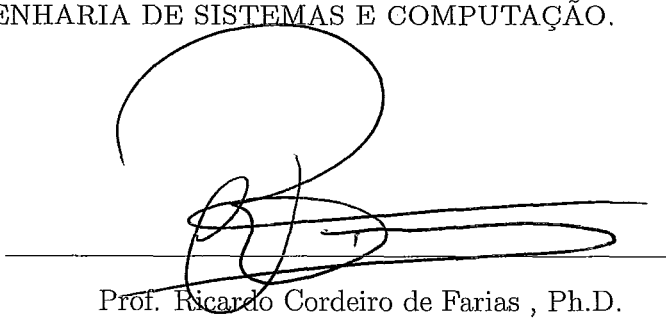
DIVISÃO ESPACIAL DE DADOS VOLUMÉTRICOS REGULARES PARA RENDERIZAÇÃO OUT-OF-CORE

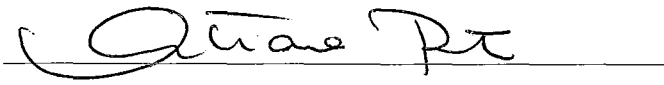
Luis Carlos dos Santos Coutinho Retondaro

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

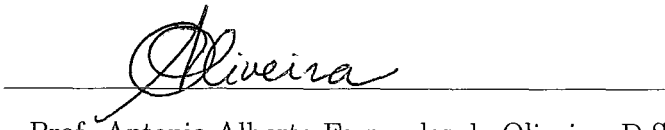
Aprovada por:



Prof. Ricardo Cordeiro de Farias , Ph.D.



Prof. Cristiana Bentes, D.Sc.



Prof. Antonio Alberto Fernandes de Oliveira, D.Sc.

RIO DE JANEIRO, RJ - BRASIL

MARÇO DE 2008

RETONDARO, LUIS CARLOS DOS SANTOS COUTINHO

Divisão Espacial de Dados Volumétricos Regulares para Renderização Out-Of-Core [Rio de Janeiro] 2008

XII, 48 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2008)

Dissertação - Universidade Federal do Rio de Janeiro, COPPE

1. Visualização Volumétrica
2. Renderização Out-of-core
3. Renderização de Alto Desempenho

I. COPPE/UFRJ II. Título (série)

Dedico este trabalho aos que, sem pedir nada em troca, se comprometeram com o meu sucesso: sustentando em oração, apoiando, sugerindo e criticando. E também aos que estiveram um pouco mais distantes, mas me honraram com a possibilidade de compartilhar um pouco de suas experiências. A Jesus Cristo, meu Senhor e Rei!

Agradecimentos

As palavras não traduzem a intensidade do sentimento de gratidão que tenho. Porém, desejo reverenciar algumas pessoas que souberam ser humanas e me prestaram muitos cuidados.

À minha amada Raquel, pelo amor incondicional e por suportar as ausências e os tortuosos dias de recomeço constante. Obrigado, porque suas lágrimas alimentaram minha fé. Às minhas filhas Paula e Isabela, pela simplicidade de cada gesto e cada bilhetinho que suas minúsculas mãos puderam produzir, revelando que seus corações são gigantes.

À minha mãe Delacilda por saber me ouvir com toda atenção, deixando sempre uma palavra sábia e útil em todas as situações. À meu pai Antônio, por saber incentivar e inflamar os ânimos, nas horas de incerteza ou dificuldade. À minha irmã Fernanda, por ser exemplo e fonte de tanta inspiração construtiva em minha carreira. À meu irmão Eduardo, por se tornar mais próximo, nas horas que precisei de um amigo. À minha irmã Raquel, pela forma especial como eleva meu conceito pessoal, além do que realmente mereço.

Aos demais familiares, cujas participações diretas ou indiretas, muito me influenciaram.

Aos irmãos em Cristo, que perseveraram comigo em todo tempo.

Aos colegas do LCG, que me ensinaram e ajudaram em muitos aspectos, apoiando e fundamentando alguns argumentos que hoje defendo.

Aos funcionários e professores, sempre interessados e solícitos comigo. Obrigado pela expressão de responsabilidade e comprometimento.

Em último, porém mais relevante, ao Deus criador dos céus e da terra, Pai de Jesus Cristo, pelo estabelecimento do seu plano em minha vida. Obrigado Senhor por considerar o desejo do meu coração, mesmo eu não sendo digno de honra.

Esta realização é também fruto da ação destas pessoas, as quais agradeço e presto meus sentimentos de mais profunda consideração. Deus os abençôe!

Rio de Janeiro, Março de 2008

Luis Carlos dos Santos Coutinho

Retondaro

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

DIVISÃO ESPACIAL DE DADOS VOLUMÉTRICOS REGULARES PARA RENDERIZAÇÃO OUT-OF-CORE

Luis Carlos dos Santos Coutinho Retondaro

Março/2008

Orientador: Ricardo Cordeiro de Farias

Programa: Engenharia de Sistemas e Computação

Um grande desafio em visualização volumétrica, é obter imagens de alta qualidade em tempo real. Isto é particularmente interessante para os conjuntos de dados científicos, onde os resultados altamente precisos são requeridos para assegurar a sua interpretação exata. O RZSweep é uma versão do algoritmo de renderização volumétrica ZSweep, para malhas regulares. O algoritmo é baseado no paradigma do plano de varredura (sweep plane). Embora, alguma pesquisa já tenha sido feita para dados irregulares, o RZSweep é a primeira tentativa de explorar as potencialidades do paradigma do plano de varredura para malhas regulares. O RZSweep, entretanto, possui apenas uma implementação sequencial incore, ou seja, toda a malha deve ser carregada na memória principal. A proposta deste trabalho é de implementar a versão out-of-core do RZSweep, fazendo com que a malha de dados regular seja armazenada no disco segundo a estrutura hierárquica de uma octree e a leitura dos dados seja feita por demanda, buscando-se folhas da octree. Além disso, o PRZSweep-OOC como é chamado, implementa o paralelismo aumentando a performance da renderização de grandes massas de dados, que não cabem na memória de um computador. O paralelismo é implementado através da execução em cluster de PCs. As aplicações para este trabalho podem ser: análise de imagens médicas, arqueologia e paleontologia, ramo petrolífero, dinâmica de fluídos, engenharia aeroespacial, fenômenos químicos.

