especificação do OpenGL foi desenvolvida pela Silicon Graphics Inc. (SGI) em 1992 e hoje é um padrão mantido pelo Khronos Group, um consórcio tecnológico sem fins lucrativos. Implementações do OpenGL são muitas vezes fornecidas pelos fabricantes de dispositivos gráfi-

cos e para diversas plataformas. Plataformas onde OpenGL está disponível incluem o Microsoft Windows, Mac OS X e Linux. Variantes do OpenGL são encontrados no Nintendo GameCube, Wii, Nintendo DS, PlayStation 3, PlayStation Portable, iPhone, Android e Symbian OS.

Como alternativa ao OpenGL existe o Direct3D, uma API proprietária mantida e implementada pela Microsoft apenas para suas plataformas. Ambos possuem as mesmas capacidades de comunicação com o *hardware* gráfico, portanto foi escolhido o OpenGL para este trabalho devido à suas vantagens multiplataforma e as bibliotecas do Laboratório de Processamento de Imagem e Computação Gráfica (LAPIX) fornecerem maior suporte para esta API.

O OpenGL opera como uma máquina de estados, embora existam conceitos de objeto para manipulação de alguns elementos, como Vertex Buffer Objects (VBO) e Framebuffer Objects (FBO) (SEGAL; AKELEY, 2006). Os Framebuffer Objects fornecem suporte para armazenamento do resultado de um passo de renderização em uma textura, que poderá ser usada em um passo seguinte de renderização. Esta técnica é conhecida como *render-to-texture*.

A programação das etapas de Vertex, Geometry e Pixel Shading é feita através de uma linguagem própria, que possui sua própria especificação.

3.2.2 GLSL

GLSL, ou OpenGL Shading Language, é linguagem utilizada para as etapas programáveis do *pipeline* especificado pelo OpenGL. Ela é baseada em ANSI C e muitas características foram mantidas, exceto quando elas conflitam com problemas de performance ou facilidade de implementação. Possui suporte a tipos vetoriais e matriciais para maior concisão nas típicas operações gráficas. Esta linguagem foi projetada especificamente para ser usada em um ambiente OpenGL, estando algumas informações da máquina de estado OpenGL disponíveis para o programa GLSL (KESSENICH; BALDWIN; ROST, 2004).

Assim como esta é a linguagem de *shading* para o OpenGL, o Direct3D também possui uma, chamada High Level Shading Language (HLSL). Além disso existe a linguagem C for Graphics (Cg) desenvolvida pela Nvidia com o objetivo de facilitar a programação de *shaders* quando isso só era possível com linguagem *assembly*. Cg pode ser usada tanto com OpenGL ou Direct3D. Como já foi definido o uso de OpenGL, foi escolhido GLSL por também ser um padrão neutro entre fabricantes.

A Figura 3.5 mostra o fluxo de dados no *pipeline* do ponto de vista do programador de *shaders*. As informações de conectividade e vértices são alimentadas pela aplicação para o processo de renderização. Os vértices são processados no Vertex Shader e enviados para a etapa

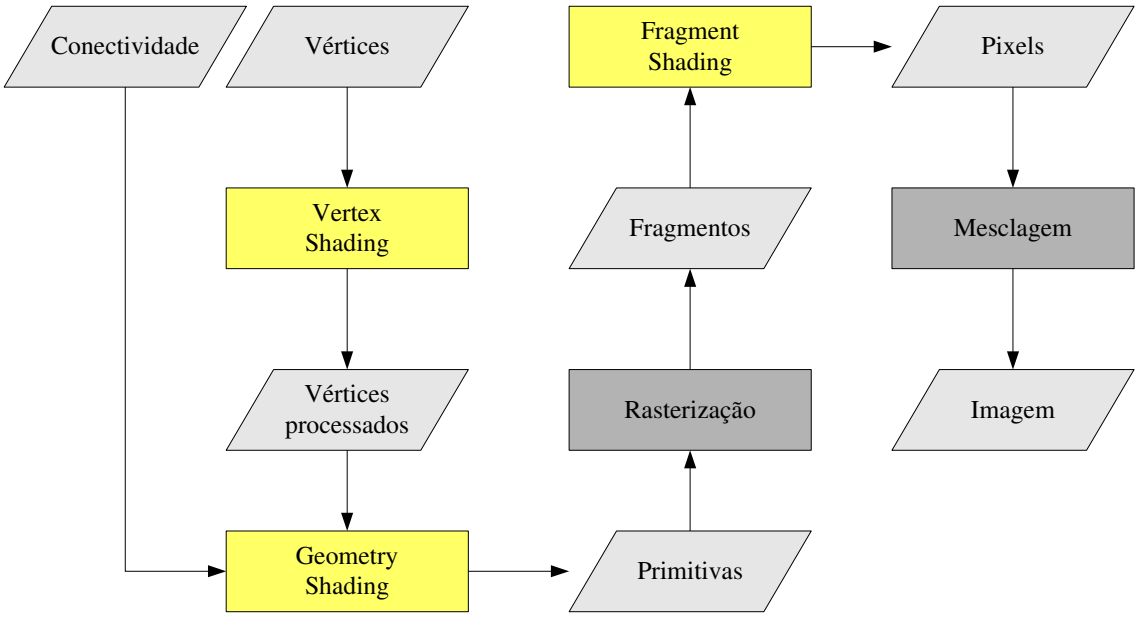


Figura 3.5: Visão geral do *pipeline* do ponto de vista do programador de *shaders*.

de Geometry Shading, que recebe também a informação de conectividade entre os vértices, ou seja, as arestas. Em seguida as primitivas são rasterizadas por uma funcionalidade fixa e os fragmentos gerados são processados no Fragment Shading, nome dado à etapa equivalente de Pixel Shading. Por fim, os *pixels* resultantes são mesclados na imagem através de uma funcionalidade fixa.

As funcionalidades programáveis podem passar dados para as funcionalidades seguintes de uma forma restrita. No código do *shader*, as variáveis que possuem estes dados são identificadas pela palavra-chave "varying". Variáveis desse tipo são tratadas de forma diferente dependendo da funcionalidade. Além disso, existem dados constantes que podem ser acessados em todas as etapas, como acesso às texturas, informações referentes ao estado da máquina OpenGL e outras constantes definidas pela aplicação. Estes dados são acessados através de variáveis com o qualificador "uniform". Nem todas as informações estarão disponíveis para todas as funcionalidades.

As entradas e saídas para a etapa de Vertex Shading estão ilustradas na Figura 3.6. O Vertex Shader tem acesso às matrizes de transformação e a responsabilidade de aplicá-las nos vértices. Note que o shader não tem conhecimento de outros vértices, a não ser o qual ele está processando. Por este motivo, esta etapa é paralelizável. As variáveis do tipo "varying" nesta funcionalidade podem ser lidas e escritas, estando associadas ao vértice sendo processado.

O Geometry Shader ainda não faz parte da especificação do GLSL (KESSENICH, 2009),

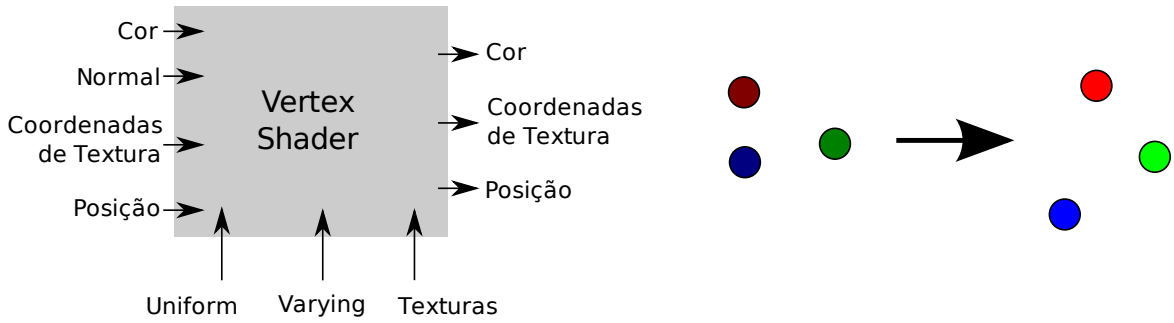


Figura 3.6: Entradas e saídas do Vertex Shader.

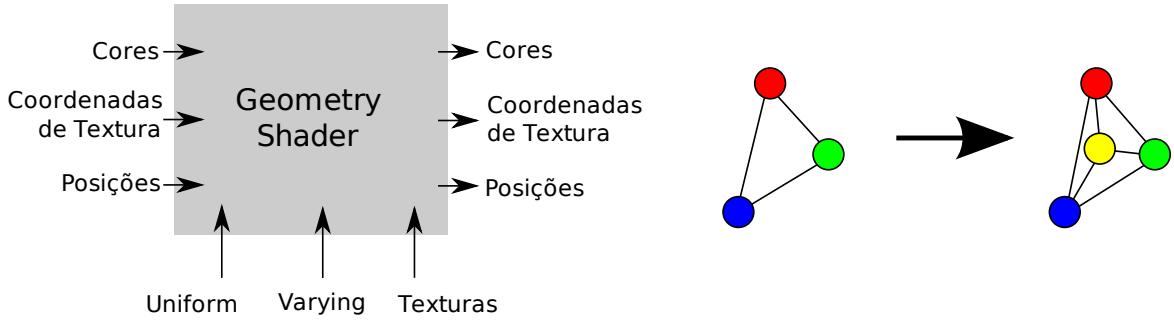


Figura 3.7: Entradas e saídas do Geometry Shader.

porém é possível programar esta funcionalidade através de uma extensão, presente pelo menos na implementação de OpenGL da Nvidia, chamada GL_EXT_geometry_shader4. A Figura 3.7 mostra as entradas e saídas desta funcionalidade. Ao contrário do Vertex Shader, o Geometry Shader tem acesso à todos os vértices de uma primitiva, a conexão entre os vértices e opcionalmente os vértices das primitivas adjacentes. Com estas informações é possível executar, por exemplo, algoritmos de subdivisão ou gerar primitivas proceduralmente. Os atributos e variáveis "varying" de cada vértice podem ser lidos e escritos.

A Figura 3.8 ilustra as entradas e saídas do Fragment Shader. Esta funcionalidade define a cor dos *pixels* que foram gerados pela rasterização. Opcionalmente, o *shader* pode alterar a profundidade do fragmento, como também descartá-lo completamente. As variáveis do tipo "varying" e outros atributos do vértice são interpolados para cada fragmento. Como esta é a última funcionalidade, estas variáveis são somente para leitura. É possível obter a posição do fragmento em coordenadas de tela, mas não é possível alterá-la.

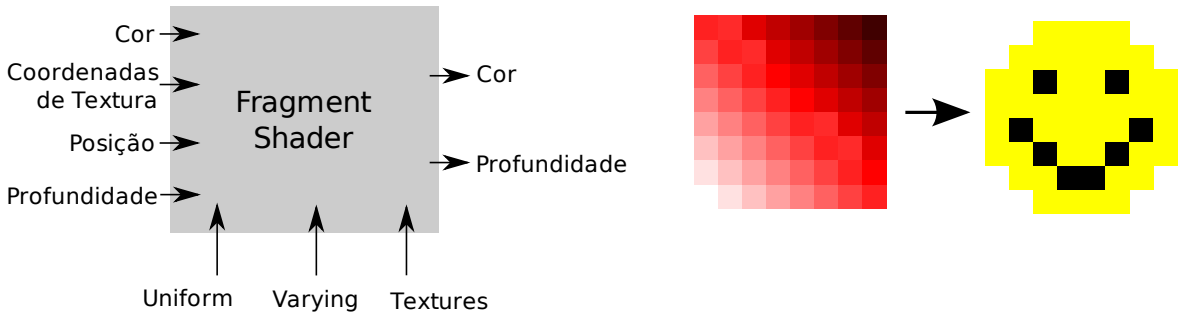


Figura 3.8: Entradas e saídas do Fragment Shader.

3.3 Renderização volumétrica acelerada

Como visto no Capítulo 2, a renderização de volumes possui características diferentes de tratamento dos dados comparado ao *pipeline* apresentado neste capítulo. Ainda assim, é possível aproveitar a aceleração que as placas gráficas fornecem, através de técnicas que foram adaptadas ou criadas para rodar nestes dispositivos. Uma das características utilizadas nestes métodos é o fato da GPU acessar rapidamente texturas tridimensionais com interpolação trilinear.

