Aceleração da Técnica de Cubos Marchantes para Visualização Volumétrica com Placas Gráficas

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Marcos Vinícius Mussel Cirne e aprovada pela Banca Examinadora.

Campinas, agosto de 2011.

Prof. Dr. Hélio Pedrini (Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

ERRATA

Na folha **i** e **ii** - **Onde se lê:** Marcos Vinícius Mussel Cirne **Leia-se:** Marcos Vinicius Mussel Cirne

Prof. Dr. Paulo Licio de Geus
Coord. de Pós-Graduação
Instituto de Computação - Unicamp
Matrícula 10.326-8

FICHA CATALOGRÁFICA ELABORADA POR
MARIA FABIANA BEZERRA MÜLLER - CRB8/6162
BIBLIOTECA DO INSTITUTO DE MATEMÁTICA, ESTATÍSTICA E COMPUTAÇÃO CIENTÍFICA - UNICAMP

C496a

Cirne, Marcos Vinícius Mussel, 1987-
Aceleração da técnica de cubos marchantes para visualização volumétrica com placas gráficas / Marcos Vinícius Mussel Cirne. - Campinas, SP : [s.n.], 2011.

Orientador: Hélio Pedrini.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Computação gráfica. 2. Visualização. 3. Imagem tridimensional. 4. Processamento de imagens. I. Pedrini, Hélio,1963-. II. Universidade Estadual de Campinas. Instituto de Computação. III.Título.

Informações para Biblioteca Digital

Título em inglês: Acceleration of the marching cubes technique for volum visualization with graphics cards
Palavras-chave em inglês:
Computer graphics
Visualization
Three-dimensional imaging
Image processing
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Hélio Pedrini [Orientador]
Fátima de Lourdes dos Santos Nunes Marques
Anamaria Gomide
Data da defesa: 26-08-2011
Programa de Pós-Graduação: Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 26 de agosto de 2011, pela Banca examinadora composta pelos Professores Doutores:

Profª Drª Fátima de Lourdes dos Santos Nunes Marques
EACH / USP

Profª Drª Anamaria Gomide
IC / UNICAMP

Prof. Dr. Hélio Pedrini
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Aceleração da Técnica de Cubos Marchantes para Visualização Volumétrica com Placas Gráficas

Marcos Vinícius Mussel Cirne[1]

Outubro de 2011

Banca Examinadora:

- Prof. Dr. Hélio Pedrini (Orientador)
- Profa. Dra. Fátima Nunes
  Escola de Artes, Ciências e Humanidades – EACH/USP
- Profa. Dra. Anamaria Gomide
  Instituto de Computação – IC/UNICAMP

[1]Suporte financeiro de: Bolsa do CNPq (Processo 573710/2008-2 Edital MCT/CNPq Nº 015/2008 - Institutos Nacionais de Ciência e Tecnologia)

Resumo

A visualização volumétrica possui uma série de aplicações que beneficiam diversos campos do conhecimento, como Medicina, Biologia, Geologia, Oceanografia, Meteorologia, entre outros, no que tange à análise e compreensão de fenômenos estudados em cada um deles. Ao longo dos anos, houve uma necessidade cada vez maior de se obter renderizações de volumes de dados em tempo real de uma maneira rápida, eficiente e realista. Para isso, optou-se pela aceleração do processo de visualização por meio de poderosas unidades de processamento gráfico, que são capazes de executar operações primitivas muito mais rapidamente do que processadores de propósito geral, devido ao seu alto grau de paralelismo.

O objetivo deste trabalho é descrever as principais técnicas de visualização volumétrica, com ênfase no algoritmo de cubos marchantes, utilizando tecnologias recentes de programação em GPU. A metodologia proposta para acelerar este algoritmo em unidades de processamento gráfico por meio de estruturas de dados espaciais é apresentada e discutida. Experimentos realizados com o uso de vários volumes de dados demonstram a eficácia do método desenvolvido com a arquitetura CUDA.

Abstract

Volumetric visualization has a great number of applications which benefit several knowledge domains, such as Medicine, Biology, Geology, Oceanography, Meteorology, among others, concerning the analysis and comprehension of phenomena studied at each one. Over the years, there was an increasingly need to achieve real-time volume rendering in a fast, efficient and realistic way. For that, the visualization process was accelerated by powerful graphics processing units (GPU), which are capable of executing primitive operations much faster than general-purpose processors, due to their high degree of parallelism.

The objective of this work is to describe the main volumetric visualization techniques, with emphasis on the marching cubes algorithm, using the most recent GPU programming technologies. The proposed methodology to accelerate this algorithm on GPUs by means of spatial data structures is shown and discussed. Experiments conducted with use of several volume datasets demonstrate the effectiveness of the developed method with the CUDA architecture.

Agradecimentos

À minha família, pelo grande apoio que sempre me deu, mesmo nas horas mais difíceis.

Ao meu orientador, Hélio Pedrini, por despertar o meu interesse na área de Computação Gráfica e pelas várias ideias sugeridas neste trabalho, que serviram como um grande estímulo.

À professora Fátima Nunes, da EACH/USP, pela vinculação deste trabalho ao projeto do INCT-MACC (Instituto Nacional de Ciência e Tecnologia – Medicina Assistida por Computação Científica) e pela consequente concessão da bolsa por parte da CAPES.

Ao pessoal do LIV pelo convívio durante esses mais de dois anos de Mestrado.

Sumário

Lista de Tabelas

Lista de Figuras

Lista de Algoritmos

Capítulo 1

Introdução

Visualização volumétrica é uma abordagem para extração de informações a partir de dados volumétricos e está relacionada à representação, manipulação e renderização desses dados. Ela veio à tona durante a década de 1990 como um ramo da visualização científica e possui aplicações em vários campos do conhecimento humano, tais como Medicina, Biologia, Geologia, Oceanografia, Meteorologia, entre outros, auxiliando a análise e compreensão de fenômenos estudados em cada um deles.

Apesar dos avanços significativos na área ao longo dos anos, foi observado que o uso de uma unidade de processamento central (CPU, do inglês *central processing unit*) para desempenhar tarefas gráficas de propósito geral não era suficiente para se atingir uma melhor interatividade com os usuários ou ainda uma renderização em tempo real, especialmente quando são utilizadas grandes quantidades de dados. No entanto, as constantes evoluções tecnológicas permitiram o surgimento de poderosas unidades de processamento gráfico (GPU, do inglês *graphics processing units*), as quais são capazes de renderizar modelos tridimensionais complexos, obtendo-se um alto grau de realismo.

O potencial das GPUs é impulsionado pelo seu alto nível de paralelismo e sua habilidade em realizar operações em ponto flutuante e com primitivas geométricas de uma forma rápida e eficiente. Recentemente, as GPUs têm sido bastante utilizadas para a aceleração de aplicações (tais como processamento de imagens e vídeos, simulação de dinâmica de fluidos, análise de atividade sísmica, entre outras), resultando em um ganho significativo em relação às CPUs por uma ordem de magnitude. Isso foi possível graças ao desenvolvimento da computação de propósito geral em GPUs (GPGPU, do inglês *general-purpose computing on graphics processing units*), que fez com que as arquiteturas das GPUs fossem projetadas com um foco maior na programabilidade do *hardware* gráfico, tornando-as cada vez mais flexíveis. A ideia desta técnica é aproveitar ao máximo o alto paralelismo das GPUs para otimizar o desempenho de aplicações em geral.

Mesmo que as GPUs consigam realizar um processamento gráfico mais rápido do que

as CPUs, nem todas as aplicações desenvolvidas na GPU conseguem ser mais rápidas do que suas respectivas versões em CPU. Em alguns casos, não é possível paralelizar os procedimentos que compõem uma determinada aplicação. Além disso, não vale a pena encaminhar tarefas simples para a GPU, pois há um desperdício de tempo com a transferência de informações da CPU para a GPU.

Atualmente, a tecnologia de GPGPU mais utilizada é a arquitetura CUDA (do inglês *Computing Unified Device Architecture*), da NVIDIA [39], que permite a utilização da linguagem de programação C, no que diz respeito ao desenvolvimento de algoritmos para serem executados em GPUs. A desvantagem no uso dessa arquitetura é o fato de ela ser restrita às placas gráficas da NVIDIA. Entretanto, essa restrição é compensada com um bom aproveitamento dos recursos de *hardware* dessas placas, garantindo um ótimo desempenho das suas aplicações.

Como consequência da evolução dessas tecnologias, as técnicas de visualização volumétrica também obtiveram avanços consideráveis. A aceleração dessas técnicas em GPUs tornou-se uma ferramenta bastante eficaz para a visualização e a análise de dados volumétricos, obtendo-se altas taxas de renderização em tempo real.

Um dos algoritmos de visualização volumétrica mais utilizados é o de cubos marchantes [28], a qual se enquadra em uma categoria especial de técnicas denominada extração de isossuperfícies. Vários estudos já foram realizados com o objetivo de melhorar o desempenho deste algoritmo, os principais baseiam-se no uso de estruturas de dados espaciais para armazenar os dados volumétricos e otimizar a busca pelas células que serão renderizadas, modificando a complexidade do algoritmo.

1.1 Objetivos e Contribuições

O objetivo deste trabalho é descrever alguns dos principais métodos de visualização volumétrica, juntamente com tentativas de aceleração em GPU com o uso das tecnologias mais recentes de programação em GPU. Ênfase é dada ao algoritmo de cubos marchantes, descrevendo-se também como ele pode ser acelerado na GPU e otimizado a partir de estruturas de dados espaciais, que alteram o comportamento assintótico do algoritmo. Em seguida, uma abordagem é apresentada para a implementação desta técnica juntamente com uma aplicação que utiliza a arquitetura CUDA, incluindo descrições dos algoritmos utilizados. Finalmente, análises de desempenho são efetuadas, em que se comparam os resultados obtidos pela metodologia proposta e por abordagens já existentes, mostrando o ganho de desempenho da GPU em relação à CPU.

A aplicação implementada neste trabalho é capaz de manipular volumes de diferentes tamanhos e que são objetos de estudo em vários campos de conhecimento, o que permite a agregação de uma série de conhecimentos a partir desses volumes. Além disso, ela pode

ser utilizada como um *framework* para a integração em diversos ambientes de visualização, obtendo-se altas taxas de renderização em tempo real.

1.2 Organização do Texto

Esta dissertação está organizada da seguite maneira: o Capítulo 2 faz uma revisão de alguns dos conceitos relevantes sobre GPUs, visualização volumétrica e extração de isossuperfícies, além da descrição do algoritmo de cubos marchantes e de estruturas de dados utilizadas para a sua aceleração (incluindo algumas das abordagens existentes). O Capítulo 3 descreve a metodologia proposta para a aceleração da técnica de cubos marchantes em GPU, contendo também pseudocódigos dos principais algoritmos utilizados. O Capítulo 4 faz uma discussão acerca dos resultados experimentais obtidos com a aplicação do método proposto, comparando-o com outros já existentes. Finalmente, o Capítulo 5 mostra as conclusões desta dissertação, juntamente com possibilidades de trabalhos futuros.

Capítulo 2

Conceitos e Trabalhos Relacionados

Este capítulo apresenta uma descrição acerca da arquitetura geral das GPUs e da visualização volumétrica, seguida pelos conceitos de extração de isossuperfícies, do algoritmo de cubos marchantes e de algumas das estruturas de dados espaciais utilizadas para melhorar o desempenho deste algoritmo.

2.1 Unidade de Processamento Gráfico

Uma unidade de processamento gráfico (GPU) é um processador dedicado, com alto grau de paralelismo, e que possui a especialidade de renderizar objetos 2D e 3D, produzindo resultados realistas de uma maneira eficiente. No começo, as GPUs simplesmente atuavam como um processador auxiliar à CPU, possuindo somente um núcleo com funcionalidades fixas e eram voltadas unicamente para o processamento gráfico. À medida que inovações tecnológicas surgiram, as GPUs se transformaram em um conjunto de núcleos de processamento programáveis e voltados para a computação de propósito geral, sendo aplicadas em áreas do conhecimento que vão além da Computação Gráfica, como Visão Computacional [40], Bioinformática [8], Criptografia [16], entre outras.

A arquitetura de uma GPU é definida a partir de um modelo computacional, no qual um conjunto de vértices que compõem uma cena tridimensional passa por uma série de estágios de processamento até a exibição da imagem 2D associada a esse conjunto. Tal modelo é conhecido como *pipeline* gráfico.

De acordo com Owens et al. [42], os principais estágios que formam o *pipeline* gráfico são:

- **Operações de Vértices**: consiste no processamento inicial do conjunto de vértices passado como entrada para o *pipeline*. Cada vértice apresenta diversas informações, tais como posição 3D, cor, vetor normal e textura. Os vértices são encaminha-

dos para um processador de vértices, onde são mapeados para o espaço da tela, efetuando-se também os cálculos de iluminação. Essas transformações são feitas por funções especiais, chamadas de *shaders* de vértices, servindo também para alterar as funcionalidades deste estágio do *pipeline*. Além disso, uma vez que a computação desses vértices é feita de maneira independente, esse estágio é bem adequado para a sua paralelização em *hardware*. Entretanto, o *shader* de vértices não pode adicionar nem remover vértices do conjunto inicial, além do fato de um vértice não poder acessar as informações de outros.

- **Geração de Primitivas**: depois das transformações realizadas na etapa anterior, os vértices são representados por primitivas geométricas (em geral, triângulos). Esse procedimento é realizado pelos *shaders* de geometria, os quais também são responsáveis por determinar a visibilidade de cada primitiva gerada.

- **Rasterização**: é o processo em que são definidos os valores exatos dos pixels da imagem resultante, com base nas primitivas geométricas geradas. Cada primitiva é decomposta em um ou mais fragmentos, que são encaminhados para o processador de fragmentos.

- **Operações de Fragmentos**: nesta etapa, os fragmentos passam por um processador de fragmentos, que calcula os valores finais de cor de cada um deles por meio de *shaders* de fragmentos. Para isso, são utilizadas as informações de cor e de textura dos vértices de entrada do *pipeline* gráfico. Assim como na etapa de processamento de vértices, os fragmentos podem ser processados em paralelo, havendo também restrições de acesso, em que um fragmento não pode acessar propriedades de outros. Normalmente, esta etapa é a que apresenta um maior custo computacional dentro do *pipeline*.

- **Composição**: é a etapa em que os fragmentos são transformados na imagem final. Os pixels que compõem essa imagem são transferidos para uma região específica da memória de vídeo denominada *framebuffer*, responsável pelo procedimento de exibição desses pixels na tela.