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

SPACIAL DIVISION OF REGULAR VOLUMETRIC DATA FOR OUT-OF-CORE RENDERING

Luis Carlos dos Santos Coutinho Retondaro

March/2008

Advisor: Ricardo Cordeiro de Farias

Department: Systems Engineering and Computer Science

A major challenge in volume visualization, is to get high-quality images in real time. This is particularly interesting for combinations of scientific data, where the highly accurate results are required to ensure their accurate interpretation. The RZSweep is a version of the algorithm for volumetric rendering ZSweep, to regular meshes. The algorithm is based on the paradigm of sweep plane. Although some research has already been done for irregular data, the RZSweep is the first attempt to exploit the potential of the paradigm of the plan to scan for regular mesh. The RZSweep, however, has only an implementation incore sequence, or the entire mesh must be loaded in main memory. The purpose of this work is to implement a version out-of-core of RZSweep, making the regular mesh data is stored on the disk according to the hierarchical structure of an octree and reading of data is made on demand, looking up the leaves octree. Moreover, the PRZSweep-OOC as it is called, implements the parallelism increasing the performance of the rendering of large masses of data, which do not fit in memory of a computer. The parallelism is implemented through the execution in cluster of PCs. Applications for this work can be: analysis of medical images, archaeology and paleontology, oil industry, dynamics of fluids, aerospace engineering, chemical phenomena.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A Visualização Científica é uma subárea da Computação Gráfica que considera a geração de imagens, a partir das especificações geométricas e visuais de seus componentes. A visualização científica se preocupa com a representação gráfica da informação, de forma a facilitar o entendimento de dados tri-dimensionais de grande complexidade e tamanho, como por exemplo, os dados de dinâmica dos fluidos, ou simulações espaciais.

Os dados a serem visualizados podem ser obtidos de diferentes formas e estão associados a diversas áreas do conhecimento. O processo de visualização destes dados, é denominado **Visualização Volumétrica**. A visualização volumétrica gera sinteticamente uma imagem bidimensional como sendo uma amostra de um dado volumétrico.

Através deste método, é possível visualizar a massa de dados como um corpo composto de partes semitransparentes, permitindo visualizar todo o seu interior e não apenas a sua superfície. As imagens geradas, portanto, são de alta qualidade. Visualizar as estruturas internas do dado volumétrico permite uma exploração mais detalhada de características inerentes à aplicação em questão; bem como, uma análise dinâmica da variação destas características. Algumas aplicações importantes que empregam visualização volumétrica, são:

- Imagens médicas (no auxílio de diagnóstico de doenças);

- Geologia (na detecção e exploração de recursos naturais);
- Dinâmica de fluidos (na pesquisa do comportamento físico de gases e líquidos);
- Indústria (na inspeção interna de materiais ou partes mecânicas);
- Meteorologia (na previsão do tempo e estudos de fenômenos naturais);
- Biologia (na análise microscópica de substâncias e microorganismos).

A visualização volumétrica de uma grande massa de dados é um problema dispendioso em termos computacionais. Esta tarefa é também conhecida como **Renderização de Volume**. É uma tarefa desafiadora, na medida que os dados tridimensionais são cada vez maiores, amostrados em posições discretas do espaço e na maioria dos casos variam com o tempo. Soluções para este problema devem combinar diversas técnicas matemáticas e fazer uso dos recursos computacionais mais avançados à disposição, como por exemplo, sistemas com multi-processadores e hardware gráfico programável.

As soluções já implementadas para visualização volumétrica, podem ser divididos em algoritmos que pertencem a três paradigmas de implementação: espaço do objeto, espaço da imagem e métodos de domínio. Os algoritmos que trabalham no espaço do objeto, calculam a contribuição de cada célula do volume para produzir a imagem final. Como exemplo, podemos citar o algoritmo de projeção de células (*cell projection*), no qual cada célula é projetada no plano da imagem, sendo estas projeções combinadas.

Nos algoritmos que trabalham no espaço da imagem, são analisados cada pixel da imagem a ser gerada, por exemplo: o algoritmo de traçado de raios (*ray-tracing*). Neste algoritmo, considerado inicialmente por Blinn [1], os raios são traçados a partir de cada pixel na direção do volume, calculando a contribuição de cada célula ao longo do raio, para gerar a imagem final. Nos métodos de domínio, os dados da região são transformados em um domínio alternativo, como compressão, *wavelet* ou domínio de frequência, no qual uma projeção é diretamente gerada.

No intuito de melhorar a performance na geração de imagens, são necessários algoritmos que combinem de forma híbrida as vantagens das abordagens acima

mencionadas. Uma algoritmo que pretende melhorar a performance na visualização volumétrica, vem do trabalho proposto por Farias et al. [2]. Eles propõem uma nova abordagem de plano de varredura. O algoritmo ZSweep, realiza a visualização do volume com o método de projeção de células.

A parelelização do algortimo ZSweep, é uma abordagem que se aplica a recente necessidade de mesclar o uso de recursos tecnológicos avançados, com melhores técnicas computacionais. O PZSweep [3] [4], como é chamado, foi implementado para a execução em sistemas multi-computadores. Este algoritmo possui duas características principais: explora o paralelismo potencial do ZSweep, permitindo o aumento de eficiência com o aumento de processadores e, permite a renderização *out-of-core*, isto é, pode processar um volume de dados tão grande que não cabe na memória principal do computador.

Ainda que extremamente eficiente para uma grande massa de dados, tanto o ZSweep como o PZSweep têm uma limitação: só são aplicados a dados volumétricos que sejam representados em malhas irregulares. Alguns exemplos, são os dados volumétricos de imagens oceânicas, Sistemas de Infomação Geográfica (*Geographic Information System - GIS*) e Modelos de Elevação Digital (*Digital Elevation Models – DEM*) que chegam a atingir a casa dos terabytes, e são representados em malhas irregulares.