3.3.1 Marching Cubes

Após extraída a superfície, ela já pode ser renderizada com aceleração gráfica. O propósito de acelerar o Marching Cubes é poder escolher interativamente o valor de referência para a superfície.

Dyken et al. (2007) apresenta uma implementação do Marching Cubes utilizando uma estrutura de dados hierárquica chamada HistoPyramid. A idéia principal é criar o número de triângulos necessários para a gerar a superfície, identificar o cubo ao qual o triângulo está associado e então posicionar os vértices do triângulo de acordo com a configuração do cubo pela tabela do Marching Cubes. A HistoPyramid é utilizada para acelerar a identificação dos triângulos ao cubo correspondente.

3.3.2 Renderização baseada em textura

Wilson, Vangelder e Wilhelms (1994) apresentam um método para aproveitar os recursos de aceleração no mapeamento de texturas realizado pelas placas gráficas. Este método consiste em criar uma textura tridimensional tendo como conteúdo o resultado da aplicação da função

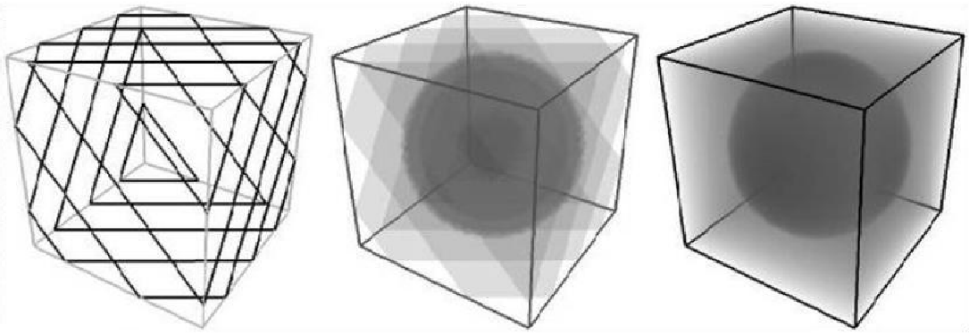


Figura 3.9: Renderização volumétrica baseada em texturas (HUI-ZHANG; JAE-CHOI, 2006).

de transferência nos *voxels* originais. Essa textura é então projetada em polígonos alinhados, ou fatias, como mostra a Figura 3.9. Embora este método seja acelerado pela GPU, ele não requer alteração nas funcionalidades do *pipeline* e portanto funciona mesmo em placas aceleradoras mais antigas, desde que suportem texturas tridimensionais.

Hui-Zhang e Jae-Choi (2006) explica que para obter uma maior qualidade com este método, são necessários muitas fatias e isso torna a renderização mais lenta. O autor propõe um método que altera a funcionalidade padrão do pipeline para conseguir um melhor resultado com menos fatias, porém a técnica é específica para renderização de superfícies.

3.3.3 Ray-casting

Kruger e Westermann (2003) acelera o *ray-casting* em GPU amostrando os raios através da rasterização e mesclagem de retângulos que fatiam o volume. A configuração inicial do *ray-caster*, que diz onde os raios começam e terminam, é obtida através da rasterização de uma caixa alinhada ao volume e o resultado é gravado em textura. Em cada passo seguinte os raios avançam, através da projeção do volume nos retângulos, e é gerada uma imagem correspondente a este progresso do raio. Esta imagem é então mesclada com as imagens geradas anteriormente. O autor apresenta esta técnica juntamente com a otimização de término antecipado do raio.

A vantagem deste método sobre a renderização baseada em textura é que na baseada em textura os cálculos de iluminação, mesclagem e acesso à dados de textura são feitos para todos os fragmentos, mesmo que eles não façam parte da imagem. Já este método altera os *shaders* de forma a descartar os cálculos para fragmentos que não contribuirão com a imagem final através do teste de profundidade.

Scharsach (2005) adapta esta técnica de forma que as amostragens de cada raio sejam feitas em apenas um passo de renderização, através de estruturas de repetição que são possíveis em GPUs mais modernas. O autor descreve também otimização de salto de espaços vazios,

refinamentos na imagem para o caso específico de renderização de superfícies e interação com geometria.

Hadwiger et al. (2008) detalha aspectos mais avançados desta técnica como níveis de detalhes com multirresolução, compactação dos dados e iluminação considerando sombras e espalhamento dos raios.

3.3.4 Splatting

Chen et al. (2004) apresenta uma técnica para acelerar o EWA Volume Splatting através da criação de quadriláteros para cada *voxel*, onde serão projetados os *splats*. Vega-Higuera et al. (2005) implementa splatting em GPU criando pontos ao invés de quadrados e armazenando-os na memória de vídeo. A criação e ordenação dos pontos é feita em uma etapa de pré-processamento na CPU, porém as etapas de renderização são realizadas totalmente em GPU.

4 Implementação

Dentre as técnicas apresentadas, apenas o Marching Cubes em CPU está implementado nas bibliotecas de visualização volumétrica do LAPIX. Por este motivo a preferência foi dada à técnicas de renderização direta, de forma a complementar essas bibliotecas. Foi escolhida então a técnica de *ray-casting* para ser implementada tanto em CPU como GPU a fim de realizar comparações. A preferência do *ray-casting* sobre o Splatting se deu pela simplicidade aparente do algoritmo. A renderização por mapeamento de texturas foi descartada pela similaridade com o *ray-casting*.

4.1 Ray-casting em CPU

Esta implementação executa todo o processo de renderização em CPU e envia a imagem resultante para a placa de vídeo na forma de textura. Desta forma, esta textura pode ser aplicada a uma geometria, como um retângulo, e ser exibida pelo OpenGL. O Algoritmo 4.1 mostra os passos do renderizador de forma geral, que também podem ser visualizados na Figura 4.1.

A implementação foi feita em etapas. Na primeira etapa, o renderizador disparava os raios para cada um dos *pixels* da imagem e colhia amostras do volume em intervalos fixos até uma distância pré-determinada. Esta abordagem é ineficiente pois tenta amostrar valores que não fazem parte do volume, mas serviu para desenvolver a base do código do renderizador, como configuração do raio. A interpolação usada na amostragem foi a de vizinho mais próximo e os valores de cor eram em escala de cinza.

Na segunda etapa, foi implementada a intersecção dos raios com o volume de forma a determinar onde cada raio começa e termina. Com isso apenas amostras que pertençam ao volume são coletadas. Foi implementada também a otimização de término antecipado do raio, reduzindo mais ainda o número de amostragens.

Por fim, a implementação foi concluída realizando a amostragem do volume com interpolação trilinear e incluindo suporte aos 3 canais de cores para as funções de transferência.

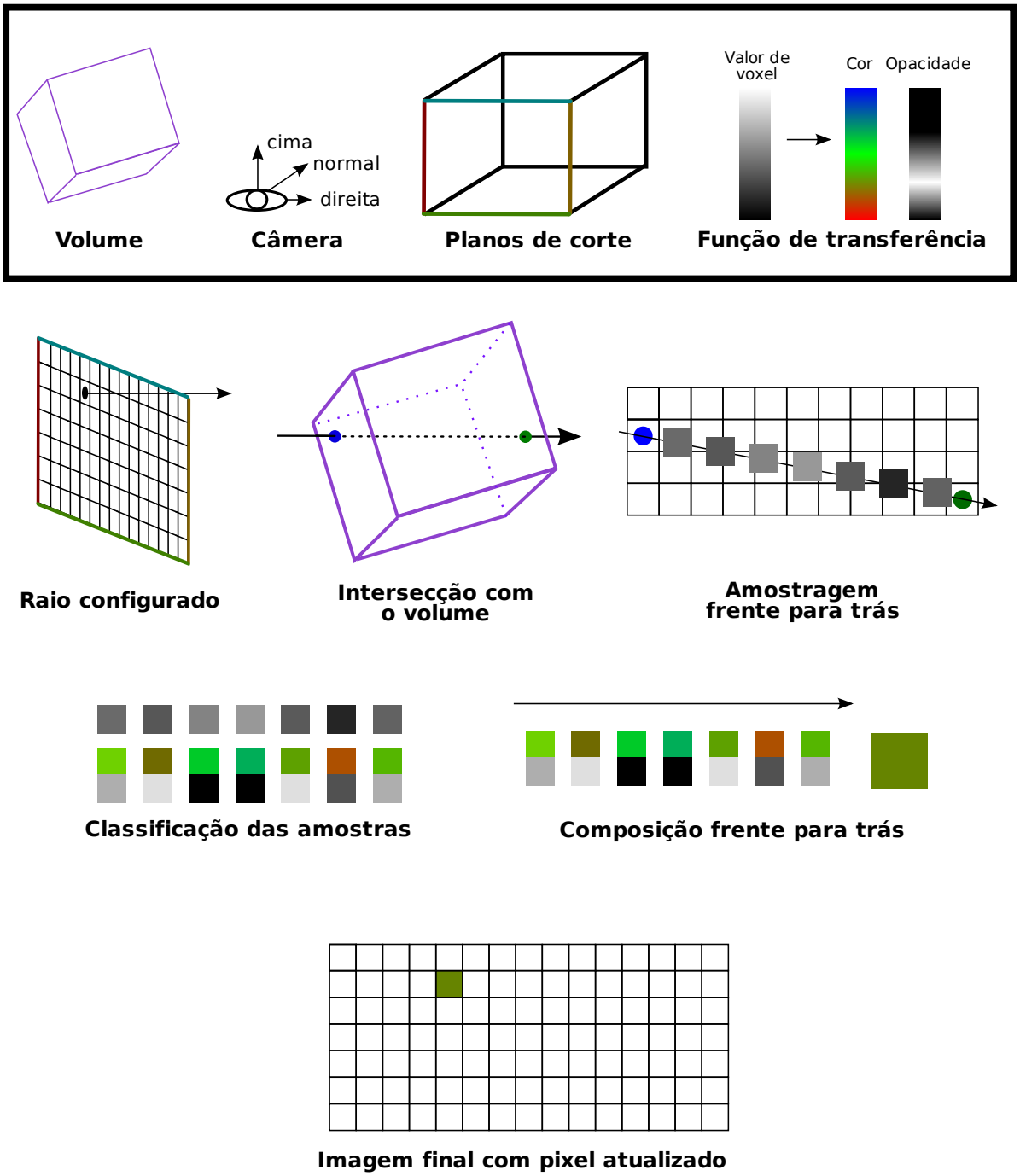


Figura 4.1: Diagrama ilustrando os passos do *ray-caster*.

Algoritmo 4.1 *Ray-casting* de volume.

```
1:  programa renderizadorRaycaster
2:  global câmera: Informações da câmera.
3:  global planos: Distância dos planos de corte ao centro da câmera.
4:  global volume: Volume a ser renderizado.
5:  resultado imagem: Imagem resultante da renderização.
6:  início
7:      para cada pixel em imagem faça
8:          raio ← configurarRaio(pixel)
9:          cor ← dispararRaio(raio)
10:         imagem_pixel ← cor
11:     fim
12: fim
```

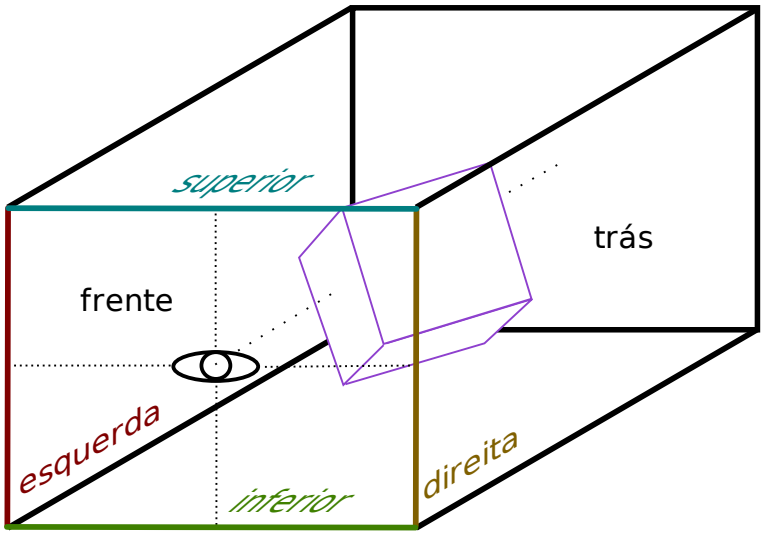


Figura 4.2: Planos de corte da projeção paralela.