Ao longo dos anos, os *shaders* foram adquirindo maiores capacidades, tanto em relação à usabilidade de recursos advindos do *hardware* gráfico, quanto às suas programabilidades, fazendo com que as arquiteturas das GPUs mais modernas fossem mais focadas nas partes programáveis do *pipeline* gráfico. Com isso, a computação de propósito geral em GPUs (GPGPU) foi evoluindo cada vez mais, juntamente com a complexidade dos modelos de programação em GPU.

2.2 Visualização Volumétrica

A Visualização Volumétrica consiste em um conjunto de técnicas utilizadas para o estudo de objetos e de fenômenos naturais, provenientes de diversas áreas do conhecimento e que são representados sob a forma de dados volumétricos tridimensionais [21]. A ideia básica dessas técnicas é realizar uma projeção bidimensional (geralmente em uma tela de computador) a partir desses volumes, com o intuito de estudar as estruturas que os compõem.

A aquisição desses dados é normalmente feita por escaneamento de objetos, como ocorre na tomografia computadorizada (CT, do inglês, *Computed Tomography*), na ressonância magnética (MRI, do inglês, *Magnetic Resonance Imaging*) e na microscopia confocal. Ela também pode ser feita de maneira simulada, por meio de objetos sintéticos (como blocos de uma determinada textura) ou métodos estocásticos.

Em geral, os dados volumétricos são representados por um conjunto de elementos de volume, denominados *voxels*, em que cada um deles contém um valor específico em uma grade regular contida no espaço tridimensional. Um voxel pode ser definido por uma tupla $< x, y, z, S >$, que representa o valor $S \in \mathbb{R}$ associado a alguma propriedade (como cor, opacidade, densidade, velocidade), localizado em um posição 3D genérica (x, y, z). No caso de imagens médicas (CT e MRI), o volume é representado por um conjunto de fatias 2D obtidas de maneira regular, isto é, possuem as mesmas dimensões e contêm o mesmo número de pixels.

Segundo Elvins [11], os algoritmos fundamentais de visualização volumétrica podem ser classificados em dois grupos:

- **Renderização Direta de Volume** (DVR, do inglês, *Direct Volume Rendering*): é caracterizado pelo mapeamento direto de um voxel no espaço da tela, sem a utilização de primitivas geométricas como uma representação intermediária, sendo também apropriada para criar imagens a partir de conjuntos de dados que representam substâncias amorfas, como nuvens, fluidos e gases. Os volumes de dados são representados por voxels, que são mapeados para pixels e armazenados em uma imagem 2D, dispensando assim o uso de primitivas geométricas. Essa técnica possui a vantagem de produzir imagens de excelente qualidade, uma vez que todos os voxels podem ser usados na síntese das imagens, possibilitando a visualização do interior dos objetos [29]. Por outro lado, é uma técnica computacionalmente custosa. Algumas das técnicas de DVR, detalhadas mais adiante, incluem: *Raycasting* [26], *Splatting* [55], *Cell-Projection* [57] e *Shear-Warp* [25].

- **Renderização de Superfície** (SF, do inglês, *Surface-Fitting*): consiste em estágios de extração de características e representação de isossuperfícies (superfícies que re-

presentam um conjunto de pontos com o mesmo valor escalar), que são posteriormente renderizadas para a visualização do volume de dados. Essas isossuperfícies podem ser definidas através de primitivas de superfície (como polígonos) ou por um procedimento de limiarização. Dentre as vantagens desta técnica estão a velocidade para se gerar a imagem final e o pouco espaço de armazenamento exigido. Além disso, os métodos de SF são tipicamente mais rápidos do que os de DVR, uma vez que eles somente atravessam o volume uma única vez para extrair as superfícies. No entanto, ela sofre de uma série de problemas, como o surgimento de partes de superfícies que podem ser falsos positivos ou negativos e a manipulação incorreta de pequenos detalhes dos objetos representados pelos dados. Algumas das técnicas de SF existentes são: Cubos Marchantes [28] (e sua versão modificada, chamada de Tetraedros Marchantes [4, 14]) e Conexão de Contornos [22].

Um dos grandes desafios da Visualização Volumétrica está relacionado à renderização de volumes cada vez maiores, uma vez que a taxa de renderização depende diretamente do tamanho dos dados a serem visualizados. Mesmo com as recentes evoluções do *hardware* gráfico, bem como as tentativas de implementação dos algoritmos de visualização existentes em GPU [17, 23, 47, 51], ainda existe uma demanda no desenvolvimento de novos métodos de visualização, desejando-se uma maior interatividade em tempo real na manipulação de grandes quantidades de dados.

As próximas subseções descrevem brevemente algumas das técnicas de visualização volumétrica, além de abordagens existentes para a aceleração em GPU.

2.2.1 Raycasting

O algoritmo de *raycasting* [26] consiste em realizar uma varredura dos pixels da janela de visualização e do traçado de raios, a partir dos pixels presentes na tela, na direção do volume de dados. Os voxels percorridos por um determinado raio são calculados e usados para determinar a cor e a opacidade do pixel correspondente [21]. Desta forma, os principais elementos deste algoritmo são o volume de dados (juntamente com suas propriedades), os parâmetros de visualização, a janela de visualização e os raios ao longo dos quais são processados os pontos de amostragem. [29]. Um esquema geral desta técnica é mostrado na Figura 2.1.

Esta técnica possui a vantagem de ser simples para implementar e ser facilmente paralelizável. Entretanto, ela pode gerar artefatos (como buracos e desoclusão) nas imagens resultantes, além do fato de o custo ser proporcional ao tamanho do volume.

Com relação às abordagens mais recentes para aceleração desta técnica em GPU, Scharsach [49] propôs um método que realiza o *raycasting* no *shader* de fragmentos, introduzindo alguns modos de renderização como movimentações interativas de câmera para

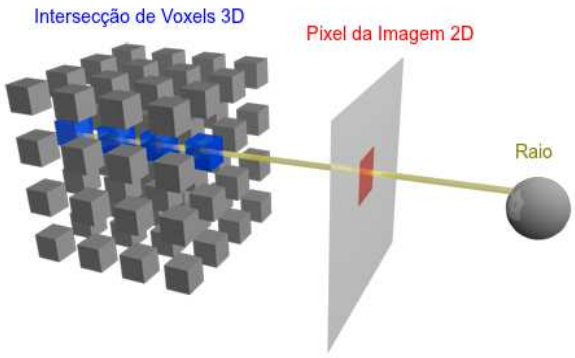


Figura 2.1: Esquema geral da técnica de *raycasting*. Imagem adaptada de [54].

o interior dos volumes de dados, útil em aplicações como a endoscopia virtual. Marsalek et al. [31] e Mensmann et al. [33] implementaram versões da técnica em CUDA, mostrando os diversos recursos presentes nessa arquitetura e comparando com outras implementações baseadas em *shaders*. Fangerau e Krömker [12] elaboraram uma outra versão do *raycasting* em CUDA que atinge um fator de aceleração de 400 vezes em relação à versão sequencial em CPU.

2.2.2 Splatting

A técnica de *splatting* [55] consiste em representar cada voxel de um volume como um *kernel* 3D, o qual possui um peso definido pelo valor discreto do voxel. Esses *kernels* são então processados por uma função denominada *footprint*, que realiza uma projeção (*splat*) para o plano da imagem 2D resultante. O conjunto de todos os *footprints*, cujos pesos são definidos pelos seus respectivos voxels, formam então a imagem final. A principal vantagem desta técnica é que somente os voxels relevantes à imagem são processados, reduzindo a sobrecarga de processamento sobre o volume de dados [34].

Xue e Crawfis [58] apresentaram três técnicas para a implementação do *splatting* em GPU, que são as renderizações por modo imediato, pelo *shader* de vértices e por convolução de pontos. O uso de um *shader* de vértices se mostrou mais rápido para volumes com até 2 milhões de voxels, ao passo que, para volumes com maior quantidade, a convolução de pontos obteve melhores resultados.

Botsch et al. [2] abordaram a questão da utilização do *splatting* de alta qualidade com filtragem EWA (do inglês, *Elliptic Weighted Average*). Eles propuseram um método para aceleração desta técnica em GPU, obtendo uma taxa de 23M de operações de *splat* por segundo e um fator médio de aceleração entre 4 e 6 vezes em relação à CPU, comparado

com métodos relacionados.

2.2.3 Cell-Projection

A técnica de *cell-projection*, proposta por Wilhelms e Van Gelder [57], consiste na renderização de volume utilizando-se projeções de volumes individuais de células no plano da imagem. Uma célula é definida como uma região retangular delimitada por oito pontos amostrais vizinhos (chamados de *cantos da célula*). O processamento é feito célula a célula, em vez do tradicional pixel a pixel, sendo que a técnica ainda realiza um pré-processamento que define o modelo padrão das células.

Esta técnica tem a vantagem de se aproveitar do fator de coerência, o que ocorre quando uma determinada célula é projetada em vários pixels vizinhos. Ela pode também reduzir o *aliasing* das imagens resultantes, que geralmente ocorre em técnicas que se baseiam em amostragem de pontos. Entretanto, há uma certa dependência de um outro algoritmo para determinar a ordem de visibilidade das células.

Marroquim et al. [30] tratam a questão da aceleração desta técnica em GPU com base em um algoritmo denominado Projeção de Tetraedros [50]. O método utiliza uma abordagem que envolve dois *shaders* de fragmentos, atingindo taxas de renderização de até 2×10^6 tetraedros por segundo.

2.2.4 Shear-Warp

Shear-Warp [25] é uma técnica baseada em uma fatoração da matriz de visualização. A ideia desta técnica, esquematizada na Figura 2.2, consiste em realizar um cisalhamento (*shearing)* das fatias que representam um volume de dados, fazendo com que elas fiquem paralelas ao plano da imagem e perpendiculares aos raios que as atravessam. A projeção resultante forma então uma imagem distorcida, devido ao cisalhamento. Finalmente, essa distorção é corrigida na etapa de *warping*, gerando a imagem final.

A principal vantagem desta técnica é o fato de ela tornar a etapa de projeção mais simples, independentemente do ponto de vista dos dados volumétricos.

2.2.5 Conexão de Contornos

Conexão de contornos é um método que realiza o traçado de contornos fechados em cada uma das fatias dos dados e depois faz a conexão entre contornos em fatias adjacentes por algum método de ladrilhamento (em geral, malhas triangulares são empregadas). Após essa etapa, é feita a escolha de parâmetros de visualização, iluminação e renderização, além de passar a superfície ladrilhada para um renderizador de superfícies.

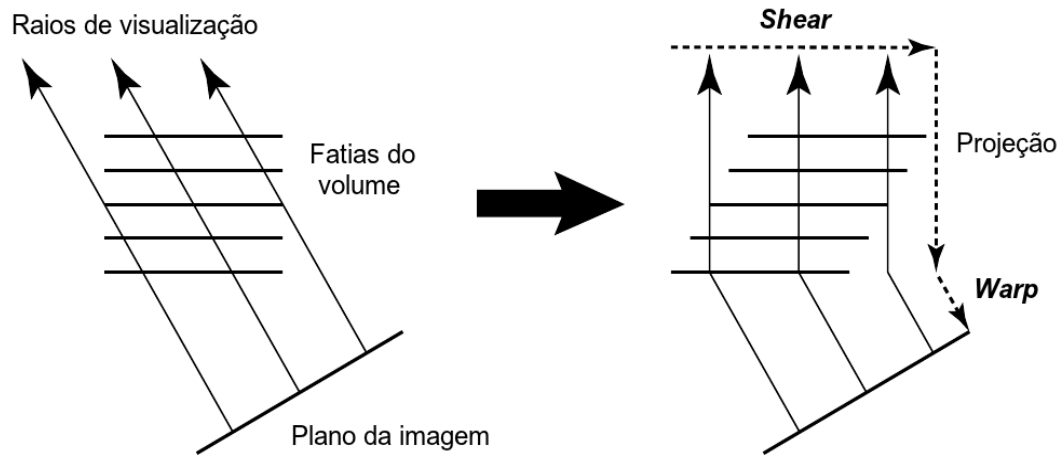


Figura 2.2: Esquema geral da técnica de *shear-warp*. Imagem adaptada de [25].

As vantagens deste método incluem a simplicidade do algoritmo e o grande número de técnicas de renderização de superfície conhecidas [11]. Além disso, a parte do algoritmo que lida com a etapa de renderização propriamente dita é paralelizável, uma vez que nenhum par de fatias adjacentes depende do resultado de um outro par.

2.3 Extração de Isossuperfícies

A extração de isossuperfícies ainda é uma das técnicas mais utilizadas para visualização e exploração de dados volumétricos. Ela envolve a geração de malhas (geralmente triangulares) que representam um volume de dados de maneira aproximada. Na área médica, por exemplo, esse procedimento é comumente usado na visualização de órgãos, tecidos e estruturas anatômicas. A principal técnica de extração de isossuperfícies é a de cubos marchantes [28], que será descrita com mais detalhes na Seção 2.4.

Uma isossuperfície pode ser definida como um conjunto de pontos que possuem o mesmo valor escalar (chamado de *isovalor*) em um volume de dados, isto é, $\{x \in \mathbb{R}^3 : f(x) = h\}$, para algum isovalor $h \in \mathbb{R}$. Normalmente, a extração de características de um volume é definida a partir de intervalos de isovalores, que são utilizados para análise de regiões específicas. Porém, alguns desses isovalores podem ocasionar a geração de artefatos nas malhas, comprometendo a qualidade do resultado final. Para atenuar esse problema, pode ser feito um pós-processamento sobre as malhas renderizadas a partir de técnicas como suavização ou até mesmo reconstrução das malhas [24]. No entanto, essas técnicas

de pós-processamento podem exigir mais tempo e consumo de memória, principalmente quando os volumes são grandes.

Com relação às abordagens para extração de isossuperfícies em GPU, Reck et al. [46] e Buatois et al. [3] propuseram métodos de extração a partir de malhas tetraédricas não-estruturadas. Tatarchuk et al. [52] mostram a implementação de um método híbrido que utiliza as técnicas de cubos marchantes e de tetraedros marchantes, com o auxílio de *shaders* de geometria em GPU. Martin et al. [32] desenvolveram uma técnica para distribuir, de maneira eficiente, toda a carga de trabalho do procedimento de extração de isossuperfícies entre recursos da GPU contidos em um *cluster*.