Os dados volumétricos também podem ser representados em malhas regulares. Exemplos de utilização muito comuns desta representação, são as imagens médicas. Estes dados têm recebido muita atenção da comunidade de científica, devido à sua importância na análise de dados médicos e científicos em 3D, adquiridos principalmente, por equipamentos de ressonância magnética (*MRI*) e tomografia computadorizada (*CT*). Devido à grande importância de se trabalhar com malhas regulares, uma versão especial do ZSweep foi criada para estes tipos de dados, o RZSweep [5]. O RZSweep é a versão serial do ZSweep, aplicada exclusivamente a malhas regulares.

O objetivo deste trabalho é inserir as funcionalidades *out-of-core* e a execução paralela no ambiente do RZSweep, que opera com dados regulares. Este algoritmo é denominado PRZSweep-OOC, porque implementa a renderização volumétrica com duas funcionalidades adicionais: a renderização *out-of-core* e o paralelismo do pro-

cessamento.

Para a implementação da renderização volumétrica *out-of-core*, foi proposto um método de subdivisão no espaço do objeto. Como o PRZSweep-OOC trabalha com malhas regulares, a representação do dado é implícita, e a subdivisão foi realizada segundo a estrutura de dados chamada *Octree*. Na *octree* são armazenadas as caixas envolventes das regiões volumétricas que serão posteriormente acessadas. Com o PRZSweep-OOC, é possível carregar na memória do computador apenas uma determinada porção do volume que será acessada no momento da renderização. Com o uso eficiente de memória, o custo computacional é minimizado

O paralelismo de processamento faz parte da solução proposta pelo PRZSweep-OOC. Através da divisão de tarefas entre processadores, a renderização de grandes massas de dados pode ser mais eficiente e de baixo custo computacional. Neste caso, a divisão proposta é baseada no espaço da imagem. A imagem é subdividida em retângulos de tamanhos iguais chamados *tiles*, e cada processador será responsável por executar a tarefa de renderização de cada um destes *tiles*.

Dessa forma, este trabalho pretende completar uma familia de algoritmos de renderização baseados no ZSweep que trabalham com grandes massas de dados (regulares ou irregulares) com método de execução eficiente e tempo reduzido.

Capítulo 2

Conceitos Básicos

Neste capítulo são tratados alguns conceitos básicos relacionados à área de renderização volumétrica. São relacionados também, os principais algoritmos de renderização volumétrica e os conceitos associados à implementação do PRZSweep-OOC.

2.1 Renderização Volumétrica

A visualização pode retratar somente a superfície do objeto por meio de uma renderização de superfície, ou retratar todo o interior do objeto por meio de uma renderização volumétrica, conforme ilustrado na Figura 2.1. Os dados a serem visualizados podem ser obtidos de diferentes formas, e estão associados a diversas áreas do conhecimento, como por exemplo, a medicina.

A visualização através da superfície, não é apropriada para dados volumétricos altamente complexos, como o caso de imagens médicas que são compostas de muitas microestruturas. Em resposta a esta dificuldade, as técnicas de renderização volumétrica são usadas para representar os dados 3D em uma simples imagem 2D projetada a partir de um dado volumétrico [7].

A imagem gerada por renderização volumétrica, contém mais informação sobre o dado do que a visualização através de superfície. Entretanto, o custo computacional é maior, devido a alta complexidade do algoritmo. É um processo extremamente cus-

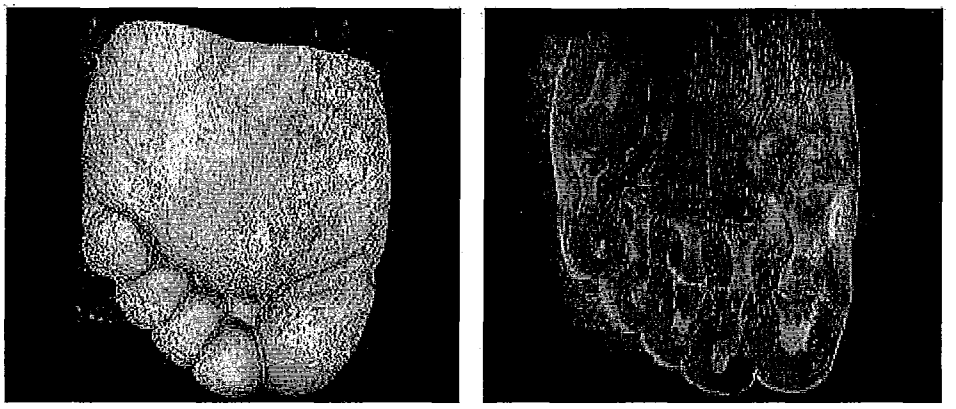


Figura 2.1: Diferença entre a visualização da superfície de um objeto (à esquerda) e do volume (à direita) [6].

toso computacionalmente. Pois, o resultado final é obtido através do processamento de todas as amostras de dados do volume, para gerar a imagem bidimensional.

2.2 Massa de Dados

Segundo Kaufman [8], cada amostra do dado volumétrico é representada por S=(x,y,z,v); sendo (x,y,z) a localização no espaço tridimensional do valor da propriedade v analisada. O valor v pode representar um campo escalar, por exemplo: densidade, temperatura, etc; ou ainda, pode representar um campo vetorial, por exemplo: velocidade, fluxo, etc.

Esta representação é feita através de malhas regulares e os dados podem ser definidos em uma matriz tridimensional, também chamada de *volume buffer*, *3D raster* ou simplesmente volume, com valores escalares Sijk; veja a Figura 2.2. Como ilustração, uma definição de matriz com valores vetoriais também pode ser vista na Figura 2.3.

Na renderização volumétrica, a imagem gerada é composta por uma matriz de elementos de figura, chamados de *pixels (picture elements)* [10], enquanto que dados volumétricos contém elementos de volume, chamados *voxels (volume elements)* [7]; conforme mostrado na Figura 2.4.

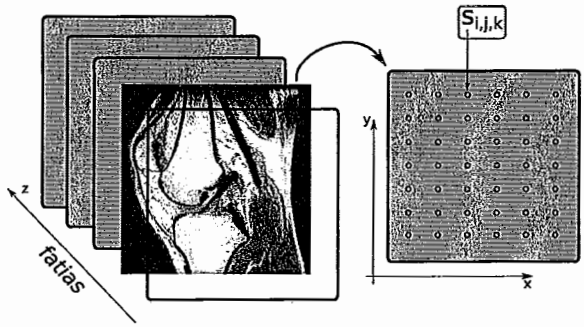


Figura 2.2: Visualização de um campo escalar. As fatias são dados amostrados da densidade de um joelho.