4.1.1 Configuração do raio

A projeção utilizada foi a paralela, por ser mais simples. Os raios partem do plano da frente, chamado de *near plane*, e seguem em direção ao plano de trás. A Figura 4.2 ilustra os planos de corte em relação à câmera formando um paralelepípedo, característica da projeção paralela, que representa a região que aparecerá na imagem renderizada. A câmera aponta para o plano de trás, sendo essa direção o vetor normal da câmera. Por convenção deste trabalho, considera-se que a posição dos planos da frente, esquerda e inferior sejam negativos, ou seja, o centro da câmera está na verdade no centro do paralelepípedo.

O Algoritmo 4.2 mostra como calcular a posição e direção do raio para determinada posição de *pixel*. Note que as variáveis "horizontal" e "vertical" não variam de acordo com a posição do *pixel*, então são calculadas apenas uma vez.

Algoritmo 4.2 Configuração do disparo do raio para determinado *pixel* da imagem.

1: **função** configurarRaio
2: **parâmetro** pixel**:** Posição do pixel na imagem. Valores no intervalo $[0, 1]$.
3: **resultado** raio**:** Origem e direção do raio a ser disparado.
4: **início**
5: $v_{cima} \leftarrow$ câmera.vetorCima
6: $v_{normal} \leftarrow$ câmera.vetorNormal
7: $v_{esquerda} \leftarrow v_{normal} \times v_{cima}$
8: origem $\leftarrow$ câmera.centro $+ v_{esquerda} *$ planos.esquerda $+ v_{cima} *$ planos.inferior $+ v_{normal} *$ planos.frente
9: horizontal $\leftarrow v_{esquerda} *$ (planos.direita $-$ planos.esquerda)
10: vertical $\leftarrow v_{cima} *$ (planos.superior $-$ planos.inferior)
11: raio.origem $\leftarrow$ origem $+$ horizontal $*$ pixel.x $+$ vertical $*$ pixel.y
12: raio.direção $\leftarrow v_{normal}$
13: **fim**

4.1.2 Disparo do raio

Após determinada a origem e a direção do raio, é feita a detecção de intersecção deste raio com o paralelepípedo que envolve o volume. O ponto de colisão mais próximo indica o ponto no volume que o raio atinge primeiro, enquanto o mais distante indica o último ponto na direção do raio que pertence ao volume. A amostragem é feita então entre esses dois pontos, que estão explicitados na Figura 2.10(a).

Uma característica dos volumes originados de imagens médicas é que a distância entre os *voxels* pode ser diferente em cada dimensão, ou seja, as proporções reais do objeto sendo representado podem estar esticadas. Esta situação está ilustrada bidimensionalmente na Figura 4.3. O espaço de imagem considera apenas o tamanho do volume armazenado, enquanto o espaço de mundo considera as proporções reais do objeto. Já o espaço normalizado ignora todas as proporções.

Considere que os raios amostrem o volume no espaço de imagem. Um raio horizontal que passe pelo centro intersectará a mesma quantidade de região escura do que um raio vertical que também passe pelo centro. Porém no objeto representado pelo volume, as regiões escuras são retangulares como mostra a ilustração no espaço de mundo. Portanto o raio vertical deveria intersectar uma maior quantidade de região escura. Por este motivo a amostragem do raio deve ser feita no espaço de mundo, sendo necessário realizar esta conversão de coordenadas ao acessar os dados do volume.

A distância entre cada ponto de amostragem ao longo do raio precisa ser pequena o suficiente para poder capturar o máximo de informações do volume e a renderização ter melhor

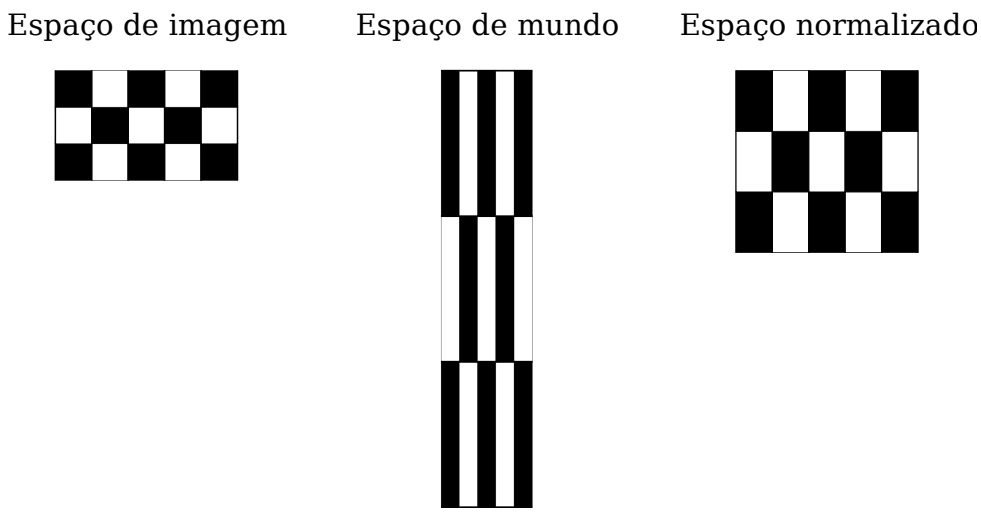


Figura 4.3: Possíveis espaços de amostragem no volume.

qualidade. Por outro lado, quanto menor o espaçamento, mais amostras serão coletadas e mais lenta se torna a renderização. Engel, Kraus e Ertl (2001) notam que de acordo com o teorema da amostragem, a frequência de coleta de amostras, que é o inverso do espaçamento, deve ser maior que a frequência de Nyquist para gerar resultados corretos. Porém as funções de transferência podem fazer com que esta frequência precise ser ainda maior. Para simplificação, neste trabalho a distância de amostragem foi definida como sendo 0.8 multiplicado pelo valor do menor espaçamento de *voxel* dentre os espaçamentos em cada dimensão, pois foi o suficiente pra obter uma qualidade razoável nos experimentos realizados.

As amostras são então coletadas e classificadas de acordo com a função de transferência. As amostras são obtidas através de interpolação trilinear entre os 8 *voxels* vizinhos mais próximos do ponto de amostragem. Aplica-se então a função de transferência na amostra e compõe-se o resultado. Feitas todas as composições, têm-se o valor de cor a ser colocado no *pixel* da imagem de onde foi disparado o raio.

A composição foi implementada de frente para trás para poder aplicar a otimização de término antecipado do raio. Se o valor acumulado de opacidade não for menor do que 1, significa que o *pixel* já está opaco e não é necessário continuar amostrando.

4.1.3 Resultados

De forma a poder comparar melhor os resultados entre as implementações de CPU e GPU, escolheu-se um volume e foram definidas algumas funções de transferência e posicionamentos de câmera. O volume utilizado foi construído a partir do empilhamento de 460 imagens carrega-

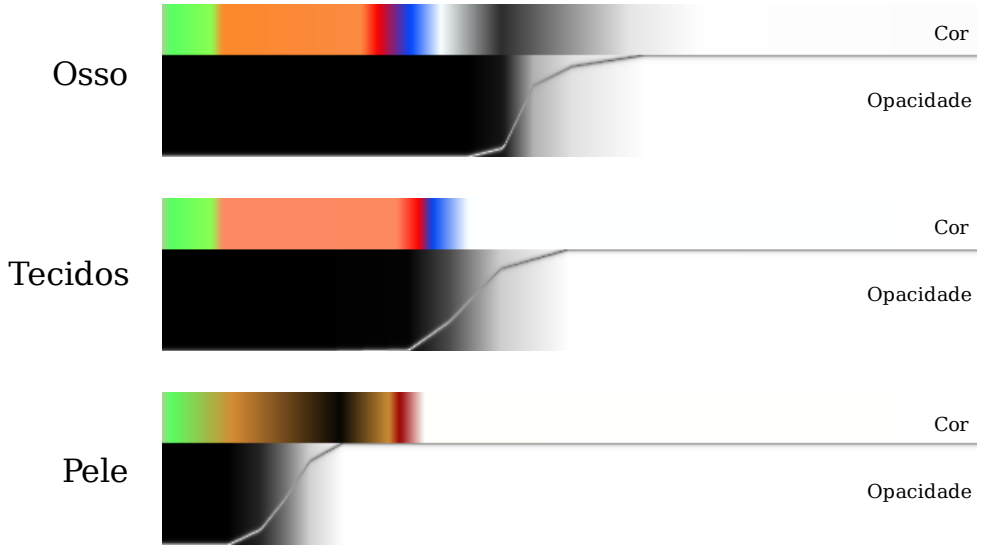


Figura 4.4: Funções de transferência usadas para gerar os resultados.

das do formato DICOM[1] de tamanho 512x512 em escala de cinza com 12 bits de profundidade.

A Figura 4.4 representa as funções de transferência para cor e opacidade na forma de degradê. Note que as funções para a cor são parecidas, sendo o controle da opacidade o maior diferencial. Além disso, foi necessário incluir uma tonalidade escura nas funções de osso e pele, pois caso contrário as imagens formadas pareceriam apenas silhuetas já que não foi implementado nenhum esquema de iluminação e portanto os resultados não proporcionam muita noção de profundidade. Os resultados das renderizações do volume as configurações especificadas podem ser observados na Figura 4.5.

Para medir a performance, foram usadas mais duas funções de transferência: opaca e transparente, tendo respectivamente opacidade máxima e a outra opacidade mínima para todos os voxels. Com a opaca, cada raio realiza apenas 1 amostra, graças à otimização de término antecipado do raio. Já a transparente não deixa de coletar nenhuma amostra. Sem a otimização de término antecipado do raio a performance para todas as funções de transferência seria a mesma da transparente, considerando a mesma configuração de câmera.

O gráfico da Figura 4.6 exibe os tempos de renderização para gerar as imagens nos tamanhos indicados. O *benchmark* foi realizado em uma máquina com processador Intel Core2 Duo Mobile P8400 de 2.26 GHz, 512 MB de RAM, placa gráfica NVIDIA GeForce 9800M GS, rodando o sistema operacional Linux 2.6.29. Foram realizadas 2 renderizações para cada configuração e calculada a média. Note que o tamanho da imagem resultante influencia diretamente no tempo de renderização, quadruplicando o tempo em segundos ao quadruplicar o número de

[1] O Digital Imaging and Communications in Medicine (DICOM) é um padrão para manipulação, armazenamento, impressão e transmissão de informações de imagens médicas.