Pascucci [43] propôs um *pipeline* para o procedimento de extração de isossuperfícies no qual a maior parte das etapas é feita na GPU, deixando para a CPU apenas a tarefa de acessar o volume de dados e encaminhar um conjunto de vértices (correspondentes às células ativas) para a GPU. Nessa proposta, a GPU não tem acesso ao volume de dados, cabendo à CPU transmitir todo o tipo de informação relevante a respeito dos vértices.

2.4 Cubos Marchantes

O algoritmo de cubos marchantes [28] foi desenvolvido com o objetivo de visualizar imagens médicas 3D, provenientes de tomografias e ressonâncias, e que são construídas a partir de várias fatias bidimensionais. Ele utiliza uma abordagem de divisão e conquista na qual um determinado volume de dados é processado por meio de suas células (voxels), geradas a partir de 8 pixels, sendo 4 de duas fatias adjacentes. O resultado final desse algoritmo é uma aproximação, feita através de malhas triangulares, da superfície que representa o volume de dados.

Em cada célula, são determinadas as intersecções entre cada uma das suas 12 arestas e a isossuperfície nela contida. Os valores dos pixels correspondentes a cada vértice da célula são então comparados a um valor h especificado pelo usuário (correspondente ao isovalor associado às isossuperfícies geradas), e esses vértices são posteriormente classificados como "dentro" ou "fora" da isossuperfície. O primeiro caso se aplica quando o valor de um vértice é maior ou igual a h e o segundo quando esse valor é menor que h. Uma vez definido o tipo de intersecção, gera-se uma aproximação da isossuperfície contida na célula. A Figura 2.3 mostra alguns resultados da execução do algoritmo de cubos marchantes para isovalores diferentes.

Como cada um dos 8 vértices das células tem apenas dois estados possíveis, tem-se então um total de $2^8 = 256$ casos de intersecção entre isossuperfície e aresta de célula. A partir dos estados de cada vértice, cria-se um índice de 8 bits, em que cada bit está associado a um determinado vértice e recebe valor 0 ("fora") ou 1 ("dentro"). Uma representação desse procedimento pode ser vista na Figura 2.4.

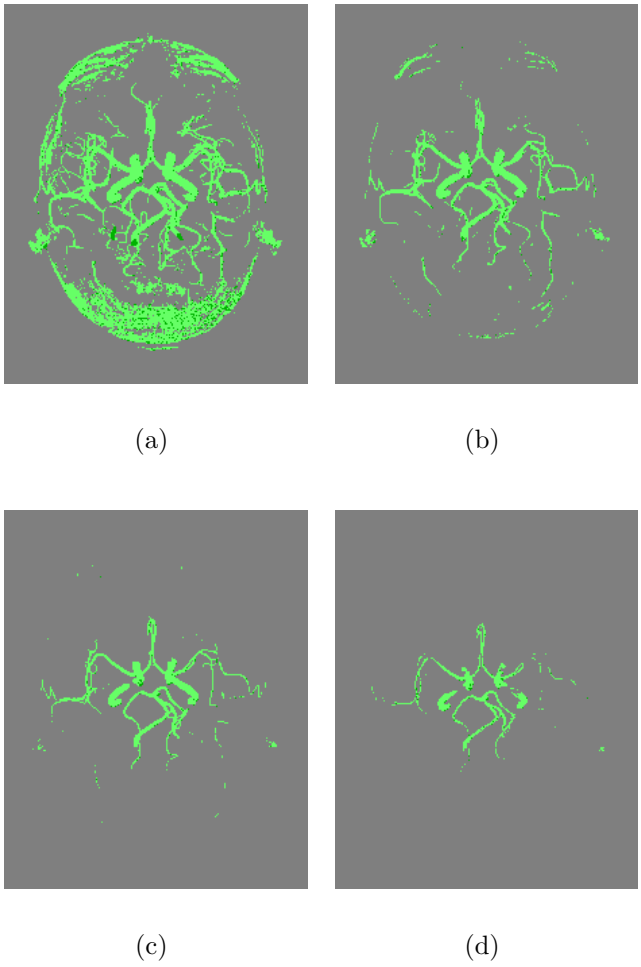


Figura 2.3: Renderização volumétrica de um volume de dados médico (angiografia) utilizando o algoritmo de cubos marchantes com isovalores diferentes: (a) 60; (b) 80; (c) 100; (d) 120.

Com isso, os casos são enumerados com valores entre 0 e 255 e listados em uma tabela contendo todas as intersecções superfície-aresta para cada um deles e que são posteriormente calculadas por meio de interpolação linear. Entretanto, alguns pares desses casos são simétricos ou complementares entre si, o que reduz o número total de casos para 15, como mostrado na Figura 2.5.

A principal vantagem deste algoritmo está no fato de que o processamento de uma célula do volume é independente das demais, o que permite uma paralelização do algoritmo. Por outro lado, ele tem como desvantagem a existência de ambiguidades topológicas existentes entre os casos, o que leva a uma possível geração de artefatos nas isossuperfícies,

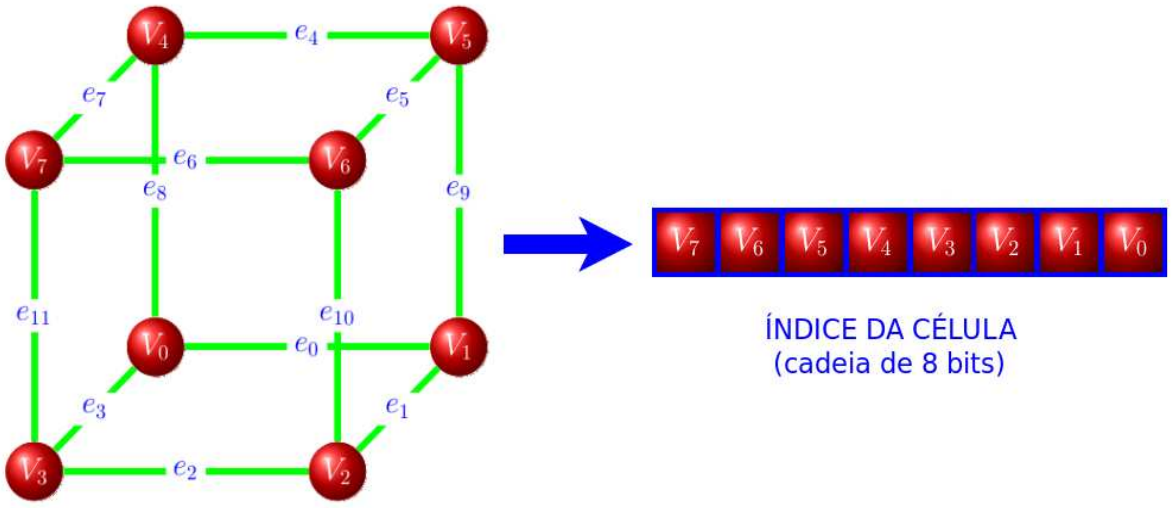


Figura 2.4: Representação de uma célula de um determinado volume de dados em um índice de 8 bits, em que cada bit está associado a um vértice da célula.

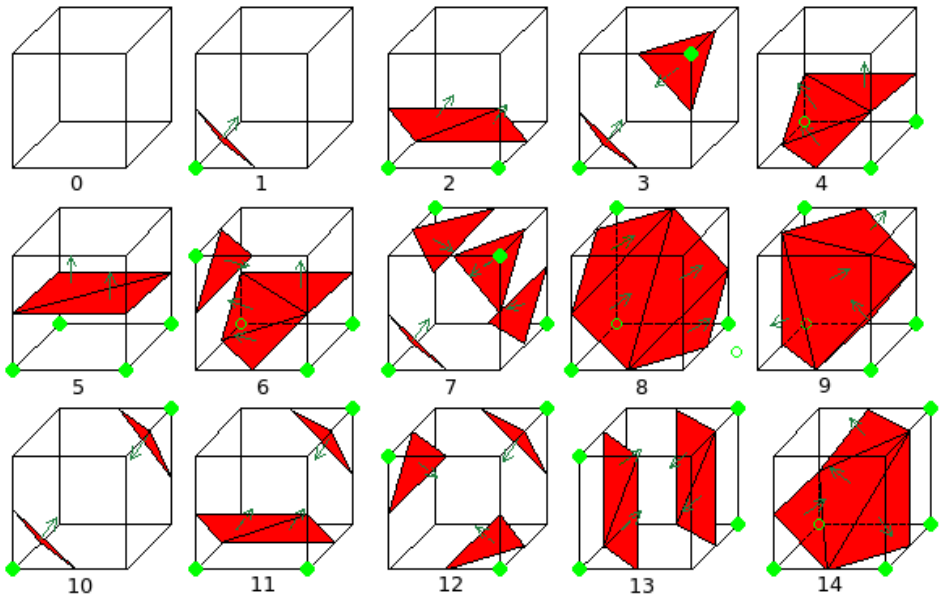


Figura 2.5: Ilustração dos 15 casos básicos do algoritmo de cubos marchantes. Os vértices verdes são aqueles classificados como "dentro" das isossuperfícies, e os demais como "fora" delas. Imagem extraída de [28].

O pseudocódigo do algoritmo pode ser brevemente descrito conforme o Algoritmo 1. Ele é executado em tempo $O(n)$, em que n é o número total de células do volume de dados. No entanto, muitas dessas células são vazias, ocasionando um desperdício no tempo total de processamento.

Algoritmo 1 CubosMarchantes(V, h)

Entrada: Volume de dados V e um isovalor h.
Saída: Lista de vértices a serem renderizados, junto com suas respectivas normais.

1: **para** cada voxel (cubo) v de V **faça**
2: Calcule um índice para v, comparando os 8 valores escalares dos vértices de v com o isovalor h.
3: Utilizando o índice calculado, verifique a lista de arestas contidas em uma tabela pré-calculada.
4: Com base nos valores escalares em cada vértice da aresta, encontre as intersecções superfície-aresta por meio de interpolação linear.
5: Calcule uma normal unitária em cada vértice de v pelo método das diferenças centrais. Interpole a normal para cada vértice do triângulo.
6: Retorne os vértices dos triângulos e as normais dos vértices.
7: **fim para**

Várias pesquisas já foram realizadas com o objetivo de otimizar este algoritmo, tanto no seu desempenho, quanto na qualidade do resultado final. Com relação ao ganho de desempenho, existem várias abordagens que procuram minimizar o tempo desperdiçado no processamento do volume de dados, observando-se apenas as suas células ativas, isto é, aquelas que são interceptadas por uma isossuperfície, reduzindo assim a complexidade do algoritmo. As principais são baseadas no uso de estruturas de dados espaciais, tais como *octree* [56], *interval tree* [6], *pirâmides de histograma* [10], entre outras.

Dentre as propostas de melhora na qualidade dos volumes visualizados, a principal delas é uma extensão do algoritmo de cubos marchantes, denominada tetraedros marchantes [4, 14]. Essa técnica também realiza o processamento do volume de dados por meio de suas células. No entanto, cada célula é dividida em 6 tetraedros (formados a partir de 4 vértices da célula) e então são verificadas as intersecções da isossuperfície contida em cada tetraedro com suas respectivas arestas.

Outra possível maneira é realizar uma extensão da tabela de casos, com o objetivo de eliminar as ambiguidades topológicas existentes entre os 15 casos tradicionais do algoritmo (Figura 2.5). Dürst [9] e Nielson e Hamann [37] mostram a existência dessas ambiguidades quando uma isossuperfície intercepta todas as 4 arestas de pelo menos uma das faces do cubo (como ocorre nos casos 3, 6, 7, 10, 12 e 13 da Figura 2.5), o que permite mais de uma possibilidade na triangulação da isossuperfície. Chernyaev [5] fez um

estudo mais completo acerca dessas ambiguidades e propôs uma nova tabela contendo 33 casos, em vez dos 15 tradicionais, garantindo uma construção topologicamente correta das isossuperfícies.

Mesmo com a eliminação das ambiguidades, esses métodos têm como desvantagem a geração de uma quantidade maior de triângulos para renderizar o volume de dados, o que piora o desempenho do algoritmo, apesar de providenciar uma melhora na qualidade do volume renderizado.

Com o advento das placas gráficas modernas, algumas abordagens que tiram um grande proveito do *hardware* gráfico foram propostas, objetivando acelerar a execução do algoritmo de cubos marchantes e, assim, obter uma taxa maior de renderização. Goetz et al. [13] fizeram uma otimização do algoritmo utilizando um *shader* de vértices, obtendo uma reconstrução interativa de isossuperfícies. Nagel [35] implementou uma versão do algoritmo em CUDA para manipular, de maneira eficiente, volumes de dados na ordem de gigabytes, os quais não podem ser alocados diretamente na memória utilizada. Johansson e Carr [19] propuseram um método que reduz a carga e o consumo de banda na CPU, armazenando os casos do algoritmo no *cache* da GPU, otimizando o procedimento de classificação das células dos volume de dados.

2.5 Aceleração com Estrutura de Dados Espaciais

Um estudo realizado por Newman [36] mostra algumas das técnicas usadas para evitar o processamento de células vazias de um determinado volume de dados. Entre elas, está o uso de estruturas de dados espaciais [48] para organizar todas as células de tal forma que apenas as células ativas sejam processadas.

Uma das classes de estruturas de dados espaciais existentes consiste em representações baseadas em intervalos, que utiliza intervalos de células para realizar o agrupamento das mesmas. A vantagem desse tipo de representação está na sua flexibilidade, podendo ser aplicada não apenas em grades regulares, mas também em grades irregulares, uma vez que ela funciona a partir de um espaço de intervalos, ao invés de utilizar um espaço de malhas.

Os principais métodos dessa classe são baseados em um tipo especial de representação de dados, denominado *span-space* [27], no qual cada célula de um volume é mapeada para um ponto bidimensional, cujas coordenadas x e y correspondem, respectivamente, aos valores mínimo e máximo entre os 8 vértices que compõem a célula. A partir de um dado isovalor h, os pontos do *span-space* que representam as células ativas são aqueles em que $x \leq h$ e $y \geq h$.

Um esquema geral do *span-space* é apresentado na Figura 2.6. A região azul corresponde às células ativas de um determinado volume de dados e as regiões amarelas, às

células que não são renderizadas, pois possuem ou a coordenada $x > h$ (região amarela à direita da região azul) ou $y < h$ (região amarela abaixo da região azul). Nenhuma célula pode ser mapeada para a região vermelha, uma vez que x nunca é maior que y.