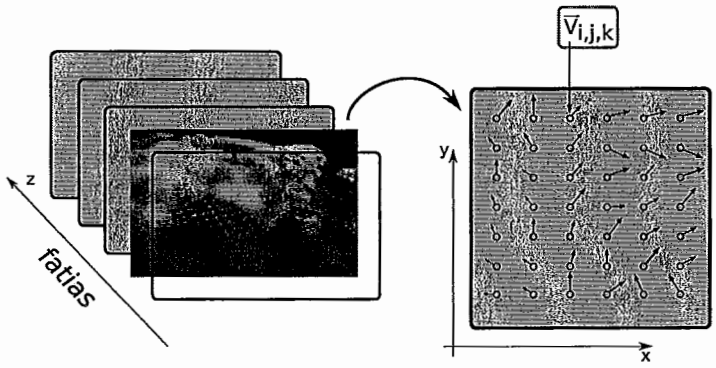


Figura 2.3: Visualização de um campo vetorial [9]. As fatias são dados computados para simulaçao de efeitos atmosféricos.

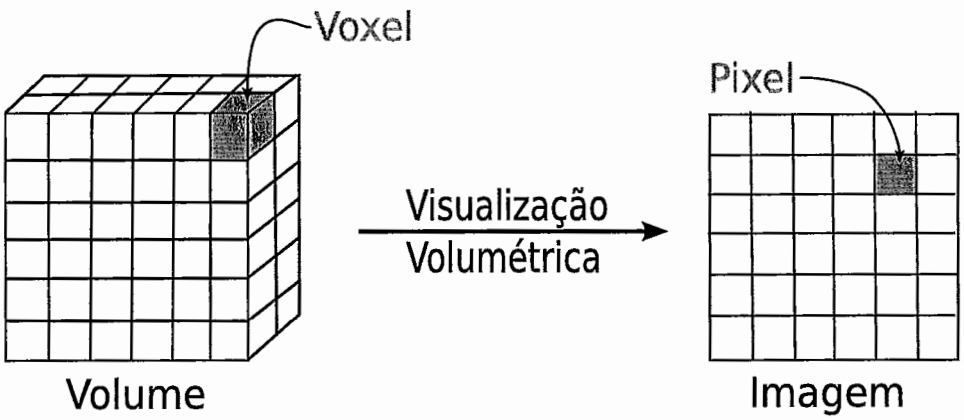


Figura 2.4: Visualização volumétrica é usada para gerar imagens a partir de informações do interior de volumes.

Na verdade, o dado volumétrico é formado como um conjunto de imagens ou

fatias (*slices*) sendo cada fatia, um conjunto de amostras S.

Os dados volumétricos são representados como uma estrutura geométrica em forma de malha poliedral compostas por células ou voxels tridimensionais. As células referem-se à região em torno de cada amostra do dado. Se a estrutura poliedral das células formam cubos idênticos, ou seja, se o espaçamento entre as células compreendem intervalos regulares nos três eixos ortogonais a malha é dita regular.

Alguns dados podem ser representados também por malhas irregulares. Nesta caso, as células são não-estruturadas e a conectividade entre elas deve ser explicitamente fornecida. A representação de malhas regulares e irregulares podem ser vistas nas figuras 2.5 e 2.6, respectivamente.

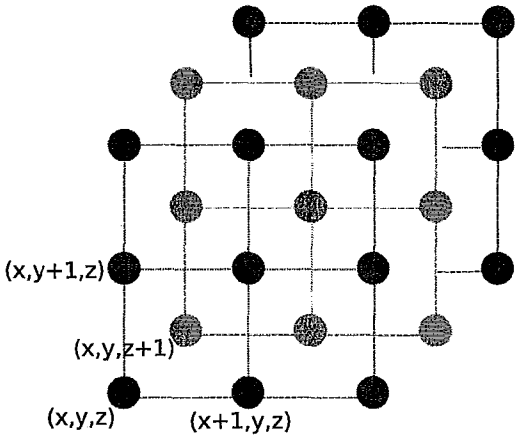


Figura 2.5: Representação em malha regular

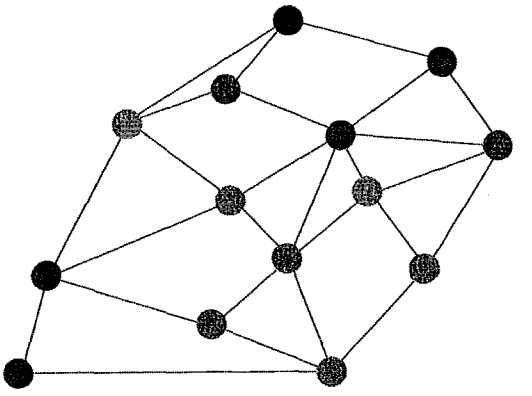


Figura 2.6: Representação em malha irregular

Além das grades regulares (*regular grid*), existem muitos outros tipos. O emprego de grades de determinado tipo, dependerá da aplicação. Podem ser empregadas também, grades retilíneas, curvilíneas e não-estruturadas; veja a Figura 2.7.

Em uma grade retilínea (*rectilinear grid*) as células são alinhadas com os eixos, mas o espaçamento na grade, ao longo dos eixos, pode ser arbitrário. Quando tal grade é transformada de forma não-linear enquanto preserva a topologia, a grade torna-se curvilínea (*curvilinear grid*), também chamada de grade estruturada (*structured grid*). Usualmente, a grade retilínea definindo uma organização lógica é chamada de espaço computacional (*computational space*) e a grade curvilínea é chamada

de espaço físico (*physical space*).

A grade é chamada de não-estruturada (*unstructured grid*) ou irregular, quando não há uma organização lógica. Um dado volumétrico não-estruturado ou irregular é uma coleção de células cuja conectividade deve ser explicitamente fornecida. Essas células podem ter um formato arbitrário, como tetraedros, hexaedros ou prismas.