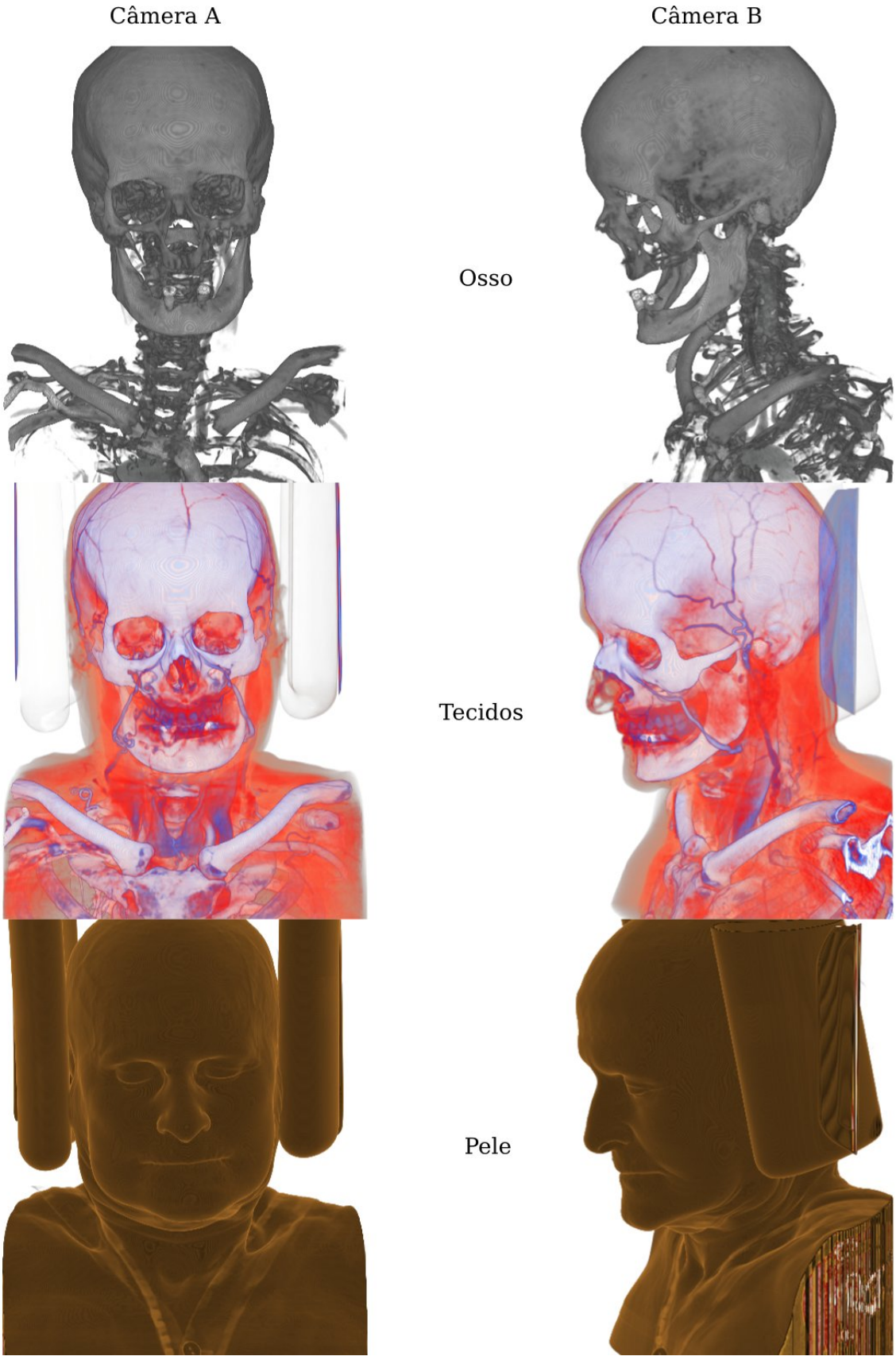


Figura 4.5: Resultados do *ray-caster* em CPU.

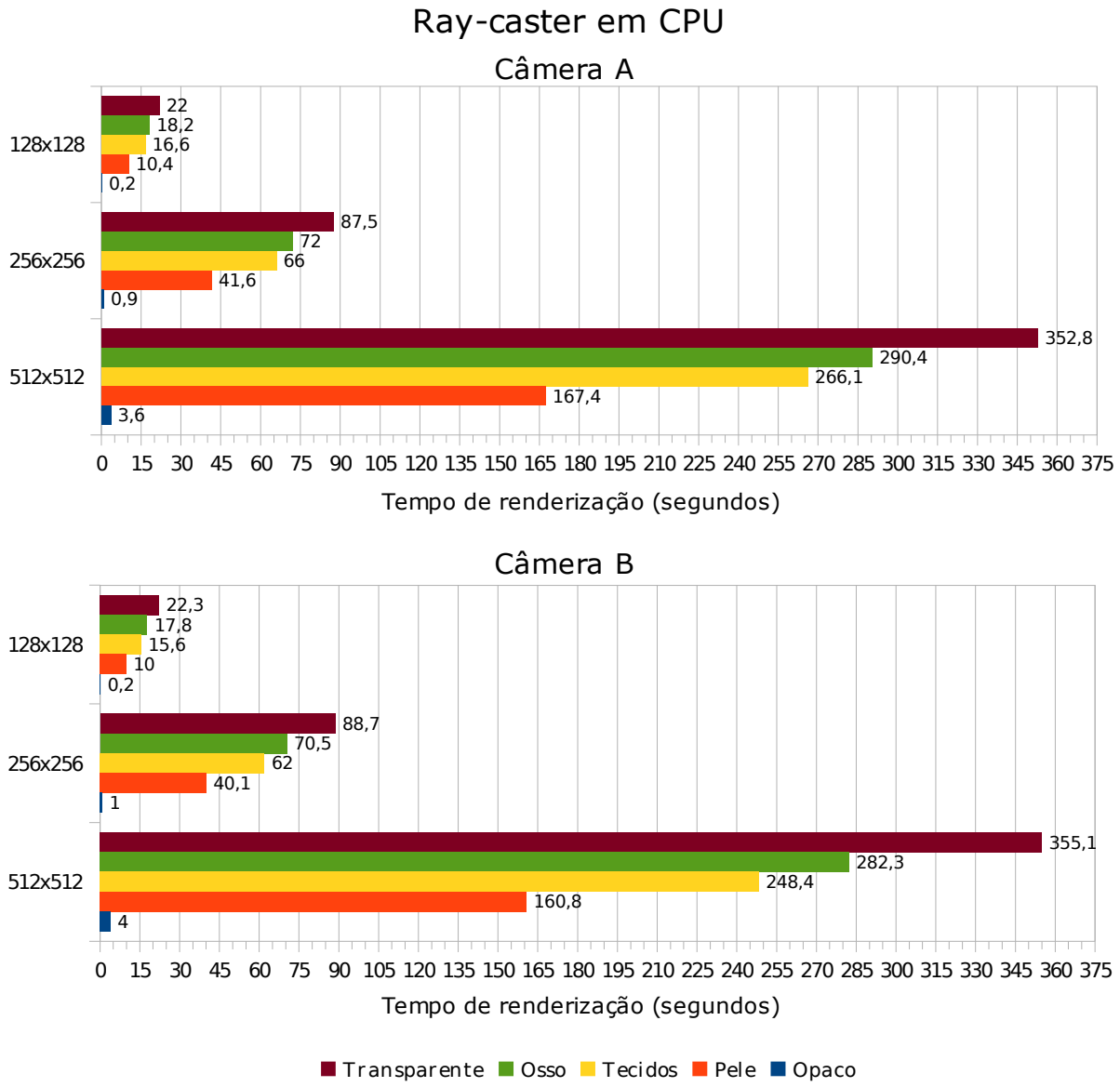


Figura 4.6: Performance do *ray-caster* em CPU.

pixels.

Mesmo nas resoluções mais baixas, é difícil afirmar que esta implementação permite uma visualização interativa do volume. É possível diminuir a resolução mais ainda para mostrar uma pré-visualização enquanto o usuário altera a função de transferência e configuração da câmera, mas ainda assim as imagem com maior resolução demoram mais de 2 minutos. Note ainda que nesta implementação não existem quaisquer cálculos de iluminação.

Para acelerar a renderização, poderiam ser implementadas otimizações no algoritmo como a de salto de espaços vazios ou ainda paralelizá-lo para aproveitar ambos os núcleos do processador utilizado. Existe ainda a possibilidade de realizar um pré-processamento, aplicando

a função de transferência em todo o volume e armazenar o resultado, o que poderia reduzir o tempo de renderização considerando que o usuário altera as configurações de câmera mais frequentemente do que altera a função de transferência. Porém optou-se por partir diretamente para a implementação em GPU com a expectativa de uma melhor relação entre ganho de performance e esforço de implementação.

4.2 Ray-casting em GPU

A interface da API do *ray-caster* em GPU é a mesma da implementação em CPU, sendo o resultado também disponibilizado na forma de textura. Como a imagem é gerada em GPU, utiliza-se a técnica de *render-to-texture* para escrever o resultado diretamente na textura, sem precisar transferir dados da CPU para GPU. O código que é executado em CPU apenas configura os estados da máquina OpenGL e define os parâmetros do *ray-caster*. O volume é armazenado na memória da placa gráfica como uma textura tridimensional, para então ser acessado pelo programa que rodará no processador gráfico.

Assim como a implementação em CPU, há duas principais etapas na renderização: configuração e disparo dos raios. A etapa de configuração dos raios utiliza a funcionalidade padrão do rasterizador para gerar duas imagens intermediárias, enquanto para o disparo dos raios altera-se o Fragment Shader para gerar a imagem final.

4.2.1 Armazenamento dos dados

Como o algoritmo agora roda no processador gráfico, a memória que este processador tem acesso diretamente é a memória da placa gráfica. No pipeline gráfico esta memória é usada para armazenar texturas e outros tipos de *buffer*. Portanto ao rodar um programa escrito em GLSL na GPU, o acesso à memória de vídeo se dá através de uma interface de acesso à textura. Isso significa que tanto o volume quanto as imagens intermediárias que serão usadas no algoritmo são armazenados na forma de textura.

Os formatos das texturas utilizadas pelo algoritmo estão indicados na Tabela 4.1. O formato L contém apenas 1 canal, chamado luminância. O formato RGBA contém os 3 canais de cores vermelho, verde e azul e mais 1 canal alfa, normalmente utilizado para representar opacidade.

Todas as texturas utilizam ponto-flutuante para representar os valores, porém como mostra a tabela, o volume utiliza uma precisão menor. O motivo disso é que alguns volumes utilizados no experimento ocupavam muita memória com o formato de 32 bits e a renderização tornava-se

Textura	Tipo	Formato	Precisão
Volume	3D	L	16 bits
Configurações dos raios	2D	RGBA	32 bits
Função de transferência	1D	RGBA	32 bits
Resultado	2D	RGBA	32 bits

Tabela 4.1: Formatos das texturas usadas no *ray-caster*.

lenta, possivelmente porque o driver de vídeo realizava *swapping* da textura entre a memória de vídeo e de sistema. A precisão do ponto-flutuante de 16 bits possui 10 bits de mantissa, de acordo com a especificação da extensão ARB_texture_float(BROWN et al., 2008) do OpenGL. Porém os volumes utilizados neste trabalho possuem pelo menos 12 bits de precisão, ou seja, a renderização em GPU vai ser menos fiel aos dados originais do volume. Em experimentos com outros volumes menores, a diferença de precisão não afetou a performance, sendo a justificativa de que no programa *shader* os valores do volume são acessados como ponto-flutuante de 32 bits.

O tamanho da textura da função de transferência foi definido como 4096, pois experimentalmente foi o suficiente para cobrir todos os valores de voxel. Note que ainda assim, a função de transferência é acessada com interpolação linear.

4.2.2 Configuração dos raios

A configuração dos raios foi implementada como descrita por Kruger e Westermann (2003). A idéia é obter duas imagens, uma indicando em que posição do volume o raio disparado deve começar a amostrar, e a outra indicando onde ele deve parar de amostrar. Desta forma é possível obter a origem e direção do raio. Por ser uma imagem, esta posição tridimensional é codificada como cor, ou seja, as componentes de vermelho, verde e azul correspondem às coordenadas x, y e z. O canal alfa é usado para indicar se um raio deve ser disparado ou não.

Essas imagens podem ser facilmente geradas através da rasterização de uma caixa nas mesmas dimensões do volume, onde a cor de cada vértice codifica a posição referente àquele vértice dentro do volume. A Figura 4.7 mostra como seriam essas imagens. A imagem da esquerda é gerada rasterizando apenas as faces da frente, enquanto na imagem da direita apenas as faces de trás. Note que as cores são representadas no intervalo entre 0 e 1, ou seja, em um eixo onde o vermelho, ou x, vale 1 para determinado *pixel*, significa que aquele ponto representa uma posição do mesmo valor da largura do volume. O tamanho destas imagens é o mesmo tamanho da imagem final do *ray-caster*, pois existe uma correspondência direta entre os *pixels*.