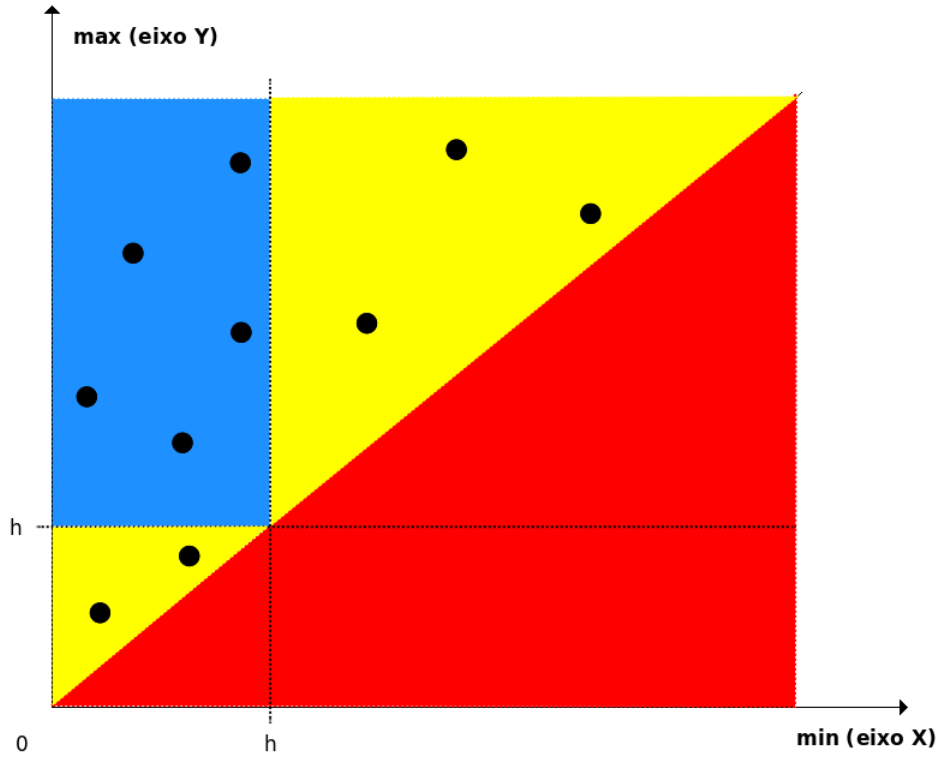


Figura 2.6: Representação do *span-space*, proposto por Livnat [27].

As próximas subseções descrevem algumas das estruturas de dados espaciais e como elas podem ser utilizadas para melhorar o desempenho do algoritmo de cubos marchantes.

2.5.1 *k*-d Tree

A *k*-d *tree* [1] é um caso particular da árvore binária de busca e é utilizada para organizar um determinado conjunto de pontos localizados em um espaço *k*-dimensional. A raiz dessa árvore e os seus nós internos representam um hiperplano que divide esse espaço em duas partes iguais (tomando-se a mediana de uma coordenada específica dentre todos os pontos) em uma determinada direção, que é definida de acordo com a profundidade do nó na árvore. A subárvore esquerda contém todos os pontos localizados à esquerda do hiperplano e a subárvore direita contém os pontos à direita. Cada um dos nós fo-

lhas armazena um único ponto. Um exemplo de construção da k-d *tree* é mostrado na Figura 2.7.

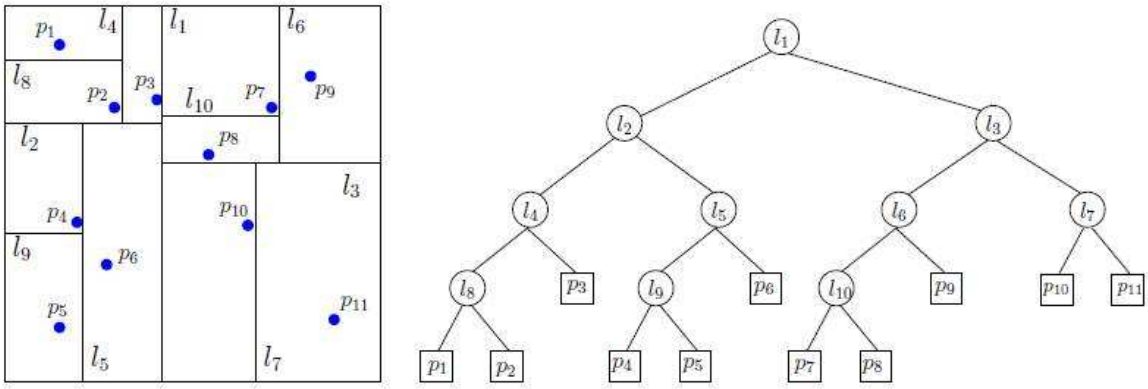


Figura 2.7: Representação de um conjunto de pontos por uma k-d *tree* ($k = 2$). Imagem extraída de [44].

Sendo n o número de pontos a serem organizados, o tempo total de construção é de $O(n \log^2 n)$, caso seja utilizado um algoritmo de ordenação para determinar a mediana dos pontos em cada profundidade. Utilizando-se um algoritmo linear para a determinação da mediana, a complexidade é reduzida para $O(n \log n)$. O tempo de busca é de $O(n^{1-1/k}+p)$, em que p é o número de pontos encontrados na busca. O melhor caso deste algoritmo ocorre quando os pontos estão localizados em um plano bidimensional ($k = 2$).

2.5.2 Interval Tree

A *interval tree* [6] é uma árvore ordenada, utilizada para armazenar intervalos de valores em uma única dimensão. Assim como a k-d *tree*, ela é uma extensão da árvore binária de busca e permite uma busca eficiente de todos os intervalos que sobrepõem um determinado intervalo ou ponto.

Um esquema geral da *interval tree* é mostrado na Figura 2.8. A raiz da árvore armazena um valor que corresponde à mediana das extremidades de todos os intervalos, além de uma lista de intervalos que contêm esse valor (representado por *mid*). A subárvore esquerda (*esq*) guarda os intervalos que estão completamente abaixo da mediana e a subárvore direita (*dir*) guarda os que estão completamente acima da mediana. Então, esse processo é repetido recursivamente para cada subárvore. O tempo total de construção é de $O(n \log n)$.

A busca em uma *interval tree* requer tempo $O(\log n + p)$ (sendo p o número total de intervalos processados), o que a torna mais eficiente do que a *k-d tree*. Por outro lado,

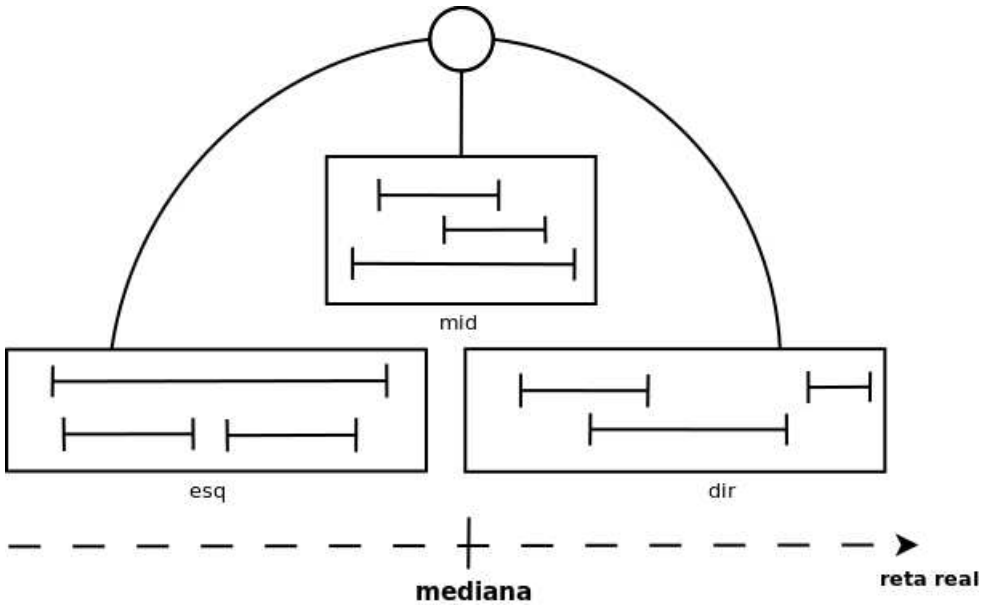


Figura 2.8: Representação geral de uma *interval tree*. Imagem adaptada de [18].

ela exige mais espaço de armazenamento.

2.5.3 Quadtree e Octree

A *quadtree* é uma estrutura de dados em que cada nó interno possui exatamente quatro filhos. Ela é usada para particionar uma determinada região localizada em um plano bidimensional em quatro regiões iguais (também chamadas de quadrantes). Essas regiões são então particionadas em outras quatro subregiões, e assim por diante, até que uma subregião fique vazia, o que caracteriza um nó folha da quadtree. Um exemplo de *quadtree* é representado na Figura 2.9.

A versão tridimensional análoga a essa estrutura é chamada de *octree*, que particiona um espaço tridimensional em oito regiões (ou octantes), conforme ilustrado na Figura 2.10.

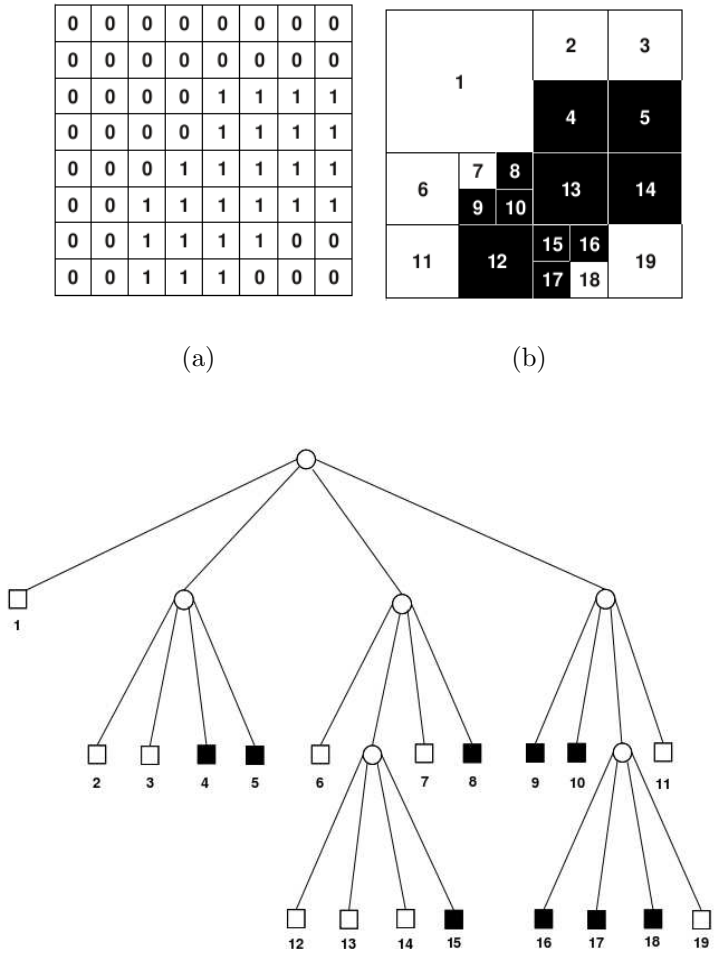


Figura 2.9: Construção de uma *quadtree* a partir de uma imagem 2D. (a) imagem binária; (b) representação das subregiões (quadrantes); (c) *quadtree* correspondente. Imagem adaptada de [48].

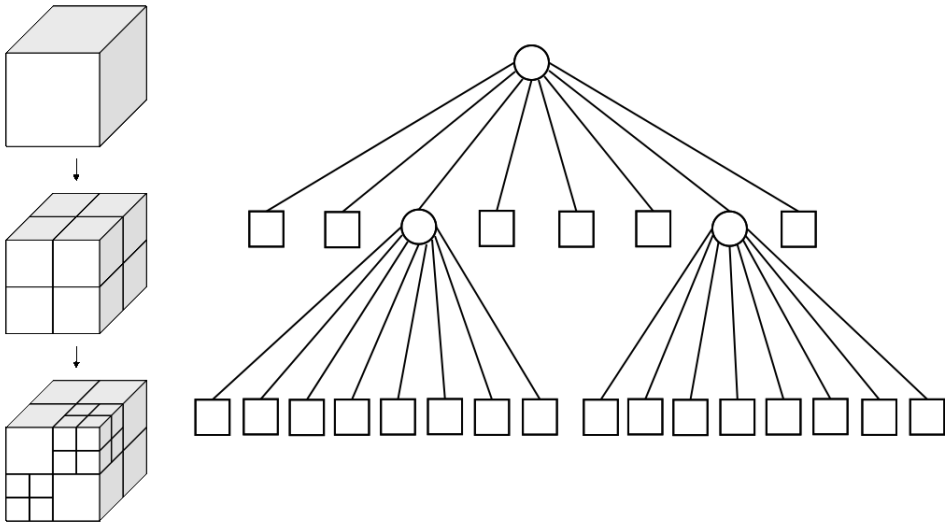


Figura 2.10: Representação de um volume por uma *octree*. Imagem adaptada de [48].

Capítulo 3

Metodologia

A evolução significativa da programabilidade do *hardware* gráfico permitiu que vários procedimentos pudessem ser acelerados pela GPU, tirando proveito de sua arquitetura paralela. No entanto, o gargalo da aceleração está no barramento de comunicação entre CPU e GPU, o que indica que nem sempre a melhor solução é transferir todas as tarefas para a GPU. Dessa forma, para maximizar o desempenho deste procedimento, é necessário um bom planejamento do *pipeline* gráfico, selecionando as tarefas que podem ser executadas na CPU e aquelas que podem ser transferidas para a GPU, bem como uma boa utilização da hierarquia de memória.

A metodologia proposta neste trabalho está restrita à técnica de cubos marchantes, ao invés de generalizar o processo de extração de isossuperfícies. Um esquema geral é mostrado na Figura 3.1, composto por 6 etapas. Uma versão preliminar da metodologia foi descrita em [7]. Na Etapa 1, a CPU faz a leitura de um volume de dados de dimensões $N_x \times N_y \times N_z$, que é então alocado tanto na memória principal (RAM, utilizada pela CPU) quanto na memória de vídeo (VRAM, utilizada pela GPU). Para uma melhor manipulação dos dados, o volume é estruturado em forma de uma grade regular de tamanho $N_x - 1 \times N_y - 1 \times N_z - 1$, em que cada posição corresponde a uma célula em forma de cubo. Com exceção do caso da *octree*, aloca-se também um espaço na memória principal para a criação do *span-space* associado ao volume de dados.