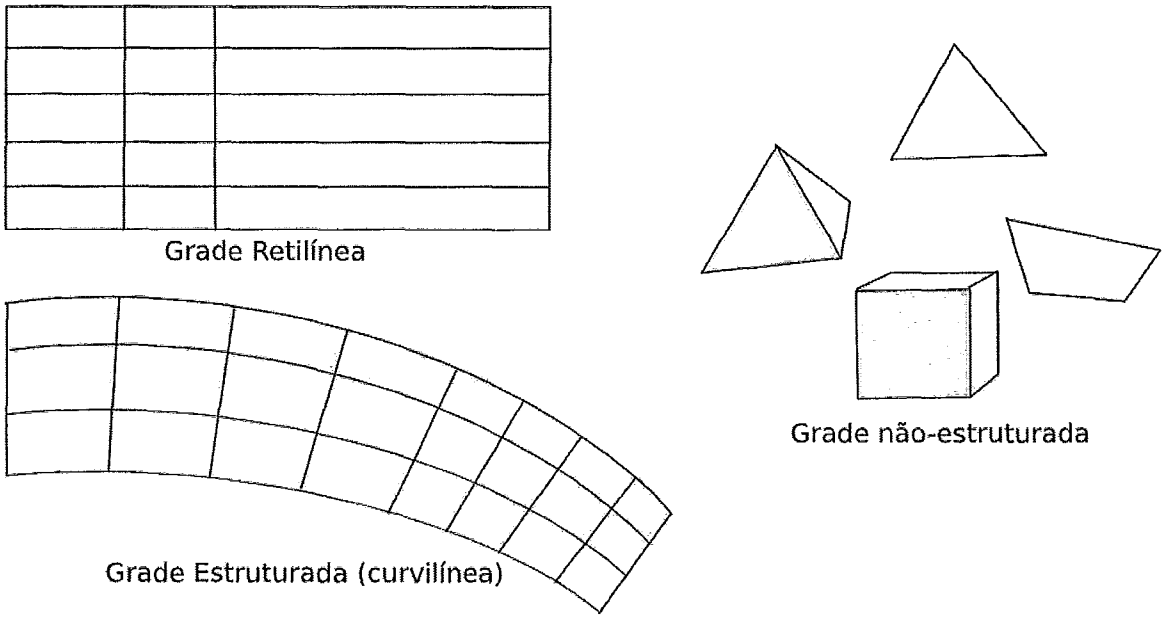


Figura 2.7: Representação em grades de diversos tipos.

Além de serem obtidos de amostras de objetos ou fenômenos reais, os dados volumétricos podem também ser produzidos por simulação computacional, ou gerados a partir de um modelo geométrico.

2.3 Algoritmos de Renderização Volumétrica

Alguns algoritmos clássicos de renderização volumétrica tentam combinar as abordagens no espaço do objeto com as técnicas baseadas no espaço da imagem, com o objetivo de melhorar a performance na geração de imagens. Entre estes, destacam-se: o algoritmo de traçados de raios, o de projeção de células e o plano de varredura.

2.3.1 O Algoritmo de Traçado de Raios

A técnica de traçado de raios (*ray-tracing*) é mais conhecida na área de visualização volumétrica por *ray-casting*. Sendo muito antiga na área de computação gráfica, seus conceitos foram inicialmente considerados em visualização volumétrica por Blinn [1].

Nesta técnica, para cada pixel do plano da imagem a ser gerada, é lançado um raio que percorre o volume do dado no ângulo de observação. Cada voxel cortado pelo raio contribui com cor e opacidade e o somatório das contribuições resulta na cor final do pixel.

A Figura 2.8 apresenta este princípio, onde um raio correspondente a um determinado pixel da tela, corta apenas um voxel do volume. Os dois pontos de interseção do raio no voxel, z1 e z2, são utilizados para se calcular o quanto esta fatia do volume interfere na cor e na opacidade do pixel em questão. Para cada voxel intersectado pelo raio a mesma computação é realizada, até que o raio deixe o volume.

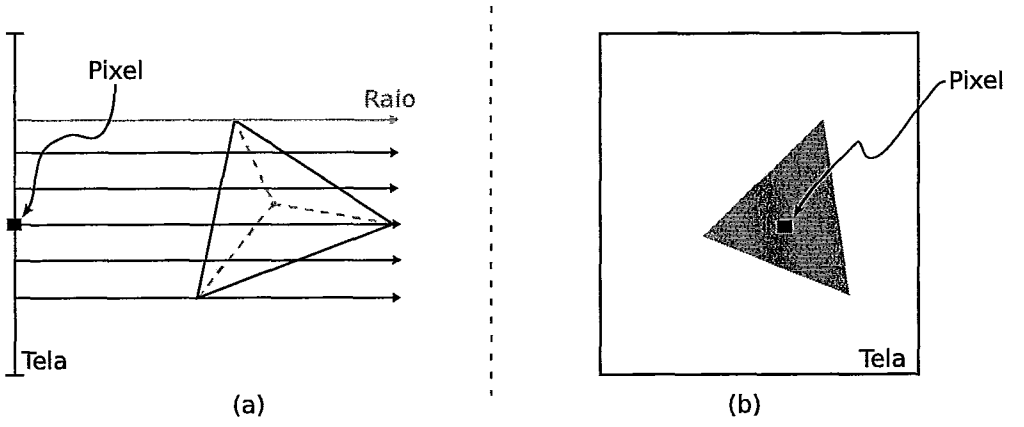


Figura 2.8: Exemplo da técnica de traçado de raios. A visão lateral dos raios atravessando uma célula do volume por pixel na tela é mostrada em (a), e o resultado em (b).

É uma técnica que demanda muito tempo de processamento, devido à grande quantidade de cálculos, pois geralmente são utilizados cálculos de interpolação trilinear, a fim de obter melhores resultados.

Uma limitação dos algoritmos baseados em traçados de raios por exemplo, é não aproveitar a coerência entre os raios, ou seja, as interseções com as células são recomputadas, para raios próximos, mesmo que tais células sejam duplicadas.