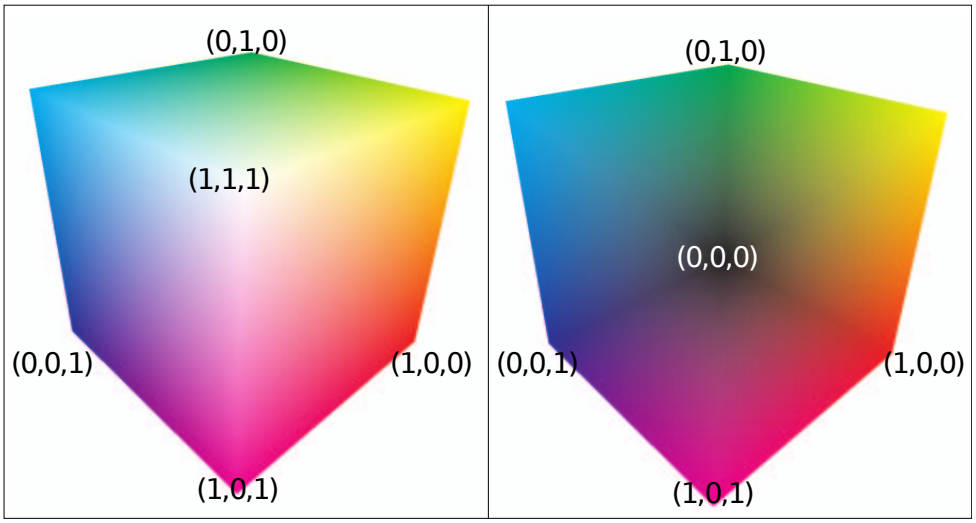


Figura 4.7: Configuração do raio via rasterização (KRUGER; WESTERMANN, 2003).

Nesta etapa são feitos então dois passos de rasterização sem substituir a funcionalidade padrão do pipeline gráfico. Além disso, o tráfego de informações entre a GPU e CPU é mínimo, pois o resultado é armazenado diretamente na memória gráfica e é acessado somente pela GPU na etapa seguinte.

4.2.3 Disparo dos raios

O disparo dos raios é feito em uma etapa de rasterização, substituindo a funcionalidade do Fragment Shader. Rasteriza-se um quadrado de forma a preencher todo o espaço de renderização, gerando exatamente o número de *pixels* desejado para a imagem resultante. Desta forma, o *shader* irá processar cada *pixel* individualmente, consultando os resultados obtidos pela etapa de configuração dos raios.

O programa que será executado em GPU constitui basicamente de um laço que amostra o volume ao longo do raio, tal como descrito por Scharsach (2005). O programa inicialmente consulta as imagens geradas na etapa de configuração do raio para determinar a origem e direção. A amostragem do volume é feita usando interpolação trilinear, que o próprio *hardware* gráfico provê de forma acelerada ao acessar texturas tridimensionais. Como foi explicado na implementação em CPU a amostragem deve ser feita em espaço de mundo, porém como o *shader* acessa a textura no espaço normalizado, também é necessário fazer essa conversão de coordenadas.

O resultado do programa é a cor do *pixel* na imagem final, que será armazenada em textura. Todo o processo desta etapa executa no processador gráfico e a imagem resultante é armazenada

	Câmera A				Câmera B			
	128	256	512	1024	128	256	512	1024
Opaco	519,5	251,5	124,3	55,4	469,3	235,2	112,6	53,5
Transparente	34,7	15,3	7,8	2,6	21,5	9,4	5,45	2,3

Tabela 4.2: Performance medida em fps para função de transferência opaca e transparente na implementação em GPU.

na memória de vídeo.

4.2.4 Resultados

Na Figura 4.8 é possível visualizar os resultados de renderização do mesmo volume e com as mesmas configurações que foram utilizadas para os resultados da implementação em CPU.

Os testes de desempenho foram feitos no mesmo hardware utilizado nos testes da implementação em CPU. Os resultados podem ser visualizados na Figura 4.9. A função de transferência opaca foi omitida do gráfico por legibilidade, mas seus resultados encontram-se na Tabela 4.2. A unidade utilizada é *frames* por segundo (fps), ou seja, quantas renderizações foram feitas em 1 segundo. Note que para as resoluções até 512x512, ao quadruplicar o número de *pixels* o tempo de renderização apenas dobra. Porém esta tendência não parece se manter para resoluções maiores, como ao comparar os resultados de 512x512 e 1024x1024.

Como foi descrito anteriormente a precisão utilizada para armazenar o volume foi a de ponto-flutuante de 16 bits, por questões de performance. A quantidade de *voxels* do volume em questão que tiveram seu valor alterado pela perda de precisão foi de 0,012% para o volume em questão. Utilizando precisão de 32 bits, as renderizações de tamanho 512x512 levaram cerca de 1 segundo. Ao comparar os resultados com ambas as precisões não se notam diferenças significantes. Nota-se diferença, porém, ao comparar com os resultados da implementação em CPU, que apresenta imagens mais esticadas verticalmente. A suspeita é que a configuração dos raios, por terem abordagens bem diferentes em cada implementação, não estejam gerando resultados precisamente iguais, fazendo com que as amostras sejam capturadas em posições diferentes.

Os resultados foram satisfatórios, considerando que a implementação em GPU renderizou a maioria das configurações cerca de 3000 vezes mais rápido do que a implementação em CPU. O motivo para um ganho tão grande se deve não apenas ao paralelismo que o processador gráfico proporciona, mas também o fato de uma das operações mais frequentes do algoritmo, que é o acesso à textura com interpolação, ser muito comum em aplicações gráficas convencionais. Por isso, a organização de memórias *cache* na GPU consegue otimizar o acesso sequencial em

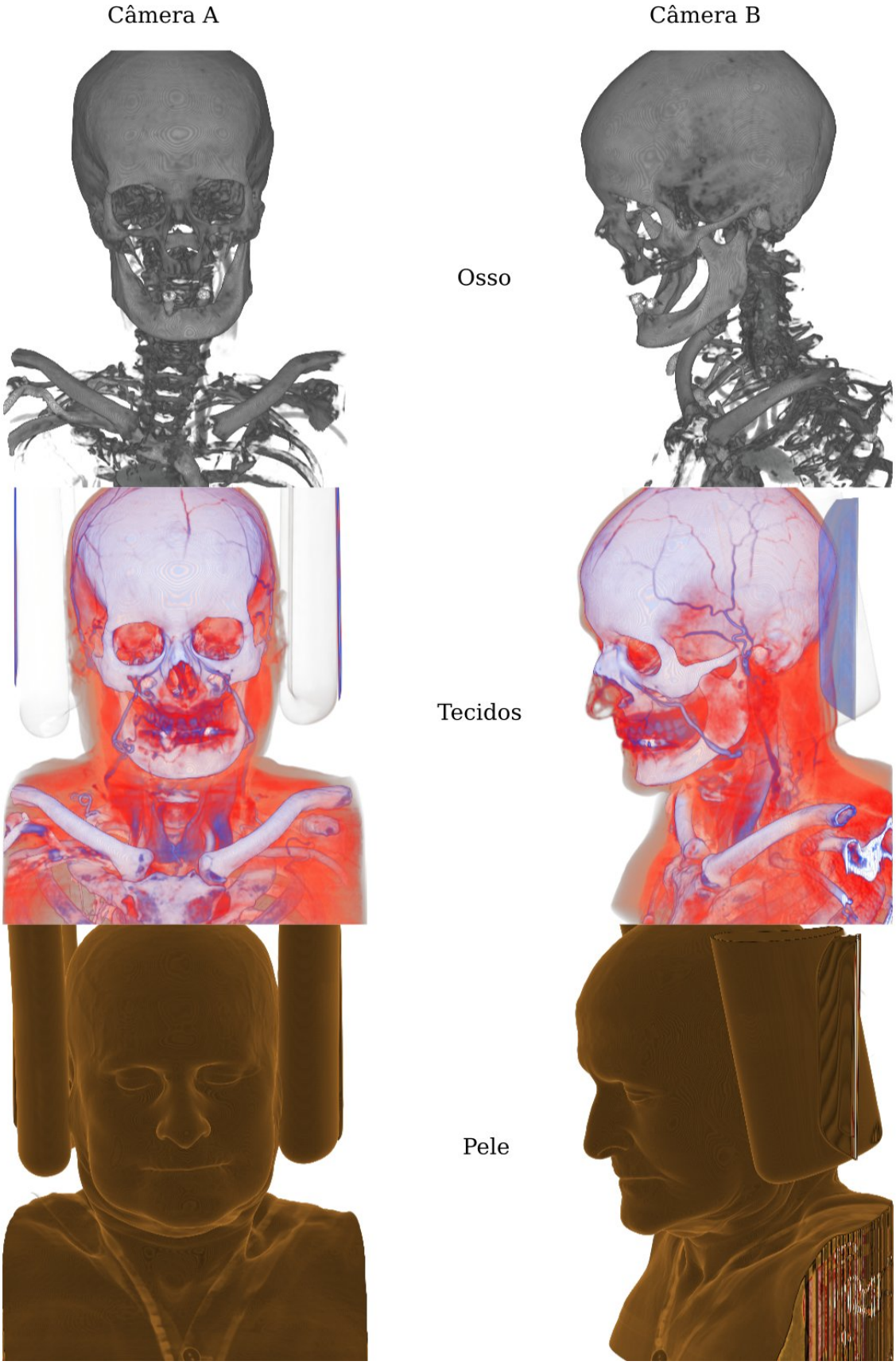


Figura 4.8: Resultados do *ray-caster* em GPU.

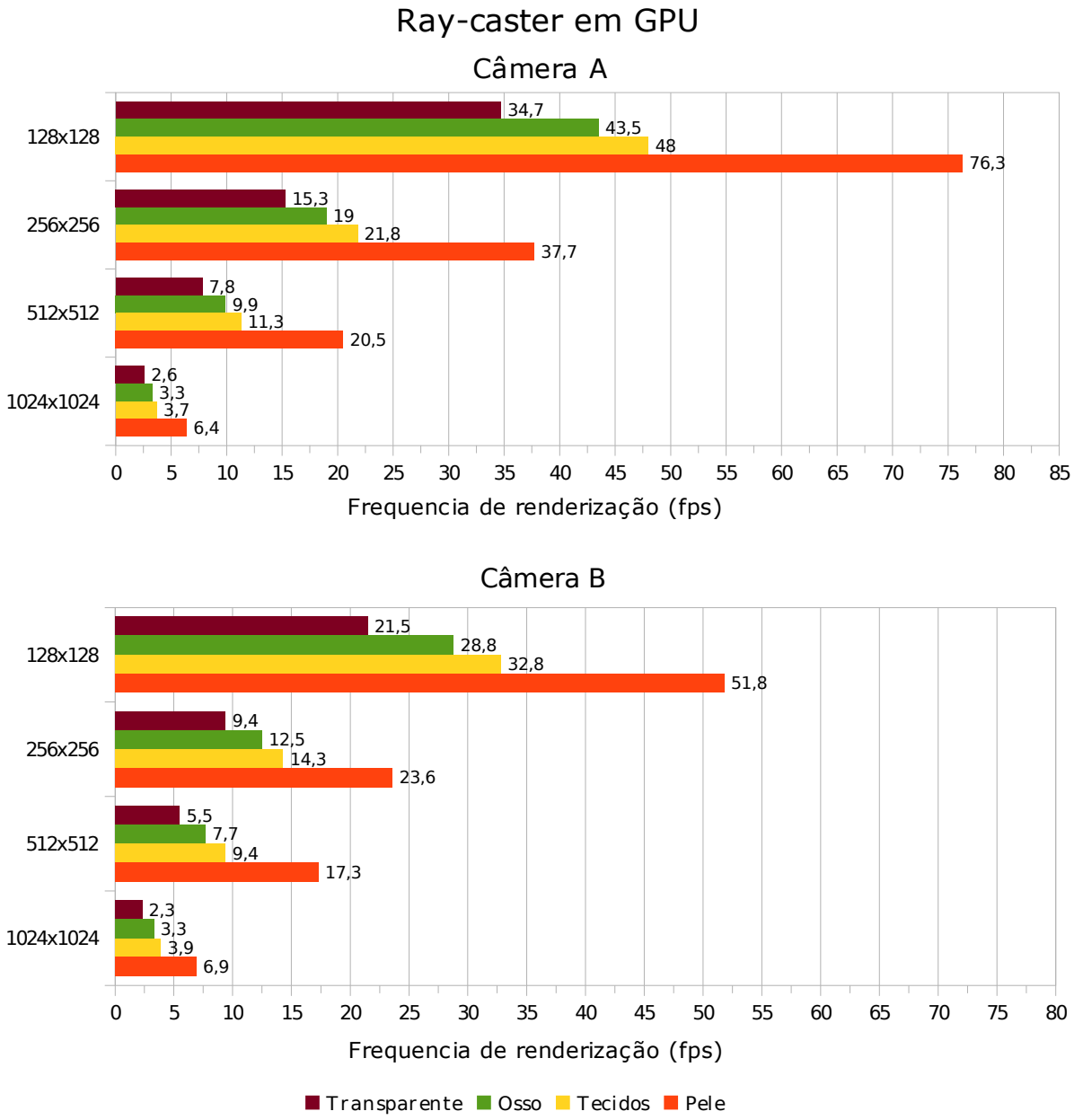


Figura 4.9: Performance do ray-caster em GPU.

ambos os eixos de uma textura 2D. Em texturas 3D, nem todas as implementações de *hardware* otimizam também no terceiro eixo (WEISKOPF, 2006).