Depois disso, uma das estruturas de dados espaciais descritas na Seção 2.5 é construída a partir do volume de dados e armazenada somente na memória principal (Etapa 2). Então, o volume de dados é liberado dessa memória, mas ele permanece alocado na memória de vídeo.

Uma vez finalizado o estágio de pré-processamento, o algoritmo de cubos marchantes é iniciado (Etapa 3). A partir de um isovalor h especificado pelo usuário, a CPU realiza uma busca na estrutura de dados, percorrendo somente os nós que correspondem às células ativas do volume, gerando uma lista dessas células que é então transferida para a GPU

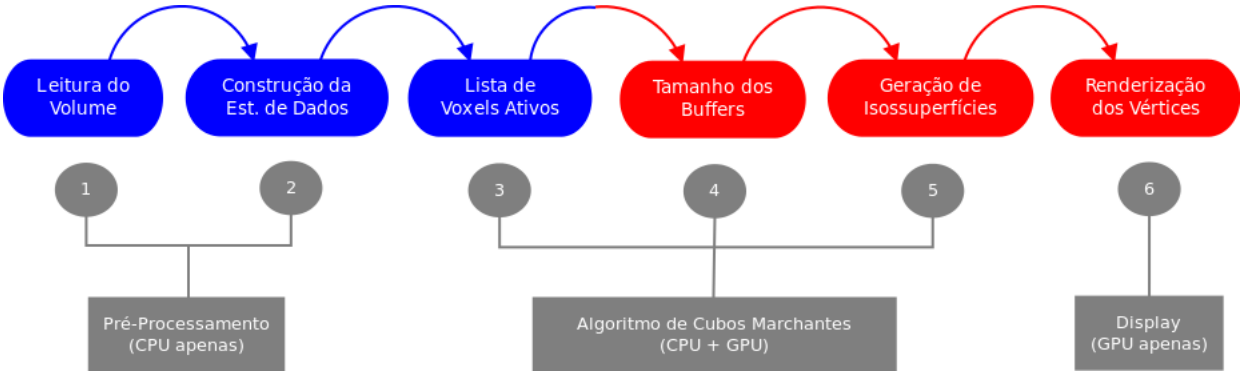


Figura 3.1: Abordagem utilizada para a aceleração do algoritmo de cubos marchantes. As etapas 1 e 2 são responsáveis pelo estágio de pré-processamento; as etapas 3 a 5 correspondem à execução do algoritmo de cubos marchantes e a etapa 6 faz a exibição dos vértices na tela. As caixas azuis representam as ações executadas na CPU e as vermelhas, aquelas na GPU.

pelo barramento de comunicação.

Com a lista de células ativas e o isovalor h, a GPU continua a executar o algoritmo de cubos marchantes. Cada célula é classificada em um dos 15 casos (mostrados na Figura 2.5) comparando o valor de h ao dos 8 vértices da célula. Depois que esse procedimento é realizado em todas as células ativas, o número total de vértices a serem renderizados é calculado, que irá definir o tamanho exato de espaço da memória de vídeo utilizado para alocar os *buffers* de vértices: um para armazenar as posições dos vértices e outro para as suas respectivas normais (Etapa 4).

Em seguida, a GPU novamente faz uma varredura pela lista de células ativas para gerar os triângulos que compõem as isossuperfícies (Etapa 5). Para cada célula ativa da lista, ela calcula as intersecções da isossuperfície com as 12 arestas da célula através da interpolação dos vértices do volume de dados. As normais dos vértices são posteriormente obtidas a partir dos triângulos gerados com base no índice e nas intersecções calculadas. Para cada triângulo, são criados dois vetores, efetua-se o produto vetorial entre eles e a normal resultante é distribuída para cada vértice do triângulo. Ao final desses procedimentos, a GPU obtém a lista de vértices e normais relativas às isossuperfícies, escrevendo-as nos seus respectivos *buffers*. Por fim, o volume de dados é renderizado a partir desses *buffers* (Etapa 6).

Todos os procedimentos executados pela GPU são paralelizados, uma vez que os resultados obtidos a partir de uma célula são independentes daqueles das demais células. Entretanto, a aceleração do algoritmo de cubos marchantes depende da maneira que essa paralelização ocorre. Quando uma tarefa é atribuída à GPU, ela cria uma quantidade

específica de blocos[1], que contêm o mesmo número de *threads*, responsáveis por executar uma parte dessa tarefa.

O número de blocos depende diretamente do número de células ativas e do número de *threads* por bloco, também chamado de *tamanho do bloco*. O tamanho do bloco é escolhido de uma maneira que ele não seja nem muito baixo, atribuindo uma alta quantidade de trabalho para todas as *threads* e não maximizando o desempenho no geral, e nem muito alto, causando uma sobrecarga de inicialização e encerramento de *threads*.

A seguir, os algoritmos utilizados para a construção e busca relativos às estruturas de dados espaciais em CPU são apresentados, além de como a técnica de cubos marchantes é acelerada na GPU.

3.1 Estruturas de Dados Espaciais

Conforme descrito na Seção 2.5, as estruturas de dados espaciais organizam as células de um determinado volume em árvores de busca, de maneira que o processamento de células vazias seja evitado. Além disso, é fundamental que os procedimentos de construção dessas estruturas resultem em árvores bem balanceadas, o que define a complexidade de pior caso na etapa de busca, obtendo assim parte da aceleração desejada (feita na CPU) no algoritmo de cubos marchantes.

Nesta seção, são descritos os pseudocódigos dos algoritmos das estruturas de dados espaciais utilizadas neste trabalho. Os algoritmos referentes à k-d tree e à *interval tree* foram adaptados das versões descritas em [18]. São também propostos outros algoritmos (*quadtree* e *octree*) com o intuito de avaliar os desempenhos nas etapas de construção e busca, comparando com algoritmos já existentes.

3.1.1 k-d Tree

A k-d *tree* é uma das estruturas de dados espaciais que se aproveitam da abordagem de *span-space* para a representação das células dos dados volumétricos. Neste caso, a ideia de utilizar esta representação é construir a k-d *tree* com respeito a pontos localizados em um plano bidimensional em vez de utilizar o espaço do volume, obtendo-se assim um ganho no desempenho do algoritmo de busca.

Na etapa de pré-processamento (Figura 3.1), o volume de dados é mapeado para um *span-space* antes da construção da k-d *tree*, uma vez que as buscas são mais rápidas quando são utilizados pontos localizados em um plano 2D do que em um espaço 3D. Além disso, cada nó armazena um ponto do *span-space*, em vez de armazenar os pontos apenas nas

[1] Conceito extraído do Modelo de Programação CUDA [39]. No modelo do OpenCL [41], o conceito é conhecido como *grupo de trabalho* (do inglês, *work group*).

folhas e as retas que dividem o espaço nos nós internos, o que exigiria um pouco mais de espaço de armazenamento.

Durante a etapa de construção (Algoritmo 2), o algoritmo começa dividindo o conjunto de pontos do *span-space* com relação à coordenada x (correspondente ao menor valor da célula associada). No nível seguinte, as divisões são feitas em relação à coordenada y (correspondente ao maior valor da célula associada), depois novamente em relação a x, e assim por diante.

Algoritmo 2 ConstroiKdTree(S, $prof$)

Entrada: Conjunto S de pontos do *span-space* associado ao volume de dados e profundidade $prof$ da recursão

Saída: Ponteiro para a raiz de uma k-d tree que armazena todos os pontos de S

```
1: se S for vazio então
2:     retorne nada
3: senão
4:     se prof for par então
5:         Encontre a mediana dos elementos de S em relação à coordenada x
6:     senão
7:         Encontre a mediana dos elementos de S em relação à coordenada y
8:     fim se
9: fim se
10: Crie um nó R (correspondente à raiz da árvore atual) contendo a mediana
11: Inicialize S_esq e S_dir como conjuntos vazios de pontos
12: Insira os elementos de S à esquerda da mediana em S_esq e aqueles à direita em S_dir
    // Criando os filhos esquerdo e direito (R_esq e R_dir) por recursão
13: R_esq ← ConstroiKdTree(S_esq, prof + 1)
14: R_dir ← ConstroiKdTree(S_dir, prof + 1)
15: retorne R
```

Nesta versão do algoritmo de construção, a determinação da mediana dos pontos (linhas 5 e 7 do Algoritmo 2) é feita a partir de um algoritmo de seleção descrito em [45], que encontra a mediana de um vetor de n elementos em tempo $O(n)$. Além disso, ele transfere a mediana para a posicão intermediária do vetor, colocando também os elementos que são menores que a mediana à esquerda dela e os maiores à direita (linha 12 do Algoritmo 2), eliminando a necessidade de ordenar o vetor em cada nível, o que implicaria um gasto de tempo muito maior na construção da árvore, além de permitir subdivisões imediatas do conjunto de pontos.

O pseudocódigo do algoritmo de busca está descrito no Algoritmo 3. Quando uma busca é realizada na k-d tree, a partir de um dado isovalor h, ela percorre somente os nós

correspondentes às células ativas do volume de dados. Desta forma, a busca é feita em tempo $O(\sqrt{n}+p)$, onde n é o número total de células do volume de dados e p é o número de células ativas processadas.

Nas linhas 8 e 13 do Algoritmo 3, é suficiente a verificação de apenas uma das coordenadas, uma vez que a busca está limitada ao canto superior esquerdo do *span-space* (ilustrado na Figura 2.6). Quando a subdivisão é feita ao longo do eixo X (valores mínimos de célula), o limite inferior é implicitamente igual a zero. Por outro lado, quando ela é feita com relação ao eixo Y (valores máximos de célula), o limite superior é implicitamente infinito.

Algoritmo 3 BuscaKdTree(N, h, $prof$, L)

Entrada: Um nó N da k-d tree, um isovalor h, profundidade $prof$ da recursão e lista L de células ativas

1: **se** N for NULL **então**
2: **retorne**
3: **fim se**

// Verifica se o isovalor h está dentro dos valores extremos da célula associada a N
4: **se** $N_x \leq h \leq N_y$ **então**
5: Insira em L a célula associada a N
6: **fim se**

7: **se** Se $prof$ for par **então**
8: **se** $h \geq N_x$ **então**
9: BuscaKdTree(N_{dir}, h, $prof+1$, L)
10: **fim se**
11: BuscaKdTree(N_{esq}, h, $prof+1$, L) *// Limite inferior é implicitamente zero*
12: **senão**
13: **se** $h \leq N_y$ **então**
14: BuscaKdTree(N_{esq}, h, $prof+1$, L)
15: **fim se**
16: BuscaKdTree(N_{dir}, h, $prof+1$, L) *// Limite superior é implicitamente infinito*
17: **fim se**

3.1.2 Interval Tree

No escopo do algoritmo de cubos marchantes, o *span-space* é bastante adequado para a construção da *interval tree*. Nesse caso, os intervalos são representados pelas células do volume de dados e as extremidades desses intervalos correspondem aos valores mínimo e máximo da célula, que nada mais são do que as coordenadas do *span-space* associadas à célula.

O Algoritmo 4 descreve a etapa de construção da *interval tree*. Inicialmente, a mediana de todas as coordenadas dos pontos (correspondentes às extremidades dos intervalos) é calculada, que servirá como base para a divisão do conjunto de pontos. A partir do conjunto inicial, são definidos 3 subconjuntos: o primeiro é formado por intervalos que estão completamente abaixo da mediana; o segundo, pelos intervalos que estão completamente acima da mediana; e o terceiro, por aqueles que contêm a mediana.

Algoritmo 4 ConstroiIntervalTree(S)

Entrada: Conjunto S de pontos do *span-space* associado ao volume de dados
Saída: Ponteiro para a raiz de uma *interval tree* que armazena todos os pontos de S

```
1: se S for vazio então
2:     retorne nada
3: senão
4:     mid ← mediana de todas as coordenadas x e y dos pontos de S
5:     Inicialize S_esq, S_midMin, S_midMax e S_dir como conjuntos vazios de pontos
6:     para todo ponto p de S faça
7:         se a coordenada y de p for menor que mid então
8:             Insira p em S_esq
9:         senão se a coordenada x de p for maior que mid então
10:            Insira p em S_dir
11:        senão
12:            Insira p em S_midMin e em S_midMax
13:        fim se
14:    fim para
15:    Ordene os pontos de S_midMin em ordem crescente com relação à coordenada x
16:    Ordene os pontos de S_midMax em ordem decrescente com relação à coordenada y
17:    Crie um nó R (correspondente à raiz da árvore atual) contendo a mediana mid e
       os conjuntos S_midMin e S_midMax
       // Criando os filhos esquerdo e direito (R_esq e R_dir) por recursão
18:    R_esq ← ConstroiIntervalTree(S_esq)
19:    R_dir ← ConstroiIntervalTree(S_dir)
20:    retorne R
21: fim se
```

Os dois primeiros conjuntos são denotados por S_{esq} e S_{dir}, respectivamente. Seja o terceiro conjunto denotado por S_{mid}. Nele é feito um tratamento com o objetivo de otimizar a etapa de busca. Primeiro, S_{mid} é duplicado, criando-se os conjuntos S_{midMin} e S_{midMax}. Em seguida, S_{midMin} é ordenado em ordem crescente com relação à coordenada x (extremidade inferior do intervalo) e S_{midMax} em ordem decrescente com relação à coordenada

y (extremidade superior do intervalo). Depois disso, um nó é criado para armazenar o valor da mediana que divide o conjunto inicial de intervalos, além dos conjuntos S_{midMin} e S_{midMax}. Por sua vez, S_{esq} e S_{dir} constituirão as subárvores esquerda e direita da *interval tree*, que são criadas recursivamente.

No algoritmo de busca, cujo pseudocódigo está descrito no Algoritmo 5, compara-se um deteterminado isovalor h, passado como entrada, com a mediana armazenada em um certo nó da *interval tree*. Se h for menor que a mediana, serão inseridas na lista de células ativas aquelas correspondentes aos pontos que têm a coordenada x menor que a mediana. Como S_{midMin} está ordenado pela coordenada x, é então feita uma varredura sequencial nesse conjunto, até que a coordenada x de um ponto desse conjunto seja maior que a mediana. Caso contrário, serão inseridas na lista as células associadas aos pontos em que a coordenada y é maior que a mediana. Então, o procedimento análogo é feito no conjunto S_{midMax}.