Posteriormente, o trabalho de Garrity [11] apresenta uma abordagem mais eficiente para o algoritmo de traçado de raios. Seu método traça os raios, em dados não-estruturados com transparência, considerando apenas a entrada do raio no volume por uma face externa (*external face*). As faces externas, também chamadas de faces da borda (*boundary faces*), são aquelas que pertencem à apenas uma célula. Em geral, o número de faces externas é bem menor que o total de faces. Com isso, Garrity testa apenas as faces externas reduzindo consideravelmente o número de testes de interseção inicial; veja a Figura 2.9. Bunyk. et al. [12] aceleram este processo computando todas as interseções dos raios com cada face externa da frente de uma única vez.

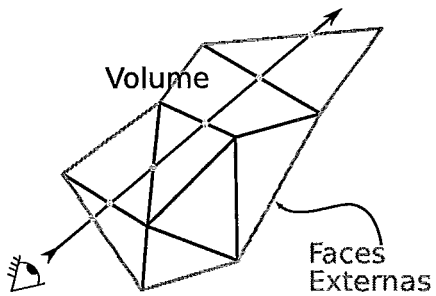


Figura 2.9: Uso das faces externas nos testes de interseção inicial do raio.

Outras versões mais atuais são consideradas, principalmente aquelas que executam o processamento em GPU. É o caso do algoritmo HARC (*Hardware-Based Ray Casting*) desenvovido por Weiler et al. [13], e o algoritmo de Espinha e Celes [14] que traz um incremento em relação ao HARC original.

Estes últimos algoritmos não são estudados no escopo deste trabalho, pelo fato da solução proposta não ser implementada em GPU. Esta escolha deve-se ao fato de que a implementação baseia-se em uma alternativa mais genérica e indcpendente da disponibilidade de recursos de hardware.

2.3.2 O Algoritmo de Projeção de Células

A técnica de projeção de células (*cell projection*) é também chamada de projeção direta (*direct projection*). Neste paradigma de renderização, são geradas imagens do volume a partir da projeção de suas células na tela. A projeção é determinada

transformando as células tridimensionais em primitivas geométricas bidimensionais no plano da imagem, ou plano de visão (*view plane*). Depois que as projeções das células são determinadas (veja a Figura 2.10), o processo de rasterização preenche as primitivas geométricas geradas com fragmentos, que são combinados na composição final do pixel. Como visto anteriormente, este método executa no espaço do objeto, pelo fato do algoritmo processar todas as células do volume.

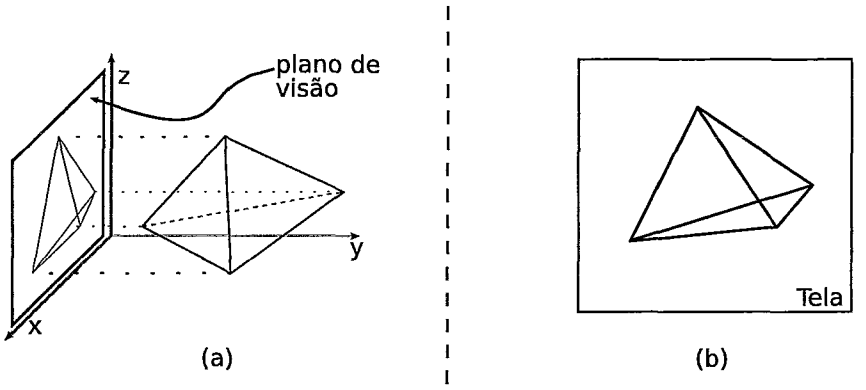


Figura 2.10: Exemplo da técnica de projeção de células. A projeção de uma célula do volume é mostrada em (a) e o resultado em (b).

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

A principal desvantagem de projeção de células é a dependência de outro algoritmo para determinar a ordem de visibilidade (*visibility order*) das células.

2.3.3 O Algoritmo Plano de Varredura

A idéia de varredura *(sweep)*, origina-se na geometria computacional [16], com o intuito de subdimensionar o problema e aproveitar a coerência entre os raios. Um algoritmo de varredura com traçados de raios para visualização volumétrica foi introduzido por Giertsen [17], onde ele cria o plano de varredura (*sweep plane*).

O plano de varredura percorre o volume para a geração da imagem final; veja a Figura 2.11. O plano caminha, perpendicular à tela, em eventos específicos, na ordem crescente da coordenada z. Estes eventos são os vértices v do volume onde a topografia dos polígonos muda e raios são traçados através do plano para determinar as cores dos pixels. Giertsen varre o modelo com um plano mantendo um conjunto de polígonos formados com a interseção do modelo com este plano. A vantagem do algoritmo de varredura decorre da atualização incremental das interseções, ao invés de arbitrária, aproveitando a coerência entre os raios.

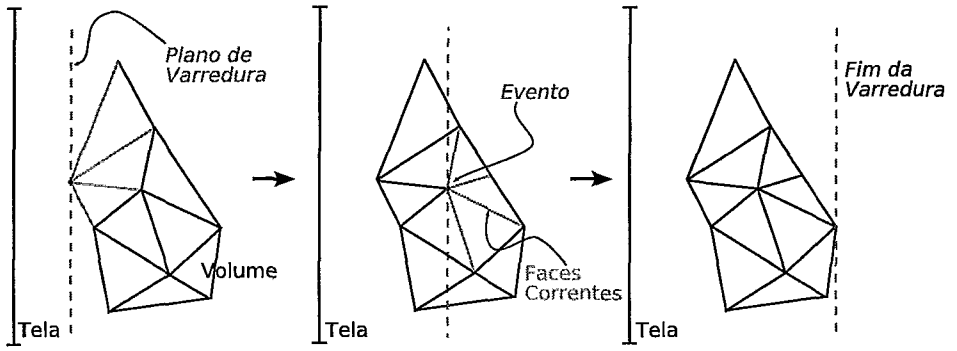


Figura 2.11: Exemplo de plano de varredura.

Mais tarde, Silva et al. [18, 19, 20], implementaram um algoritmo que incrementou o desempenho da abordagem de Giertsen, usando além do plano, uma linha de varredura (*sweep-line*) para determinar as interseções dos polígonos no plano com os raios traçados.

2.3.4 O Algoritmo ZSweep

No algoritmo ZSweep [2] o paradigma de plano de varredura, é incrementado com a utilização do método de projeção de células. Um plano paralelo ao plano da imagem percorre o corpo na direção do eixo z em ordem crescente, em eventos específicos. Estes eventos são os vértices v do volume, e a medida que ele transpassa o volume do dado, as faces dos voxels são projetadas na tela formando a imagem; veja a Figura 2.12.