Com esta implementação é possível que o usuário altere os parâmetros de renderização, como função de transferência e câmera, e veja o resultado imediatamente. A performance alcançada pode ser qualificada como interativa ou de tempo-real de acordo com a definição adotada no Capítulo 1.

4.3 Iluminação

O baixo tempo de renderização obtido com a implementação em GPU possibilitou a adição de novas características ao algoritmo. A qualidade de renderização foi melhorada através da implementação de um modelo de iluminação. O modelo escolhido foi o de Blinn-Phong (BLINN, 1977) por ser simples, apresentar as principais características de iluminação e ser o mesmo utilizado pelo OpenGL para primitivas geométricas. Neste modelo são considerados 3 componentes de iluminação a serem combinados: componente ambiente, difuso e especular. A equação que representa o modelo Blinn-Phong é a seguinte:

$$I = m_a i_a + \sum_{j \in Lights} (m_d j_d (\vec{L} \cdot \vec{N}) + m_s j_s (\vec{H} \cdot \vec{N})^{\alpha})$$

O resultado I representa a intensidade da luz em determinado canal de cor e as variáveis de material e luz possuem os valores correspondentes no mesmo canal de cor. As variáveis m_a, m_d e m_s representam a intensidade do material do objeto sendo iluminado para as componentes de ambiente, difusa e especular respectivamente. A variável i_a é uma constante de iluminação global e j_d e j_s são as intensidades da luz para as componentes difusa e especular. $\vec{N}$ é o vetor normal da superfície, $\vec{L}$ é o vetor direção da luz, $\vec{H}$ é o vetor entre $\vec{L}$ e o vetor em direção ao observador e α é o expoente especular.

A adaptação deste modelo para renderização de volumes substitui o vetor normal $\vec{N}$ em um ponto da superfície pelo vetor gradiente em um ponto do volume (HADWIGER et al., 2008). Para calcular o gradiente foi utilizado o método de diferenças centrais. Os gradientes e os cálculos de iluminação são feitos a cada renderização, não havendo nenhuma etapa de pré-processamento. Para simplificação, foi considerado apenas uma fonte direcional de luz. Foi atribuído à m_d o resultado da função de transferência e à m_a e m_s o valor 1.

Para obter resultados visualmente mais agradáveis, as funções de transferência para osso e pele tiveram os mapeamentos de cor alterados, pois com iluminação implementada, o artifício

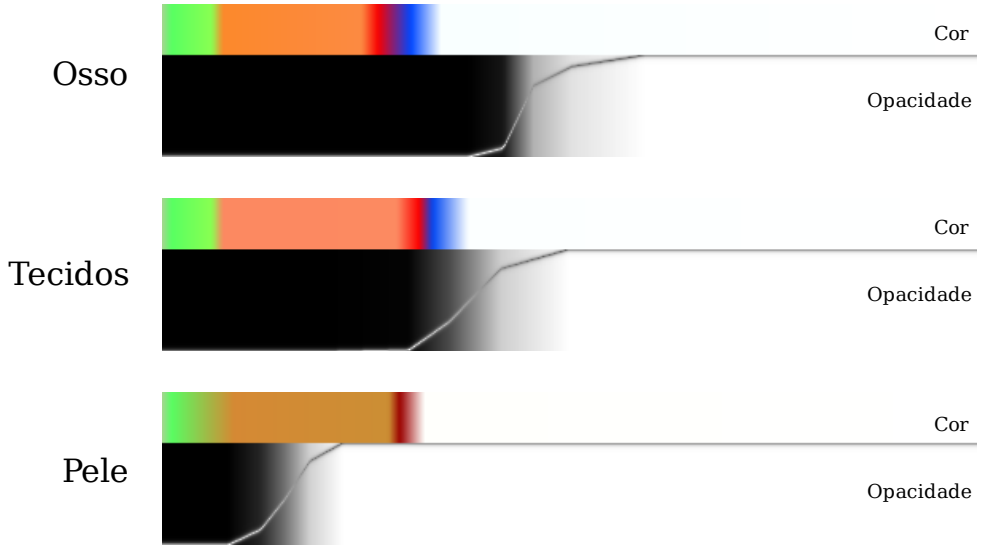


Figura 4.10: Novas funções de transferência.

de utilizar elementos mais escuros nos degradês já não é mais necessário para obter noção de profundidade. Note que isto não altera o desempenho do algoritmo, pois as funções para opacidade foram mantidas. A Figura 4.10 mostra como ficaram as novas funções de transferência e os resultados podem ser observados na Figura 4.11.

Os resultados do *benchmark* se encontram na Figura 4.12. Note que diferente dos resultados anteriores, a função de transferência do osso ficou mais rápida que a dos tecidos na maioria das situações. Isso ocorreu devido à uma otimização implementada que faz com que se calcule o gradiente e a iluminação apenas em amostras com opacidade suficiente para influenciar o resultado da composição. Desta forma se a função de transferência causa muitas amostras transparentes, como a função do osso, a renderização tende a ficar tão rápida quanto à implementação sem iluminação. Observe que por este motivo a função de transferência transparente obteve resultados muito próximos com os da implementação sem iluminação.

4.4 Projeção em perspectiva

Para tornar o renderizador mais completo, implementou-se a opção de realizar renderização com projeção em perspectiva. Assim como a iluminação esta melhoria foi feita somente no *ray-caster* em GPU. A projeção perspectiva possui os planos de corte como mostra a Figura 4.13. Note que uma característica dessa projeção é fazer com que objetos mais próximos apareçam maiores e que no algoritmo de *ray-casting* cada raio terá uma direção diferente.

A direção que o raio deve ter é a direção do vetor com origem no observador e que passa pelo ponto no plano da frente correspondente ao *pixel* que o raio representa. A etapa que deve

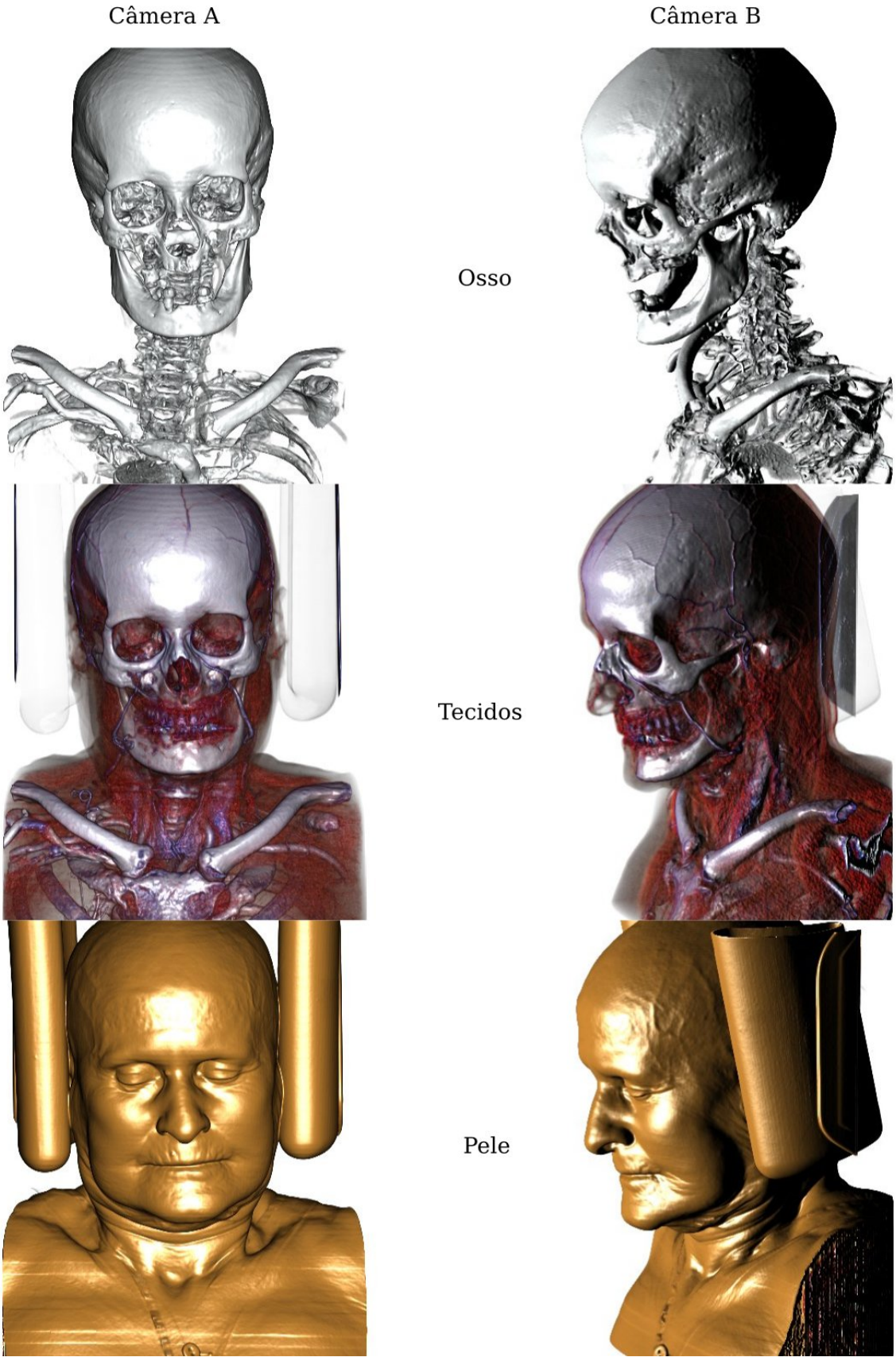


Figura 4.11: Resultados do ray-caster em GPU com iluminação.

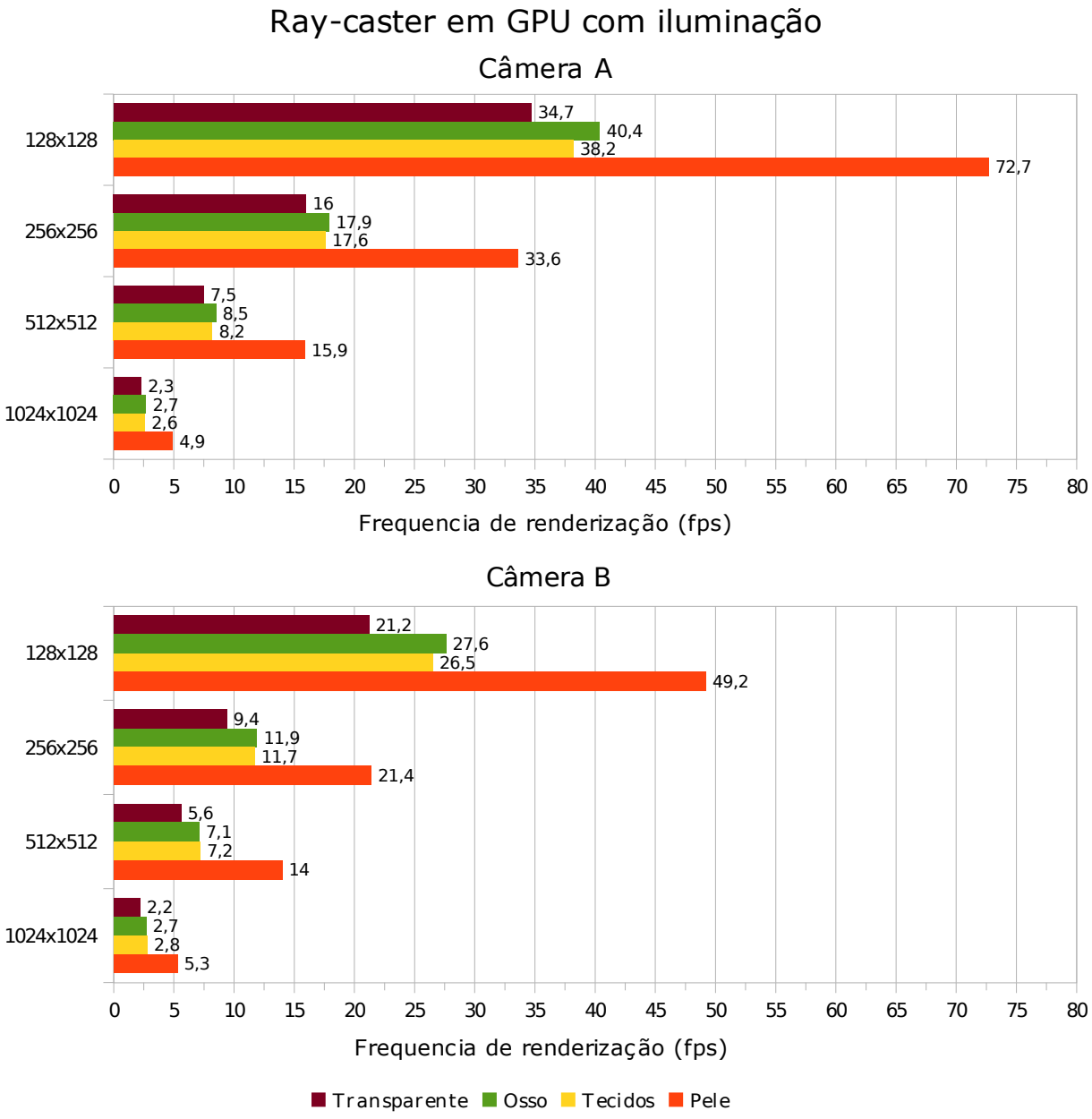


Figura 4.12: Performance do *ray-caster* em GPU com iluminação.