É importante salientar que a construção da *interval tree* pode ser feita sem a necessidade de duplicar o conjunto S_{mid} (além das ordenações de cada cópia), o que não interfere no resultado final da visualização do volume de dados. No entanto, o algoritmo de busca nesse caso torna-se um pouco ineficiente, pois será necessário fazer uma varredura por todos os pontos de S_{mid} e verificando se o isovalor h está compreendido entre as coordenadas de cada um deles.

Algoritmo 5 BuscaIntervalTree(N, h, L)

Entrada: Um nó N da *interval tree*, um isovalor h e lista L de células ativas

```
1: se N for NULL então
2:     retorne
3: fim se
4: se h < N_mediana então
5:     para todo ponto p em N_midMin tal que a coordenada x de p seja menor ou igual
       a h faça
6:         Insira em L a célula associada a p
7:     fim para
8:     BuscaIntervalTree(N_esq, h, L)
9: senão
10:    para todo ponto p em N_midMax tal que a coordenada y de p seja maior ou igual
       a h faça
11:        Insira em L a célula associada a p
12:    fim para
13:    BuscaIntervalTree(N_dir, h, L)
14: fim se
```

3.1.3 Quadtree

Uma vez que o *span-space* é um espaço 2D, ele pode ser representado por uma *quadtree*. Seja b o número de bits utilizados para armazenar os valores do volume de dados, o que significa que existem 2^b valores possíveis (variando entre 0 e $2^b - 1$) para um vértice de célula. Assim, o *span-space* é uma região definida por uma matriz de tamanho $2^b \times 2^b$, sendo que cada posição dessa matriz corresponde a um ponto do *span-space*. A construção da *quadtree* é então feita com base nessa região, em que cada nó representa uma posição da matriz.

Entretanto, mais de uma célula pode ser mapeada para um mesmo ponto no *span-space*. Para lidar com esse problema, cada posição da matriz armazena um ponteiro para uma lista das células que foram mapeadas para o ponto correspondente. Desta forma, cada nó folha da *quadtree* armazenará uma lista associada a um único ponto do *span-space* e os nós internos guardarão somente os valores mínimo e máximo de todos os pontos contidos na subregião correspondente, além de uma lista vazia. As buscas são realizadas da forma usual, percorrendo somente os nós associados às células ativas, com base em um isovalor h passado como entrada e verificando se ele está contido nos intervalos compreendidos entre os valores mínimo e máximo de cada nó interno da *quadtree*.

Os pseudocódigos dos algoritmos de construção e de busca estão descritos nos algoritmos 6 e 7, respectivamente.

3.1.4 Octree

No caso da *octree*, a árvore é construída diretamente a partir do volume de dados. Entretanto, nos casos em que pelo menos uma das dimensões não é uma potência de 2, as subregiões têm tamanhos diferentes, uma vez que se está lidando com particionamento de regiões compostas por células inteiras.

Os algoritmos de construção (Algoritmo 8) e de busca (Algoritmo 9) são semelhantes aos da *quadtree*. Na fase de construção da *octree*, os nós folhas armazenam uma determinada célula da grade associada ao volume de dados e os nós internos guardam os valores mínimo e máximo dentre todos os voxels da subregião associada.

3.2 Cubos Marchantes em GPU

Conforme explicado no início deste capítulo, a CPU inicia o algoritmo de cubos marchantes realizando a busca em uma determinada estrutura de dados associada a um volume, gerando uma lista das células ativas que são posteriormente renderizadas, o que depende diretamente do isovalor passado como entrada para o algoritmo. Essa lista é então transferida para a GPU que, por sua vez, fará um processamento paralelo das

Algoritmo 6 ConstroiQuadtree(M)

Entrada: Matriz M de listas de pontos do *span-space* mapeados para cada uma das posições $M[i, j]$

Saída: Ponteiro para a raiz de uma *quadtree* que armazena todas as listas de M

se M for vazia **então**
    retorne nada
senão se M contiver apenas um elemento **então**
    retorne um nó armazenando a lista de pontos do único elemento de L
senão
    $min \leftarrow$ menor valor dentre as coordenadas x dos pontos de cada lista de M
    $max \leftarrow$ maior valor dentre as coordenadas y dos pontos de cada lista de M
    // *Condição de grade vazia*
    se $max = 0$ **então**
        retorne nada
    senão
        para i de 1 a 4 **faça**
            $M_i \leftarrow$ região correspondente a um quadrante de M
            $N_i \leftarrow$ ConstroiQuadtree(M_i) // *onde* N_i *é um nó filho*
        fim para
    fim se
    retorne um nó armazenando min, max e os 4 filhos N_i (com $1 \leq i \leq 4$)
fim se

Algoritmo 7 BuscaQuadtree(N, h, L)

Entrada: Um nó N da quadtree, um isovalor h e a lista L de células ativas

se N for NULL **então**
    retorne
senão se a lista de pontos de N não for vazia **então**
    para todo ponto p da lista de pontos de N **faça**
        Insira em L a célula associada a p
    fim para
senão se $N_{min} \leq h \leq N_{max}$ **então**
    para cada um dos 4 filhos N_i de N, com $1 \leq i \leq 4$ **faça**
        BuscaQuadtree(N_i, h, L)
    fim para
fim se
retorne

Algoritmo 8 ConstroiOctree(G)

Entrada: Grade G contendo os voxels do volume de dados associado
Saída: Ponteiro para a raiz de uma *octree* que armazena todos os voxels de G

se G for vazia **então**
    retorne nada
senão se G contiver apenas um voxel **então**
    retorne um nó armazenando o voxel e os seus valores mínimo e máximo
senão
    $min \leftarrow$ menor valor dentre os valores mínimos de cada voxel de G
    $max \leftarrow$ maior valor dentre os valores máximos de cada voxel de G
    // Condição de grade vazia
    se $max = 0$ **então**
        retorne nada
    senão
        para i de 1 a 8 **faça**
            $G_i \leftarrow$ região correspondente a um octante de G
            $N_i \leftarrow$ ConstroiOctree(G_i) *// onde N_i é um nó filho*
        fim para
    fim se
    retorne um nó armazenando min, max e os 8 filhos N_i (com $1 \leq i \leq 8$)
fim se

Algoritmo 9 BuscaOctree(N, h, L)

Entrada: Um nó N da octree, um isovalor h e a lista L de células ativas

se N for NULL **então**
    retorne
senão se N não for um nó folha **então**
    Insira em L a célula associada a N
senão se $N_{min} \leq h \leq N_{max}$ **então**
    para cada um dos 8 filhos N_i de N, com $1 \leq i \leq 8$ **faça**
        BuscaOctree(N_i, h, L)
    fim para
fim se
retorne

células que compõem a lista, a partir de um programa específico denominado *kernel* de GPU.

Neste trabalho, o *kernel* de GPU foi implementado utilizando a arquitetura CUDA. Além dos algoritmos que realizam as etapas da técnica de cubos marchantes, alguns dos recursos inerentes à GPU foram utilizados para que esses algoritmos fossem executados mais rapidamente, implicando uma melhora na taxa de renderização do volume de dados. Dentre esses recursos estão o uso de texturas para guardar as tabelas utilizadas pelo algoritmo de cubos marchantes e o próprio volume de dados, e o uso de memória compartilhada, que permite o compartilhamento de dados entre *threads* contidas em um mesmo bloco, além de ser mais rápida do que a memória global de vídeo [39].

Os pseudocódigos dos algoritmos a seguir foram adaptados da versão disponível em [38]. Antes da execução desses algoritmos, são determinados os respectivos números de blocos (bem como os seus tamanhos) que serão utilizados para auxiliar na paralelização dos procedimentos neles executados.

O Algoritmo 10 representa o primeiro passo da execução da técnica de cubos marchantes em GPU (Etapa 4 da Figura 3.1). O objetivo deste algoritmo é calcular a quantidade exata de memória de vídeo que será utilizada para armazenar os vértices e as normais referentes ao volume de dados. Uma vez obtida essa quantidade, a GPU então aloca o espaço apropriado nos *buffers* de vértice, contidos na memória de vídeo, associados às posições dos vértices a serem renderizados e às suas respectivas normais.

Feito isso, o Algoritmo 11 é iniciado. Um ponto importante a ser observado neste algoritmo é a repetição da etapa de cálculo do índice das células (linha 4 dos Algoritmos 10 e 11), o que na GPU é mais rápido do que armazenar os eventuais resultados provenientes do Algoritmo 10 em um vetor e depois recuperar os respectivos valores no algoritmo subsequente. Em seguida, os *buffers* de vértices e de normais (P e N, respectivamente) são preenchidos e o volume de dados é visualizado, encerrando as etapas da técnica de cubos marchantes.

Algoritmo 10 CalculaNumVertices(V, N, h)

Entrada: Lista V de voxels ativos, lista N do número de vértices que compõem as isossuperfícies de cada voxel de V e isovalor h

1: **para** cada voxel v de V em paralelo **faça**
2: Determine as posições dos 8 vértices de v.
3: Para cada posição, faça uma busca na textura do volume de dados para determinar os respectivos valores escalares dos vértices.
4: Calcule o índice de v, comparando os seus valores escalares com o isovalor h.
5: Determine o número de vértices da isossuperfície contida em v a partir do índice calculado (fazendo uma busca na textura da tabela de casos) e adicione esse valor em N.
6: **fim para**

Algoritmo 11 GeraIsossuperficies(P, N, V, h)

Entrada: *Buffers* P e N de posições e normais dos vértices a serem renderizados, lista V de voxels ativos e isovalor h

1: **para** cada voxel v de V em paralelo **faça**
2: Determine as posições dos 8 vértices de v.
3: Para cada posição, faça uma busca na textura do volume de dados para determinar os respectivos valores escalares dos vértices.
4: Calcule o índice de v, comparando os seus valores escalares com o isovalor h.
5: Calcule, por interpolação, as intersecções em cada aresta de v utilizando memória compartilhada e armazene em um vetor I
6: Determine o número de vértices da isossuperfície contida em v a partir do índice calculado (fazendo uma busca na textura da tabela de casos).
7: **para** cada triângulo t de v **faça**
8: Encontre as arestas de v interceptadas pelos vértices de t fazendo uma busca na textura da tabela de intersecções.
9: Calcule a normal de t (que será a mesma para os 3 vértices de t).
10: A partir de I, adicione as posições dos vértices de t em P.
11: Adicione as normais de t em N.
12: **fim para**
13: **fim para**

Capítulo 4

Resultados Experimentais

Os testes foram executados em um processador AMD Phenom II X6 1090T 3.2 GHz com 4 GB de RAM e uma placa gráfica NVIDIA GeForce GTS 450 com 1 GB de VRAM, utilizando a linguagem de programação C e as API's OpenGL e CUDA.

Os experimentos foram realizados com volumes de 8 bits (disponíveis em [53]), em que cada valor escalar varia entre 0 e 255. A Tabela 4.1 mostra uma lista dos volumes que foram utilizados nos testes, além de suas respectivas dimensões (em voxels), isovalores e o número de triângulos renderizados na tela (o que depende do isovalor). Os resultados das renderizações de cada um desses volumes, realizadas a partir da aplicação descrita neste trabalho, são exibidos na Figura 4.1.

Nome do Volume	**Dimensões**	**Isovalor**	**Nº de Triângulos**
Combustível	$64 \times 64 \times 64$	10	11534
Átomo de Hidrogênio	$128 \times 128 \times 128$	20	47864
Angiografia	$256 \times 320 \times 128$	80	84974
Ventrículos	$256 \times 256 \times 124$	120	167214
Motor	$256 \times 256 \times 128$	155	207592

Tabela 4.1: Lista de volumes de dados com seus respectivos tamanhos, isovalores e número de triângulos gerados.

A Tabela 4.2 mostra a taxa média de quadros por segundo (FPS, do inglês *Frames Per Second*) da execução do algoritmo de cubos marchantes em CPU e em GPU para todas as estruturas de dados descritas na Seção 2.5. Esses resultados não consideram o tempo gasto na leitura do volume e na construção da estrutura de dados (o que não é realizado na versão força bruta do algoritmo), considerando somente os eventos que ocorrem entre a busca na estrutura de dados e a visualização do volume na tela. Além disso, nenhum

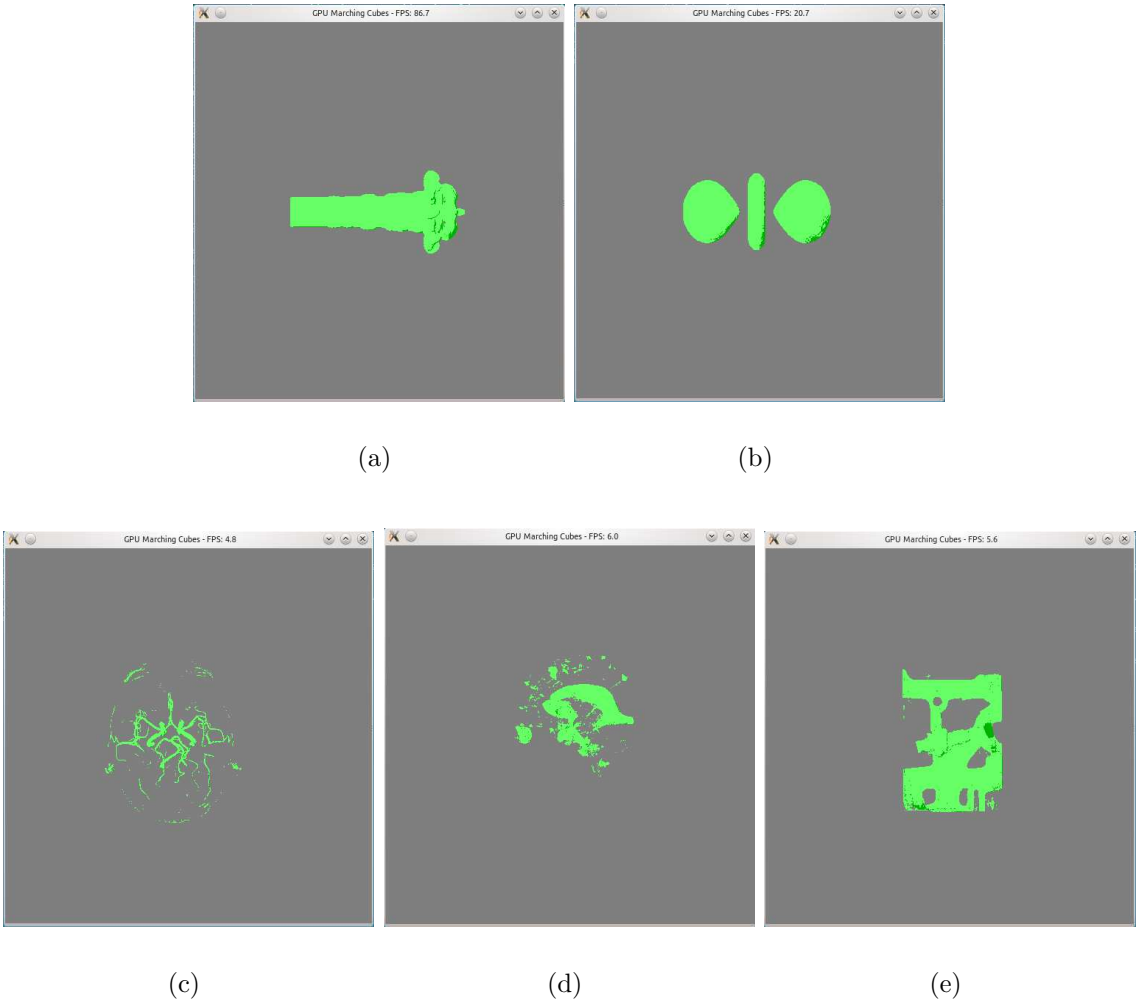


Figura 4.1: Resultados das renderizações geradas pela aplicação de cada volume de dados, com os isovalores listados na Tabela 4.1. (a) combustível; (b) átomo de hidrogênio; (c) angiografia; (d) ventrículos; (e) motor.

tipo de animação foi feito sobre os volumes renderizados, o que ocasionaria discrepâncias no cálculo do FPS.