Cada face projetada na tela gera um conjunto de interseções com uma série de pixels. Essas interseções são utilizadas, tal qual no algoritmo de traçado de raios,

para o cálculo da contribuição da face na cor e opacidade final de cada pixel.

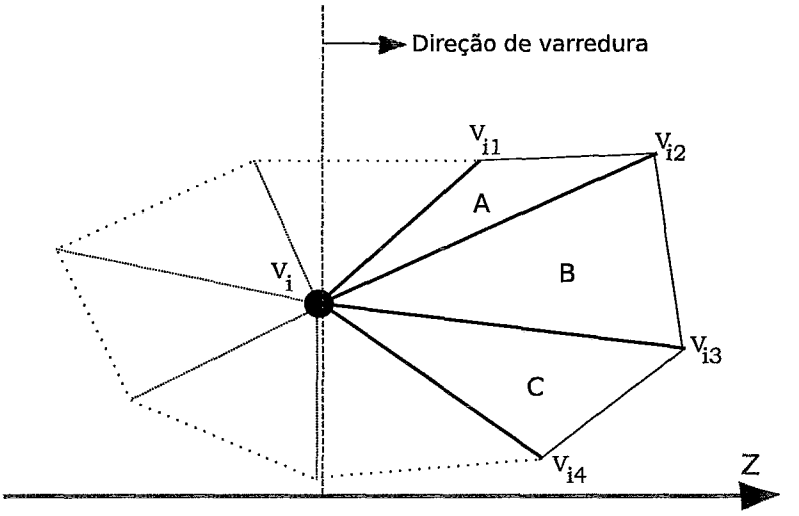


Figura 2.12: Quando o plano de varredura encontra o vértice V_i, as células A, B e C são encontradas e então as faces (V_i,V_{i1}), (V_i,V_{i2}), (V_i,V_{i3}) e (V_i,V_{i4}) são projetadas.

O primeiro passo do algoritmo é ordenar os vértices na ordem crescente da coordenada z, determinando a ordem dos eventos. Em cada evento do ZSweep, as faces ligadas ao vértice do evento são projetadas.

A projeção de faces no ZSwep, entretanto, é diferente dos métodos tradicionais porque utiliza composição adiantada. O ZSweep computa, na fase de projeção de faces, o valor z desta projeção sobre os pixels da imagem. Para cada pixel p, é criada uma lista com as projeções das faces, ordenada por ordem crescente de z.

Os cálculos de iluminação em p, entretanto, são efetivamente realizados quando o plano de varredura atinge um determinado valor alvo de z, chamado *target-z*. Ou seja, a projeção apenas incrementa a lista de interseções, até que o *target-z* é alcançado.

O *target-z* de um plano que intersectou o vértice v é um plano paralelo que passa pelo vértice com maior coordenada z, vizinho de primeira ordem de v, e portanto, pertencente à célula incidente em v.

Quando um plano atinge seu *target-z*, é realizada a composição na lista de interseções gerando a iluminação parcial dos pixels. O interessante desta composição antecipada é que no *target-z* a lista de interseções já pode ser corretamente com-

posta e todas as interseções processadas podem ser retiradas das listas evitando assim que estas listas cresçam indefinidamente. Neste momento, um novo *target-z* será computado, seguindo o critério explicado acima.

A eficiência do algoritmo ZSweep vem:

- do fato do algoritmo explorar a ordem global implícita aproximada que a ordenanação em z, dos vértices, induz nas células incidentes a eles; portanto, apenas uma pequena quantidade de interseções é encontrada fora de ordem;
- do uso de composição adiantada, que mantém baixa a necessidade de memória do algoritmo e evita o custo da ordenação de todas as interseções para cada pixel.

2.3.5 O Algoritmo RZSweep

O algoritmo RZSweep [5, 21] utiliza o mesmo princípio do ZSweep para executar a renderização do volume através da varredura exclusivamente em dados regulares. Neste algoritmo, o plano de varredura faz a triagem do evento que será computado, posicionando sobre determinado vértice do volume.

A computação é reduzida, pois somente é realizada quando o plano de varredura ultrapassa o vértice v do volume correspondente ao evento localizado. As principais vantagens desta abordagem aplicadas a malhas regulares, são:

- A triagem do evento está relacionado à ordem de profundidade do vértice, reduzindo o problema da terceira para a segunda dimensão;
- A coerência espacial dos dados é mantida;
- Não existe exigência de informações de conectividade entre células, pois a localização dos vértices é implícita

A Figura 2.13 descata o funcionamento do algoritmo RZSweep. No passo 1, o aLgoritmo irá ler o dado volumétrico, que ficará em memória. Este dado será acessado conforme o plano de varredura vai avançando, para computar as faces incidentes ao vértice ultrapassado pelo plano.

RZSweep

1. Ler o conteúdo do dado proveniente de um único arquivo binário, em que cada valor escalar ocupa um byte;
2. Iniciar a renderização pelo vértice mais próximo (menor valor de z) que é um dos oito vértices do bounding box do volume;
3. Enquanto a Heap não está vazia {
    * Recuperar o próximo vértice da heap que será o vértice corrente;
    * Marcar o vértice corrente como vértice varrido (*swept*);
    * Determinar os vértices adjacentes que defines as novas faces incidentes ao vértice corrente;
    * Se e somente se os vértices ainda não foram enviados (*not sent*)
        - Enviar os vértices adjacentes para a heap;
        - e configurar como enviados (*sent*)
    * Projetar as novas faces somente se todos vértices das faces estiveram marcadas como não enviadas (*not swept*)

}

Figura 2.13: Pseudo-codigo do RZSweep.

No passo 2, o algoritmo seleciona o vértice mais próximo, considerando a menor distância entre a profundidade z do vértice e o plano de varredura. Inicialmente, o vértice mais próximo é selecionado com base em um dos oito vértices do *bounding box* do volume.

No passo 3, há o laço de renderização do algoritmo. A ordenação das coordenadas z para o caminho do plano de varredura, é realizada pelo algoritmo de ordenação por seleção, chamado *HeapSort*, ou simplesmente *heap*. Cada vértice a ser percorrido pelo plano de varredura é recuperado da *heap* até que ela esteja vazia. No pior caso, a *heap* tem complexidade *O(n log(n))*. Este laço é dividido em algumas etapas.