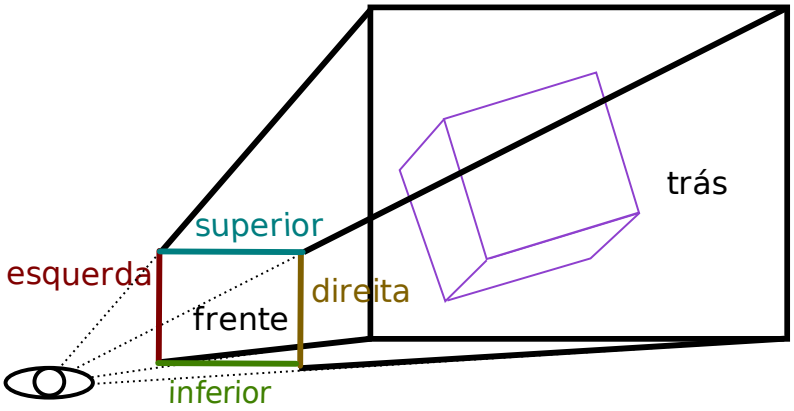


Figura 4.13: Planos de corte da projeção em perspectiva.

ser modificada é a etapa de configuração dos raios, que na implementação em GPU corresponde à geração das texturas de origem e destino dos raios. A modificação consiste apenas em rasterizar as faces frontais e traseiras da caixa utilizando projeção em perspectiva ao invés de projeção ortogonal.

A diferença entre a projeção paralela e a em perspectiva em geometria é somente os valores da matriz de projeção, portanto não há grande impacto na performance na configuração dos raios. Porém o número de amostras a serem coletadas é diferente assim como quando se altera a posição e direção da câmera. Não foram feitos *benchmarks* com projeção em perspectiva pois em experimentos simples não houve grandes diferenças comparado à simplesmente mudar a posição da câmera em projeção paralela.

A Figura 4.14 exibe os resultados obtidos com a projeção em perspectiva e câmeras com campo de visão de 120 graus.

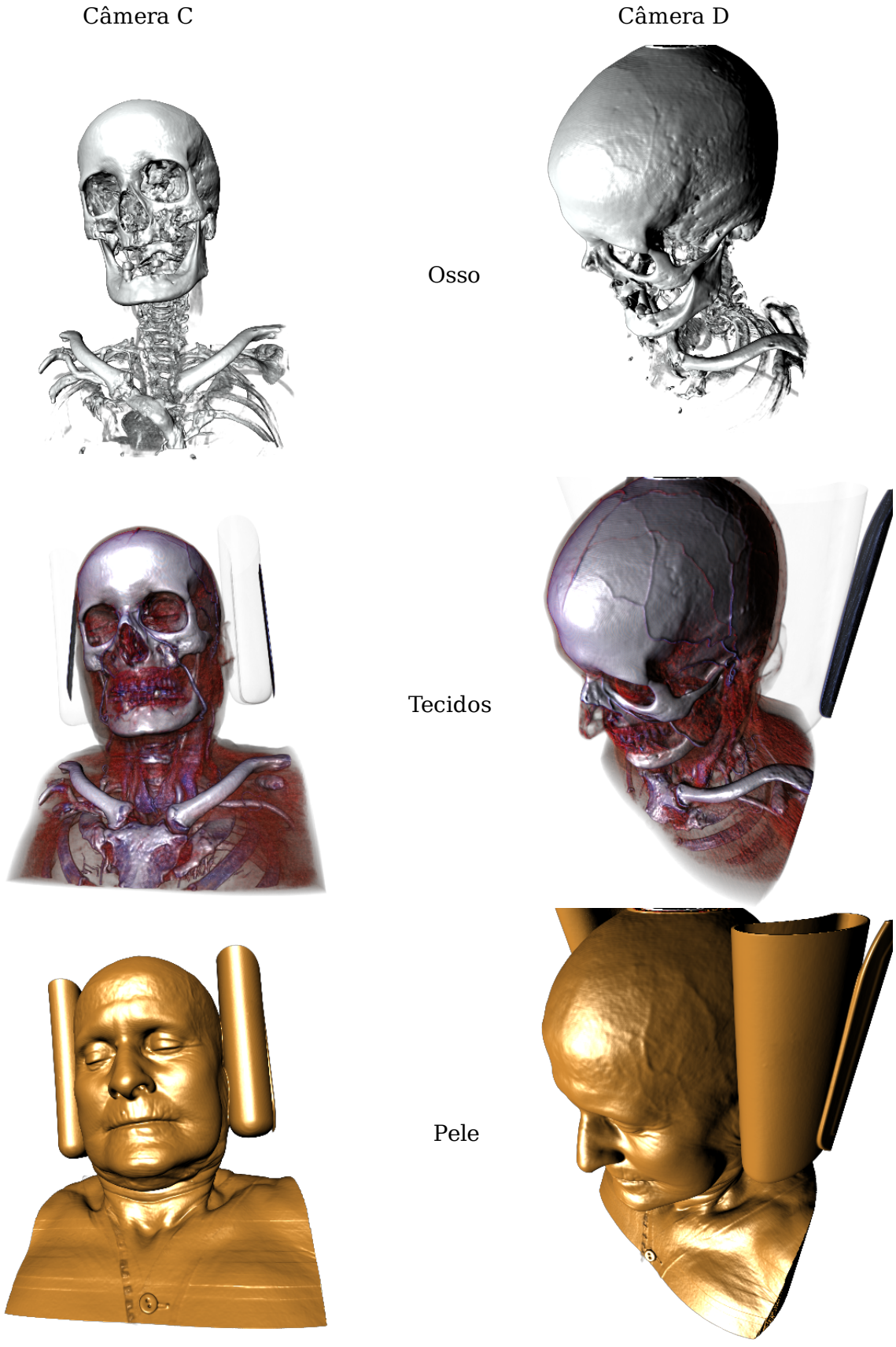


Figura 4.14: Resultados do *ray-caster* com projeção em perspectiva.

5 *Conclusões e trabalhos futuros*

Este trabalho atingiu o objetivo de realizar renderização de volumes em velocidade interativa, graças ao poder computacional dos processadores gráficos modernos. Foi possível inclusive implementar uma técnica de iluminação para gerar imagens com melhor expressividade. Um gráfico comparando as implementações se encontra na Figura 5.1, em escala logarítmica. Por último foi adicionado o suporte à projeção em perspectiva para tornar o renderizador em GPU mais completo. A aplicação enfatizada neste trabalho foi a de visualização de dados de imagens médicas, porém este mesmo algoritmo pode ser utilizado nas outras aplicações citadas no Capítulo 1 com pouca ou nenhuma modificação.

O desenvolvimento da versão em CPU foi o de programação convencional, com acesso às bibliotecas já existentes e ferramentas de depuração bem conhecidas. Porém, alguns algoritmos auxiliares para renderização volumétrica tiveram que ser implementados, como intersecção de raio com paralelepípedo, interpolação trilinear, composição de amostras e consulta a funções de transferência. Já para versão em GPU foi necessário utilizar outra linguagem para fazer o programa de renderização, configurar a interação do programa executando CPU com o executando em GPU e com depuração limitada a exibir valores na tela na forma de imagem. Por outro lado, como a linguagem GLSL é direcionada para aplicações gráficas, muitos dos algoritmos auxiliares já estavam implementados. Para especificamente o caso da implementação realizada neste trabalho programar em GPU possuiu mais vantagens do que desvantagens.

A limitação da implementação em processador gráfico se resume à memória gráfica, que de modo geral costuma ter menos capacidade que a memória de sistema. Na Seção 4.2 foi mencionada a necessidade de armazenar o volume com uma precisão numérica menor do que a original para ocupar menos memória, ou haveria queda de performance. Embora a queda de performance ainda fosse aceitável, comparado à performance em CPU, não foi feito experimentos com volumes maiores ou placas gráficas com menos memória para explorar este limite. Existem placas gráficas com mais memória, para justamente esse tipo de aplicação, porém não se teve acesso a essas placas para realizar mais experimentos. A limitação da implementação em CPU é a performance, pois não se conseguiu performance interativa com ela. Para ame-

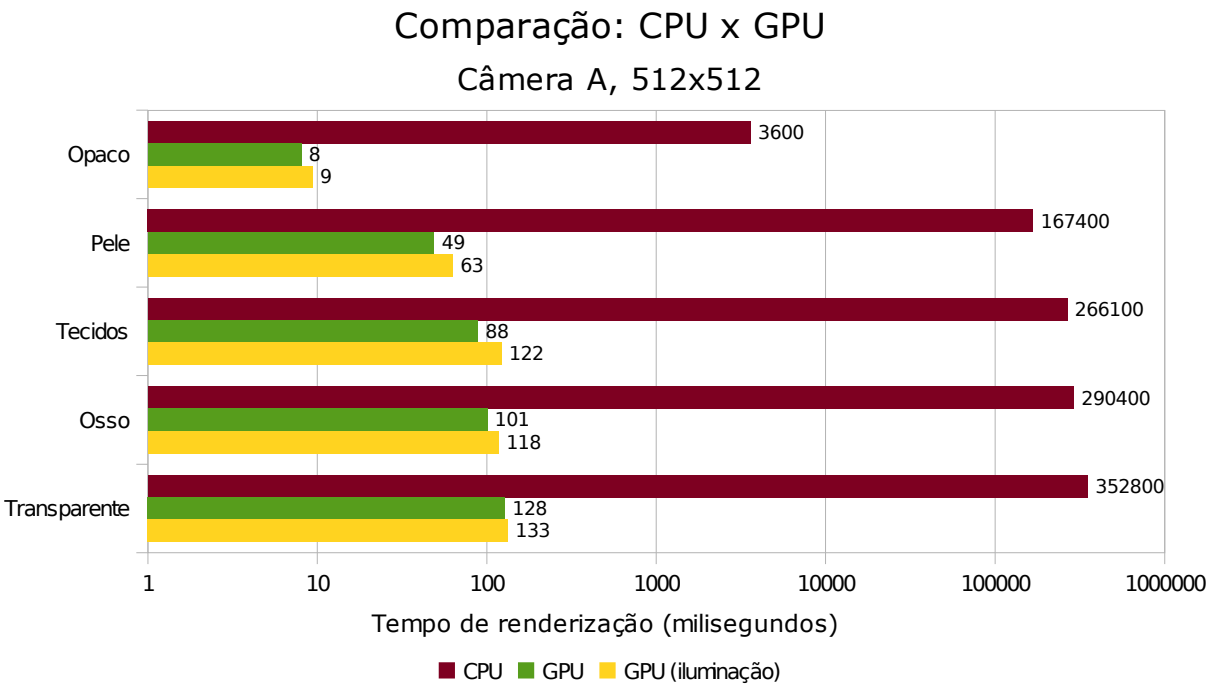


Figura 5.1: Comparação de performance entre as implementações.

nizar esta limitação, seria necessário realizar otimizações mais elaboradas e aproveitar mais o suporte a paralelismo do processador, porém como visto no Capítulo 3 as GPUs ainda têm uma tendência de crescimento de poder de processamento maior, tornando discutível se vale a pena otimizar esta implementação.