Como pode ser observado, a *interval tree* propiciou os melhores resultados (destacados em negrito na Tabela 4.2), não apenas entre todas as estruturas de dados, mas também no fator de aceleração comparado à versão força bruta do algoritmo de cubos marchantes em CPU, obtendo uma aceleração de 16.4 vezes, no caso do volume de dados da angiografia. Isso foi esperado porque a *interval tree* possui, assintoticamente, um tempo melhor de busca em relação às outras estruturas. Na média, o fator de aceleração entre CPU e GPU

CPU					
Volume	**Força Bruta**	**k-d Tree**	**Interval Tree**	**Quadtree**	**Octree**
Combustível	82.4	106.0 (1.3)	**111.5 (1.4)**	107.5 (1.3)	104.7 (1.3)
Hidrogênio	14.8	25.5 (1.7)	**32.1 (2.2)**	30.2 (2.0)	28.8 (1.9)
Angiografia	3.6	8.2 (2.3)	**11.5 (3.2)**	10.1 (2.8)	9.2 (2.6)
Ventrículos	4.0	7.7 (1.9)	**10.8 (2.7)**	10.1 (2.5)	9.4 (2.3)
Motor	3.7	6.4 (1.7)	**8.9 (2.4)**	8.4 (2.3)	7.7 (2.1)
GPU					
Volume	**Força Bruta**	**k-d Tree**	**Interval Tree**	**Quadtree**	**Octree**
Combustível	94.6 (1.1)	149.5 (1.8)	**178.9 (2.2)**	170.4 (2.1)	146.9 (1.8)
Hidrogênio	25.3 (1.7)	67.6 (4.6)	**88.5 (6.0)**	87.3 (5.9)	49.7 (3.4)
Angiografia	6.1 (1.7)	39.2 (10.9)	**59.0 (16.4)**	51.0 (14.2)	28.2 (7.8)
Ventrículos	6.8 (1.7)	20.7 (5.2)	**32.4 (8.1)**	27.5 (6.9)	19.7 (4.9)
Motor	6.4 (1.7)	17.5 (4.7)	**28.5 (7.7)**	22.1 (6.0)	16.9 (4.6)

Tabela 4.2: Taxa média de quadros por segundo da execução do algoritmo de cubos marchantes em CPU e em GPU, com diferentes estruturas de dados.

para a mesma estrutura de dados ficou entre 2.0 e 6.0,

Vale lembrar que as normais dos vértices dos volumes renderizados não são pré-calculadas, o que diminuiria a quantidade de cálculos feitos pela GPU e ocasionando um ganho de desempenho maior. Desta forma, pode-se comparar os resultados da Tabela 4.2) com aqueles obtidos por Johansson [18], no caso em que as normais são calculadas na GPU durante a execução do algoritmo de cubos marchantes. Johansson obteve um ganho máximo de 4.6 vezes para a *interval tree*, bem inferior aos 16.4 obtidos na abordagem desenvolvida neste trabalho. No caso da *k*-d *tree*, foi possível obter um ganho máximo de 10.8 vezes, contra 4.4 de Johansson.

Outro fato importante a ser observado é o desempenho da *quadtree*, que obteve resultados melhores do que a *k*-d *tree* e a *octree*, ficando somente atrás da *interval tree*. O ganho máximo obtido por ela, em relação à versão força bruta da técnica de cubos marchantes, foi de 14.1 vezes para o volume da angiografia. Para os outros volumes (com exceção do combustível), o ganho ficou entre 5.8 e 6.9 vezes.

Comparando a *k*-d *tree* com a *octree*, tem-se que a *octree* forneceu melhores resultados em CPU, mas teve um desempenho pior do que a *k*-d *tree* na GPU. Isso se deve ao fato do uso da abordagem do *span-space* na *k*-d *tree*, o que não ocorre na *octree*, que opera diretamente no volume de dados.

Com relação ao tempo de construção das estruturas de dados, a *quadtree* teve um de-

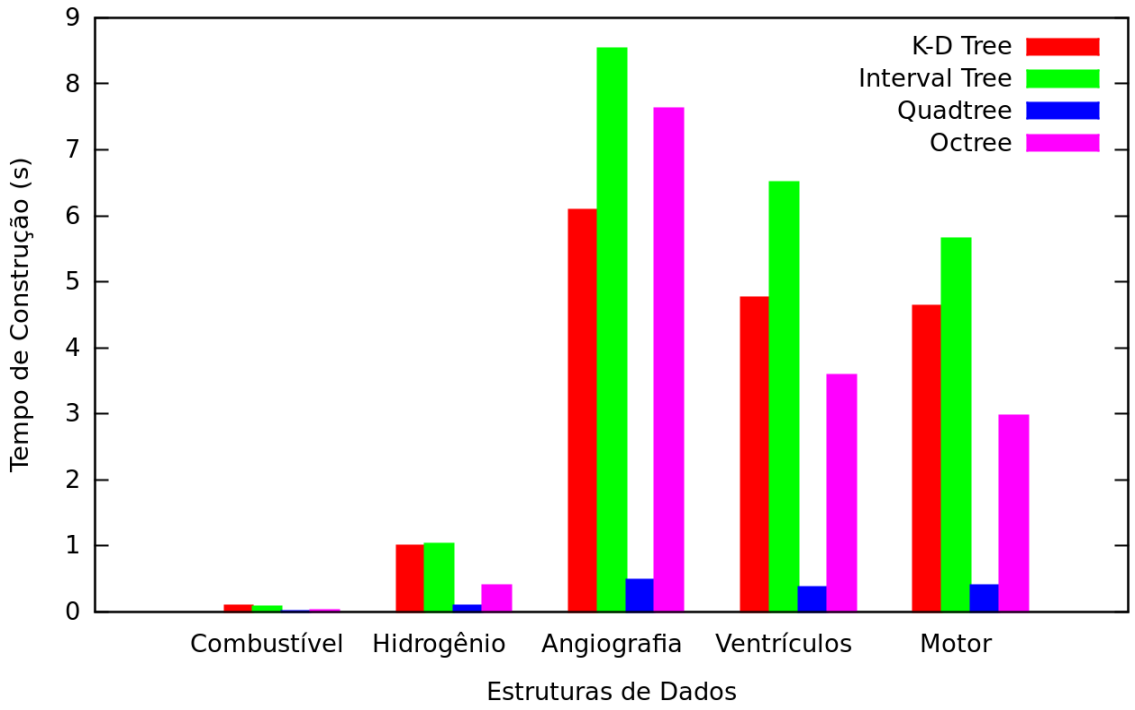
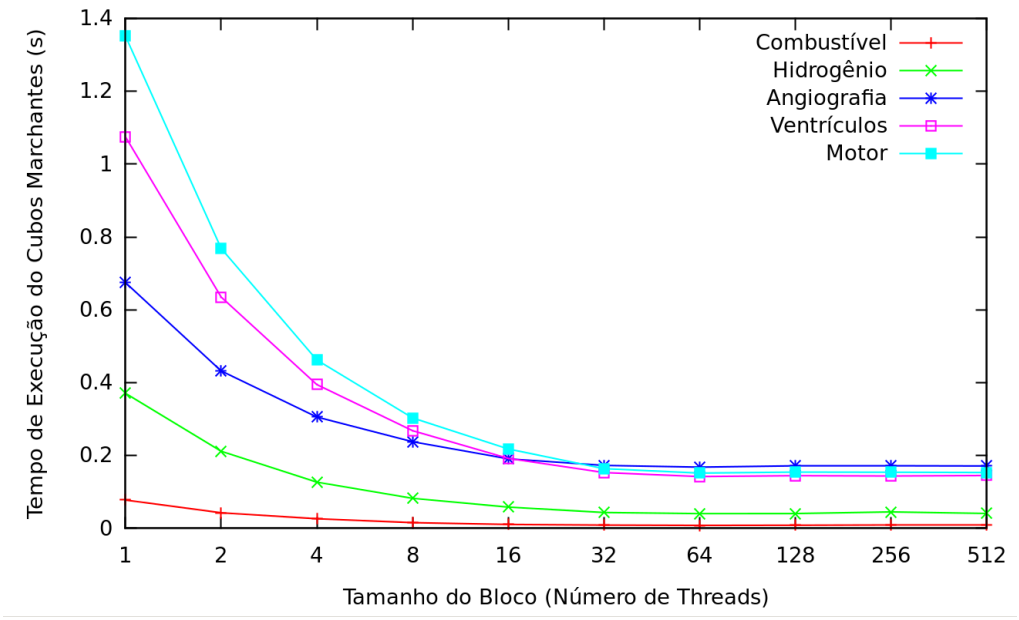


Figura 4.2: Tempos médios de construção das estruturas de dados para cada um dos volumes utilizados.

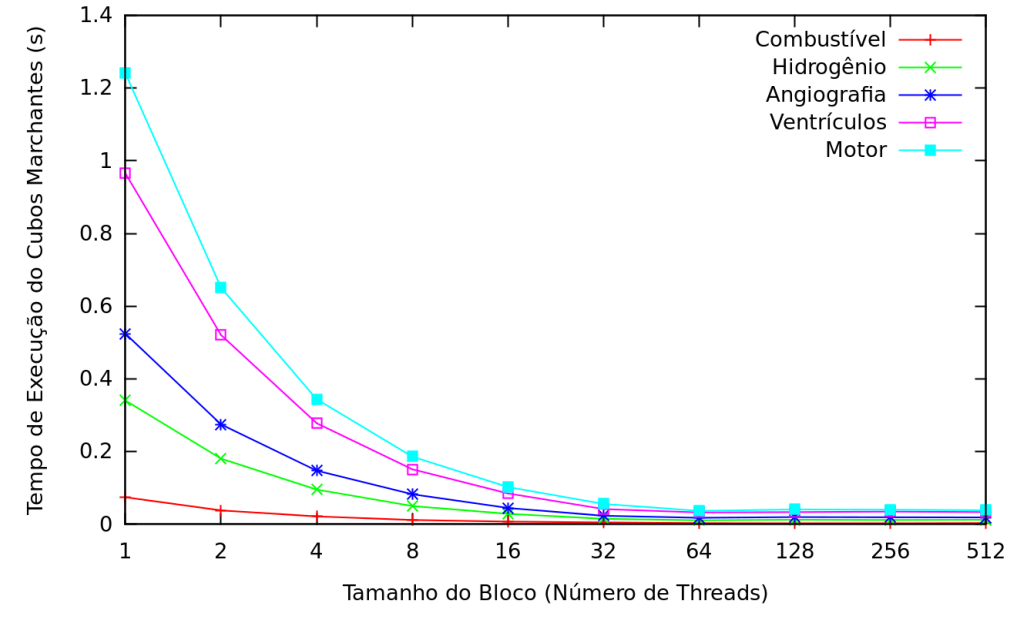
sempenho bem acima das demais estruturas, como mostrado na Figura 4.2. Para todos os volumes utilizados nos testes, o tempo de construção da *quadtree* foi inferior a 1 segundo, não havendo aumento significativo de tempo à medida que os tamanhos dos volumes ficam maiores, o que é bem eficiente para aplicações reais que processam grandes volumes de dados. Para as outras estruturas, o tempo de construção aumenta consideravelmente entre os volumes.

A *octree* forneceu os segundos melhores tempos de construção. Apesar de o seu algoritmo de construção ser semelhante ao da *quadtree*, o fato de ela ser aplicada no espaço 3D ocasiona um gasto maior de tempo, uma vez que o número total de subdivisões realizadas é maior. Na sequência, aparece a k-d *tree*, que possui um tempo de construção pior que o da *octree* por causa das sucessivas execuções do algoritmo de busca da mediana dos pontos do *span-space* (linhas 5 e 7 do Algoritmo 2). Por fim, a *interval tree* forneceu os piores tempos de construção entre as estruturas testadas. Tal fato se deve ao tempo gasto com as ordenações das listas de células (linhas 15 e 16 do Algoritmo 4), que têm uma complexidade de tempo maior do que a busca da mediana.

A Figura 4.3 mostra dois gráficos que ilustram a média de tempo de 50 execuções



(a)



(b)

Figura 4.3: Tempos médios de execução do algoritmo de cubos marchantes em GPU para diferentes tamanhos de bloco. O gráfico (a) mostra os resultados para a versão força bruta e (b) para a *interval tree*.

da implementação proposta do algoritmo de cubos marchantes em GPU para diferentes valores de tamanho de bloco, utilizando cada um dos volumes de dados e seus respectivos valores listados na Tabela 4.1. O gráfico (a) representa as execuções da versão força bruta do algoritmo de cubos marchantes (que é executada sem a aceleração com estruturas de dados) e (b) as execuções que utilizam uma *interval tree* para armazenar os voxels provenientes dos volumes.