Como trabalho futuro, pode-se pensar na aplicação deste algoritmo em outros contextos, como o de visualização de dados geológicos ou moleculares, como também a integração com outras aplicações tais como renderização de simulação de dinâmica de fluido e mesclagem da renderização com primitivas geométricas. Outra ramificação de desenvolvimento seria explorar funções de transferência multidimensionais (KNISS; KINDLMANN; HANSEN, 2002) para especificar melhor as regiões do volume. É possível ainda adicionar iluminação mais avançada, considerando fenômenos como sombra e reflexão (HADWIGER et al., 2008). Outra linha de desenvolvimento seria implementar otimizações para aumentar a performance ou comprimir os dados do volume para tentar amenizar a limitação de memória das GPUs.

Referências Bibliográficas

BLINN, J. F. Models of light reflection for computer synthesized pictures. In: *SIGGRAPH '77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 1977. p. 192–198. ISSN 0097-8930. Disponível em: <http://dx.doi.org/10.1145/563858.563893>.

BROWN, P.; LEECH, J.; MACE, R.; PAUL, B. *ARB_texture_float*. February 2008. Disponível em: <http://www.opengl.org/registry/specs/ARB/texture_float.txt>.

CHEN, W.; REN, L.; ZWICKER, M.; PFISTER, H. Hardware-accelerated adaptive ewa volume splatting. In: *VIS '04: Proceedings of the conference on Visualization '04*. Washington, DC, USA: IEEE Computer Society, 2004. p. 67–74. ISBN 0-7803-8788-0. Disponível em: <http://dx.doi.org/10.1109/VIS.2004.38>.

CHERNYAEV, E. V. *Marching Cubes 33: Construction of Topologically Correct Isosurfaces*. [S.l.], 1995. Disponível em: <http://citeseer.ist.psu.edu/old/chernyaev95marching.html>.

DIETRICH, C. A.; SCHEIDEGGER, C.; COMBA, J. L. D.; NEDEL, L.; SILVA, C. T. Edge groups: An approach to understanding the mesh quality of marching methods. *Visualization and Computer Graphics, IEEE Transactions on*, v. 14, n. 6, p. 1651–1666, 2008. Disponível em: <http://dx.doi.org/10.1109/TVCG.2008.122>.

DREBIN, R. A.; CARPENTER, L.; HANRAHAN, P. Volume rendering. In: *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 1988. p. 65–74. ISBN 0-89791-275-6. Disponível em: <http://dx.doi.org/10.1145/54852.378484>.

DYKEN, C.; ZIEGLER, G.; THEOBALT, C.; SEIDEL, H. P. *GPU Marching Cubes on Shader Model 3.0 and 4.0*. Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany, August 2007. Disponível em: <http://domino.mpi-inf.mpg.de/internet/reports.nsf/NumberView/2007-4-006>.

ELVINS, T. T. A survey of algorithms for volume visualization. *ACM SIGGRAPH Computer Graphics*, v. 26, n. 3, p. 194–201, 1992. Disponível em: <http://portal.acm.org/citation.cfm?doid=142413.142427>.

ENGEL, K.; KRAUS, M.; 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: ACM Press, 2001. p. 9–16. ISBN 158113407X. Disponível em: <http://dx.doi.org/10.1145/383507.383515>.

FERNANDO, R. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education, 2004. ISBN 0321228324. Disponível em: <http://portal.acm.org/citation.cfm?id=983868>.

FOLEY, J. D.; van Dam, A.; FEINER, S. K.; HUGHES, J. F. *Computer graphics (2nd ed. in C): principles and practice*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996. ISBN 0-201-84840-6. Disponível em: <http://portal.acm.org/citation.cfm?id=208249>.

GINGOLD, R. A.; MONAGHAN, J. J. Smoothed particle hydrodynamics - theory and application to non-spherical stars. *Mon. Not. Roy. Astron. Soc.*, v. 181, p. 375–389, November 1977. Disponível em: <http://adsabs.harvard.edu/cgi-bin/nph-bib_query?bibcode=1977MNRAS.181..375G>.

HADWIGER, M.; KNISS, J. M.; REZK-SALAMA, C.; WEISKOPF, D. *Real-time Volume Graphics*. 1. ed. [S.l.]: A K Peters, 2006. Hardcover. ISBN 1568812663.

HADWIGER, M.; LJUNG, P.; SALAMA, C. R.; ROPINSKI, T. Advanced illumination techniques for gpu volume raycasting. In: *SIGGRAPH Asia '08: ACM SIGGRAPH ASIA 2008 courses*. New York, NY, USA: ACM, 2008. p. 1–166. Disponível em: <http://dx.doi.org/10.1145/1508044.1508045>.

HUI-ZHANG; JAE-CHOI. High quality gpu rendering with displaced pixel shading. SPIE - The International Society for Optical Engineering, February 2006. Disponível em: <http://www.isis.georgetown.edu/CAIMR/DesktopModules/ViewDocument.aspx?DocumentID=148>.

KESSENICH, J. *The OpenGL Shading Language*. [S.l.], February 2009. Disponível em: <http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.40.05.pdf>.

KESSENICH, J.; BALDWIN, D.; ROST, R. *The OpenGL Shading Language*. [S.l.], April 2004. Disponível em: <http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.10.59.pdf>.

KNISS, J.; KINDLMANN, G.; HANSEN, C. Multidimensional transfer functions for interactive volume rendering. *Visualization and Computer Graphics, IEEE Transactions on*, IEEE Educational Activities Department, Piscataway, NJ, USA, v. 8, n. 3, p. 270–285, November 2002. ISSN 1077-2626. Disponível em: <http://dx.doi.org/10.1109/TVCG.2002.1021579>.

KRUGER, J.; WESTERMANN, R. Acceleration techniques for gpu-based volume rendering. In: *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*. Washington, DC, USA: IEEE Computer Society, 2003. ISBN 0769520308. Disponível em: <http://dx.doi.org/10.1109/VIS.2003.10001>.

LEVOY, M. Display of surfaces from volume data. *Computer Graphics and Applications, IEEE*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 8, n. 3, p. 29–37, May 1988. ISSN 0272-1716. Disponível em: <http://dx.doi.org/10.1109/38.511>.

LEVOY, M. Efficient ray tracing of volume data. *ACM Trans. Graph.*, ACM Press, New York, NY, USA, v. 9, n. 3, p. 245–261, July 1990. ISSN 0730-0301. Disponível em: <http://dx.doi.org/10.1145/78964.78965>.

LORENSEN, W. E.; 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: ACM Press, 1987. v. 21, n. 4, p. 163–169. ISSN 0097-8930. Disponível em: <http://dx.doi.org/10.1145/37401.37422>.

LUEBKE, D.; HARRIS, M.; KRÜGER, J.; PURCELL, T.; GOVINDARAJU, N.; BUCK, I.; WOOLLEY, C.; LEFOHN, A. Gpgpu: general purpose computation on graphics hardware. In: *SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes*. New York, NY, USA: ACM Press, 2004. Disponível em: <http://dx.doi.org/10.1145/1103900.1103933>.

MAX, N. L. *Optical Models for Direct Volume Rendering*. 1995. 99–108 p. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.17.5902>.

MÖLLER, T. A.; HAINES, E.; HOFFMAN, N. *Real-Time Rendering 3rd Edition*. Natick, MA, USA: A. K. Peters, Ltd., 2008. 1045 p.

NEOPHYTOU, N.; MUELLER, K. Gpu accelerated image aligned splatting. In: *Volume Graphics, 2005. Fourth International Workshop on*. [s.n.], 2005. p. 197–242. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1500542>.

NEWMAN, T. S.; YI, H. A survey of the marching cubes algorithm. *Computers & Graphics*, v. 30, n. 5, p. 854–879, October 2006. Disponível em: <http://dx.doi.org/10.1016/j.cag.2006.07.021>.

NGUYEN, H. *GPU Gems 3*. [S.l.]: Addison-Wesley Professional, 2007. Hardcover. ISBN 0321515269.

OPENGL; SHREINER, D.; WOO, M.; NEIDER, J.; DAVIS, T. *OpenGL(R) Programming Guide : The Official Guide to Learning OpenGL(R), Version 2 (5th Edition)*. [S.l.]: Addison-Wesley Professional, 2005. Paperback. ISBN 0321335732.

PORTER, T.; DUFF, T. Compositing digital images. In: *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 1984. p. 253–259. ISBN 0-89791-138-5. ISSN 0097-8930. Disponível em: <http://dx.doi.org/10.1145/800031.808606>.

SCHARSACH, H. Advanced gpu raycasting. In: *Proceedings of the 9th Central European Seminar on Computer Graphics*. [s.n.], 2005. Disponível em: <http://www.cg.tuwien.ac.at/hostings/cescg/CESCG-2005/papers/VRVis-Scharsach-Henning.pdf>.

SEGAL, M.; AKELEY, K. *The OpenGL Graphics System: A Specification*. [S.l.], December 2006. Disponível em: <http://www.opengl.org/registry/doc/glspec21.20061201.pdf>.

SILVA, A. F. B.; NOBREGA, T. H. C.; CARVALHO, D. D. B.; INÁCIO, R. T.; von Wangenheim, A. Framework for interactive medical imaging applications. In: *COLIBRI 2009 - Colloquium of Computation: Brazil / INRIA, Cooperations, Advances and Challenges*. Porto Alegre: Sociedade Brasileira de Computação - SBC, 2009. p. 126–129.

TAMASI, T. Evolution of computer graphics. In: *NVISION 08*. San Jose, California: NVIDIA Corporation, 2008.

van Baar, J.; GROSS, M.; ZWICKER, M.; ZWICKER, M.; PFISTER, H.; PFISTER, H.; JEROEN; MARKUS. Ewa volume splatting. In: *IEEE Visualization*. [s.n.], 2001. v. 2001, n. 2001, p. 29–36. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.3.4788>.

VEGA-HIGUERA, F.; HASTREITER, P.; FAHLBUSCH, R.; GREINER, G. High performance volume splatting for visualization of neurovascular data. In: *Visualization, 2005. VIS 05. IEEE*. [s.n.], 2005. p. 271–278. Disponível em: <http://dx.doi.org/10.1109/VISUAL.2005.1532805>.

WEISKOPF, D. *GPU-Based Interactive Visualization Techniques (Mathematics and Visualization)*. 1. ed. Springer, 2006. Hardcover. ISBN 3540332626. Disponível em: <http://www.worldcat.org/isbn/3540332626>.

WESTOVER, L. Interactive volume rendering. In: *VVS '89: Proceedings of the 1989 Chapel Hill workshop on Volume visualization*. New York, NY, USA: ACM, 1989. p. 9–16. Disponível em: <http://dx.doi.org/10.1145/329129.329138>.

WESTOVER, L. Footprint evaluation for volume rendering. In: *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 1990. p. 367–376. ISBN 0-201-50933-4. Disponível em: <http://dx.doi.org/10.1145/97879.97919>.

WILLIAMS, L. Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph.*, ACM, New York, NY, USA, v. 12, n. 3, p. 270–274, 1978. ISSN 0097-8930. Disponível em: <http://dx.doi.org/10.1145/800248.807402>.

WILSON, O.; VANGELDER, A.; WILHELMS, J. *Direct Volume Rendering via 3D Textures*. Santa Cruz, CA, USA, 1994. Disponível em: <http://portal.acm.org/citation.cfm?id=902673>.

WITTENBRINK, C. M.; MALZBENDER, T.; GOSS, M. E. Opacity-weighted color interpolation for volume sampling. In: *Volume Visualization, 1998. IEEE Symposium on*. [s.n.], 1998. p. 135–142. Disponível em: <http://dx.doi.org/10.1109/SVV.1998.729595>.