A partir de ambos os gráficos, pode-se notar que, para tamanhos de bloco maiores que 64, o tempo médio de execução do algoritmo de cubos marchantes tem uma pequena alta. E isso não depende de usar ou não uma estrutura de dados para a aceleração do algoritmo, tampouco do tipo de estrutura de dados utilizado. Em outras palavras, o comportamento das curvas dos gráficos é semelhante para qualquer estrutura. Conforme especificado no Capítulo 3, embora um número alto de *threads* seja sinônimo de uma alta paralelização, o tempo gasto para criá-las também torna-se maior, o que estabiliza o desempenho do algoritmo de uma forma geral. Assim, o tamanho de bloco utilizado para executar os testes de desempenho da técnica de cubos marchantes foi definido em 64.

O principal gargalo da abordagem proposta reside no fato de que as buscas nas estruturas de dados são realizadas na CPU. Uma vez que os algoritmos de busca em árvore são, em geral, recursivos e como a arquitetura CUDA não tem suporte a funções recursivas, se versões iterativas desses algoritmos de busca na GPU fossem implementados, haveria um grande consumo de tempo e espaço somente para criar a quantidade adequada de pilhas e laços utilizados para simular as chamadas recursivas.

Capítulo 5

Conclusões e Trabalhos Futuros

Este trabalho apresentou uma contextualização geral de algumas técnicas inerentes à Visualização Volumétrica, juntamente com as tentativas de melhora de desempenho por intermédio de placas gráficas. Além disso, foi descrito o algoritmo de cubos marchantes, um dos mais utilizados em renderização, além de métodos de aceleração deste algoritmo com estruturas de dados espaciais e de uma abordagem para a sua aceleração em GPU com o uso da arquitetura CUDA. Experimentos foram conduzidos com vários volumes de dados e os tempos de execução do método proposto para cada estrutura de dados são apresentados.

Os resultados demonstram que foi possível acelerar o algoritmo de cubos marchantes em GPU por um fator de cerca de 16 vezes, comparado à versão força bruta do algoritmo em CPU. A estrutura de dados *interval tree* forneceu as melhores taxas de renderização, mas o tempo gasto com a construção da estrutura mostrou-se relativamente alto. Em compensação, a *quadtree* produziu resultados bastante satisfatórios para a construção, mesmo para volumes maiores, além de taxas de renderização um pouco piores que os da aceleração com a *interval tree*.

Um possível trabalho futuro é a implementação da metodologia descrita neste trabalho em OpenCL [41], que é uma plataforma aberta e trabalha com qualquer placa gráfica, independentemente do seu fabricante. No entanto, o fato de o OpenCL ser uma plataforma genérica pode implicar uma penalização no desempenho geral das aplicações [20]. Em outras palavras, espera-se que a implementação do algoritmo de cubos marchantes em OpenCL forneça resultados um pouco piores do que aqueles apresentados no Capítulo 4.

Para melhorar o desempenho da técnica de cubos marchantes em GPU, pretende-se também testar a possibilidade de transferir as estruturas de dados para a GPU e realizar o procedimento de busca de maneira paralela, por meio de um algoritmo de busca em grafos (como as implementações em CUDA das buscas em largura e em profundidade [15]). Não se sabe, porém, se os custos gerados com a representação do grafo associado à estrutura

por meio de uma lista de adjacências (representação mais adequada para árvores ou grafos esparsos) em GPU são suficientes para se obter um aumento na taxa de renderização com as buscas pelas células ativas do volume de dados.

Futuramente, espera-se também que as técnicas de visualização volumétrica em geral sejam aceleradas com o auxílio da nova geração de processadores, que integram CPU e GPU em um mesmo circuito, permitindo que ambos tenham acesso aos mesmos recursos de *hardware*. Assim, o gargalo existente na comunicação entre as duas unidades é eliminado, o que pode otimizar ainda mais o desempenho dessas técnicas.

Bibliografia

[1] J. L. Bentley. Multidimensional Binary Search Trees used for Associative Searching. *Communications of the ACM*, 18(9):509–517, 1975.

[2] M. Botsch, A. Hornung, M. Zwicker, and L. Kobbelt. High-Quality Surface Splatting on Today's GPUs. *Proceedings of Eurographics/IEEE VGTC Symposium Point-Based Graphics*, pages 17–141, 2005.

[3] L. Buatois, G. Caumon, and B. Lévy. GPU Accelerated Isosurface Extraction on Tetrahedral Grids. In *Advances in Visual Computing*, volume 4291 of *Lecture Notes in Computer Science*, pages 383–392. Springer Berlin / Heidelberg, 2006.

[4] S. Chan and E. Purisima. A New Tetrahedral Tesselation Scheme for Isosurface Generation. *Computers & Graphics*, 22(1):83–90, 1998.

[5] E. V. Chernyaev. Marching Cubes 33: Construction of Topologically Correct Isosurfaces. Technical report, Institute for High Energy Physics, Moscow, Russia, 1995.

[6] P. Cignoni, P. Marino, C. Montani, and R. Scopigno. Speeding up Isosurface Extraction using Interval Trees. *IEEE Transactions on Visualization and Computer Graphics*, 3:158–170, 1997.

[7] M. Cirne and H. Pedrini. Acceleration of the Marching Cubes Technique for Volumetric Visualization on Graphics Processing Unit Using Spatial Data Structures. In *Proceedings of the XVII Simpósio Brasileiro de Sistemas Multimídia e Web*, Florianópolis, SC, Brazil, 2011.

[8] L. Dematte and D. Prandi. GPU Computing for Systems Biology. *Briefings in Bioinformatics*, 11:323–333, 2010.

[9] M. Dürst. Letters: Additional Reference to Marching Cubes. *ACM Computers & Graphics*, 22(4):72–73, 1988.

[10] C. Dyken, G. Ziegler, C. Theobalt, and H. P. Seidel. High-Speed Marching Cubes using HistoPyramids. *Computer Graphics*, 27(8):2028–2039, July 2008.

[11] T. T. Elvins. A Survey of Algorithms for Volume Visualization. *SIGGRAPH Computer Graphics*, 26(3):194–201, 1992.

[12] J. Fangerau and S. Krömker. Parallel Volume Rendering Implementation on Graphics Cards Using CUDA. In R. Keller, D. Kramer, and J.-P. Weiss, editors, *Facing the Multicore-Challenge*, volume 6310 of *Lecture Notes in Computer Science*, pages 143–153. Springer Berlin / Heidelberg, 2011.

[13] F. Goetz, T. Junklewitz, and G. Domik. Real-Time Marching Cubes on the Vertex Shader. In *Proceedings of Eurographics*, Dublin, Ireland, Aug./Sept. 2005.

[14] A. Guéziec and R. Hummel. Exploiting Triangulated Surface Extraction using Tetrahedral Decomposition. *IEEE Transactions on Visualization and Computer Graphics*, 1(4):328–342, Dec. 1995.

[15] P. Harish and P. J. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *Proceedings of the 14th International Conference on High Performance Computing*, pages 197–208, Goa, India, 2007.

[16] O. Harrison and J. Waldron. GPU Accelerated Cryptography as an OS Service. In M. Gavrilova, C. Tan, and E. Moreno, editors, *Transactions on Computational Science XI*, volume 6480 of *Lecture Notes in Computer Science*, pages 104–130. Springer Berlin / Heidelberg, 2010.

[17] Y. Heng and L. Gu. GPU-based Volume Rendering for Medical Image Visualization. In *Proceedings of 27th Annual International Conference of the Engineering in Medicine and Biology Society*, volume 5, pages 5145–5148, 2005.

[18] G. Johansson. Accelerating Isosurface Extraction by Caching Cell Topology with Graphics Hardware. Master's thesis, University College Dublin, 2005.

[19] G. Johansson and H. Carr. Accelerating Marching Cubes with Graphics Hardware. In *Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative Research*, Toronto, ON, Canada, Oct. 2006.

[20] K. Karimi, N. G. Dickson, and F. Hamze. A Performance Comparison of CUDA and OpenCL. *Computing Research Repository*, 2010.

[21] A. Kaufman. *Volume Visualization*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1991.

[22] E. Keppel. Approximating Complex Surfaces by Triangulation of Contour Lines. *IBM Journal of Research and Development*, 19(1):2–11, 1975.

[23] J. Kruger and R. Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings of the 14th IEEE Visualization*, pages 38–43, Seattle, WA, USA, 2003.

[24] U. Labsik, K. Hormann, M. Meister, and G. Greiner. Hierarchical Iso-Surface Extraction. *Journal of Computing and Information Science in Engineering*, 2:323–329, 2002.

[25] P. Lacroute. *Fast Volume Rendering using a Shear-Warp Factorization of the Viewing Transformation*. PhD thesis, Stanford University, Stanford, CA, USA, 1995.

[26] M. Levoy. Efficient Ray Tracing of Volume Data. *ACM Transactions on Graphics*, 9(3):245–261, July 1990.

[27] Y. Livnat, H.-W. Shen, and C. R. Johnson. A Near Optimal Isosurface Extraction Algorithm Using The Span Space. *IEEE Transactions on Visualization and Computer Graphics*, 2(1):73–84, Mar. 1996.

[28] W. Lorensen and H. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *Computer Graphics*, 21(4):163–169, July 1987.

[29] I. H. Manssour and C. M. D. S. Freitas. Visualização Volumétrica. *Revista de Informática Teórica and Aplicada*, 9(2):97–126, 2002.

[30] R. Marroquim, A. Maximo, R. C. Farias, and C. Esperança. Volume and Isosurface Rendering with GPU-Accelerated Cell Projection. *Computer Graphics Forum*, 27:24–35, 2008.

[31] L. Marsalek, A. Hauber, and P. Slusallek. High-Speed Volume Ray Casting with CUDA. In *Proceedings of IEEE Symposium on Interactive Ray Tracing*, Los Angeles, CA, USA, Aug. 2008.

[32] S. Martin, H.-W. Shen, and P. McCormick. Load-Balanced Isosurfacing on Multi-GPU Clusters. In *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization*, pages 91–100, May 2010.

[33] J. Mensmann, T. Ropinski, and K. H. Hinrichs. An Advanced Volume Raycasting Technique using GPU Stream Processing. In *Proceedings of International Conference on Computer Graphics Theory and Applications*, pages 190–198, Angers, France, May 2010.

[34] K. Mueller, N. Shareef, J. Huang, and R. Crawfis. High-Quality Splatting on Rectilinear Grids with Efficient Culling of Occluded Voxels. *IEEE Transactions on Visualization and Computer Graphics*, 5(2):116–134, Apr. 1999.

[35] H. R. Nagel. GPU Optimized Marching Cubes Algorithm for Handling Very Large, Temporal Datasets. In *Workshop on State-of-the-Art in Scientific and Parallel Computing*, Trondheim, Norway, May 2008.

[36] T. S. Newman and H. Yi. A Survey of the Marching Cubes Algorithm. *Computers & Graphics*, 30(5):854–879, Oct. 2006.

[37] G. M. Nielson and B. Hamann. The Asymptotic Decider: Resolving the Ambiguity in Marching Cubes. In *Proceedings of the 2nd Conference on Visualization*, pages 83–91, San Diego, CA, USA, Oct. 1991.

[38] NVIDIA. CUDA C/C++ SDK Code Examples, 2011. `http://developer.nvidia.com/cuda-cc-sdk-code-samples`.

[39] NVIDIA. NVIDIA CUDA Zone, 2011. `http://www.nvidia.com/object/cuda_home.html`.

[40] J. F. Ohmer. Computer Vision Applications on Graphics Processing Units. Master's thesis, Queensland University of Technology, Brisbane, Australia, 2007.

[41] OpenCL. OpenCL - The Khronos Group, 2011. `http://www.khronos.org/opencl/`.

[42] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[43] V. Pascucci. Isosurface Computation Made Simple: Hardware Acceleration, Adaptive Refinement and Tetrahedral Stripping. In *Joint Eurographics - IEEE TVCG Symposium on Visualization*, pages 293–300, Konstanz, Germany, 2004.

[44] E. Polytechnique. Mean-shift clustering and image segmentation, 2011. `http://www.enseignement.polytechnique.fr/informatique/INF562/TD/TD7/INF562-TD7-1.html`.

[45] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes in C*. Cambridge University Press, 2nd edition, 1992.

[46] F. Reck, C. Dachsbacher, R. Grosso, G. Greiner, and M. Stamminger. Realtime Isosurface Extraction with Graphics Hardware. In *Eurographics 2004 Short Presentations and Interactive Demos*, pages 33–36, 2004.

[47] D. Ruijters and A. Vilanova. Optimizing GPU Volume Rendering. *Journal of WSCG*, 14:9–16, 2006.

[48] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, USA, 1990.

[49] H. Scharsach. Advanced GPU Raycasting. In *Proceedings of CESCG*, pages 69–76, Vienna, Austria, 2005.

[50] P. Shirley and A. Tuchman. A Polygonal Approximation to Direct Scalar Volume Rendering. *ACM SIGGRAPH Computer Graphics*, 24(5):63–70, Nov. 1990.

[51] C. Silva, J. Comba, S. Callahan, and F. Bernardon. A Survey of GPU-Based Volume Rendering of Unstructured Grids. *Revista de Informática Teórica and Aplicada*, 12(2):9–29, 2005.

[52] N. Tatarchuk, J. Shopf, and C. DeCoro. Real-Time Isosurface Extraction Using the GPU Programmable Geometry Pipeline. In *ACM SIGGRAPH 2007 Courses*, pages 122–137, San Diego, CA, USA, 2007.

[53] VolVis. Volumes de Dados, 2011. `http://www.volvis.org`.

[54] VolViz. Volume Ray Casting, 2011. `http://www.volviz.com`.

[55] L. Westover. Footprint Evaluation for Volume Rendering. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, pages 367–376, New York, NY, USA, 1990. ACM.

[56] J. Wilhelms and A. V. Gelder. Octrees for Faster Isosurface Generation. *ACM Transactions on Graphics*, 11(3):201–227, July 1992.

[57] J. Wilhelms, Jane, and A. V. Gelder. A Coherent Projection Approach For Direct Volume Rendering. *ACM SIGGRAPH Computer Graphics*, 25(4):275–284, July 1991.

[58] D. Xue and R. Crawfis. Efficient Splatting Using Modern Graphics Hardware. *Journal of Graphics, GPU and Game Tools*, 8(3):1–21, 2003.