Na primeira etapa, o objetivo é recuperar o próximo vértice que será computado. Como a busca é feita utilizando a *heap*, com certeza ele será o vértice mais próximo. Na segunda etapa, o vértice corrente é marcado como varrido (*sweept*), a fim de

seja minimizado o tamanho da *heap*. Pois, nesta estrutura, só os vértices adjacentes ao vértice corrente são inseridos. Na próxima etapa, os vértices adjacentes são determinados e inseridos na *heap*. Isto facilitará a seleção dos vértices a serem computados posteriormente. Com a determinação deste vértices, as faces incidentes ao vértice corrente podem ser projetadas.

Este algoritmo sequencial, processa um volume de dados que necessariamente deve caber na memória principal do computador. Caso isto não ocorra, o gerenciamento da memória é totalmente deixado a cargo do Sistema Operacional. Neste caso, o custo computacional aumenta bastante, devido às interrupções do Sistema Operacional, para operações de Entrada/Saída necessárias para a paginação de memória.

Capítulo 3

Renderização *Out-of-Core*

Neste capítulo são apresentados as recentes pesquisas em renderização *out-of-core* de dados volumétricos. São relatados ainda, os principais problemas em renderizar grandes massas de dados, bem como as soluções propostas para este tipo de implementação.

3.1 Renderização de Grandes Massas de Dados

O principal problema em manipular uma grande massa de dados é o gerenciamento de memória. O sistema operacional do computador é o responsável por este gerenciamento. Uma das técnicas mais sofisticadas e poderosas de gerenciamento de memória é chamada Memória Virtual. Através desta técnica as memórias principal e secundária são combinadas, dando ao usuário a ilusão de existir uma memória muito maior que a capacidade real da memória principal.

Este conceito fundamenta-se em não vincular o endereçamento feito pelo programa, dos endereços físicos da memória principal. Desta forma, programas e suas estruturas de dados deixam de estar limitados ao tamanho da memória física disponível, pois podem possuir endereços associados à memória secundária (disco rígido).

Existe um forte relacionamento entre a gerência de memória virtual e a arquitetura de hardware do sistema computacional. Por motivos de desempenho, é comum

que algumas funções de gerência de memória virtual sejam implementadas diretamente no hardware. Além disso, o código do Sistema Operacional deve levar em consideração várias características específicas da arquitetura, especialmente o esquema de endereçamento do processador.

O sistema operacional utiliza alguns algoritmos para verificar qual a melhor política de gerenciamento, levando em conta diversos fatores, como a arquitetura de hardware e o tamanho da memória do sistema.

Esta implementação, em sua grande maioria, faz com que ocorram buscas de dados no disco e a sua posterior carga na memória principal. Dependendo do tipo de arranjo encontrado pela política do sistema operacional, e do tamanho do dado a ser manipulado, o custo computacional pode se tornar alto.

Apesar da política do sistema operacional ser eficiente, ela não aproveita a coerência espacial dos dados acessados na memória, por desconhecer o padrão de acesso ao dado da aplicação. Engler [22] sugere em sua pesquisa que alguns processos de usuário realizem seu próprio gerenciamento de memória. E pesquisas nesta área já são realizadas, por exemplo, em aplicações multimídia [23] e comunicadores pessoais portáteis [24].

O termo *out-of-core* refere-se à solução propostas por algoritmos que implementam a capacidade de renderizar dados que são grandes demais para caber na memória principal de um computador. Tais algoritmos devem ser otimizados para buscar eficiência e ter acesso aos dados armazenados na memória secundária, como o disco rígido *(Hard Disk)*, por exemplo. As vantagens desta implementação são:

- Utiliza menos chamadas de sistema (*system calls*) ou interrupções do Sistema Operacional;
- Aproveita a coerência espacial dos dados, pois a carga na memória é realizada por demanda, de acordo com o conteúdo correspondente à região do volume que está sendo analisada;

No carregamento sob demanda, o algoritmo deve selecionar a porção do dado no disco, referente ao volume pesquisado e carregar somente este conteúdo. Como a maioria das operações do algoritmo estão relacionadas a dados adjacentes, menos acesso a disco serão necessários por parte do Sistema Operacional.

Recentes pesquisas estão sendo conduzidas na área de renderização *out-of-core*. Lindstrom [25, 26], propõe um algoritmo *out-of-core* para a simplificação de grandes modelos poligonais, utilizando agrupamento de vértices, com o objetivo de atenuar os problemas de renderização, e permitir uma visualização de qualidade.

No trabalho proposto por Yu [27], a cena a ser renderizada é particionada hierarquicamente, a fim de permitir a computação contínua a nível de detalhe hierárquico *(hierarchical level of detail (HLOD))*. Aplicações como o *Parallel iWalk* [28], que utilizam modelo de comunicação Cliente/Servidor, também se beneficiam de algoritmo *out-of-core* para a visualização de dados muito volumosos.

Deste modo, a renderização *out-of-core* torna-se importante vantagem do algoritmo implementado por este trabalho, por ser a melhor opção em termos de desempenho. A seção seguinte apresenta como o acesso por demanda ao dado pode ser realizado.

3.2 Divisão Hierárquica

O algoritmo de renderização out-of-core, geralmente utiliza um processo de divisão hierárquica no espaço do objeto. Este processo pode ser realizado na fase de pré-processamento, permitindo que mais tarde, o algoritmo possa acessar as partes subdivididas do dado, através de uma estrutura de dados específica. Um caso particular de subdivisão espacial chamado *octree*, ou seja, árvore com oito filhos, é frequentemente utilizado para este fim.

Alguns trabalhos de pesquisa em renderização *out-of-core* utilizam o método de divisão hierárquica em *octrees*. Por exemplo, no trabalho de renderização proposto por Horng-Shyang [29], a simulação do espaço é feita utilizando octrees, onde ele implementa diversas técnicas para a simulação eficiente de nuvens e sua rápida renderização.

Os dados são armazenados hierarquicamente na *octree*, sendo o nó raiz correspondente à caixa envolvente do volume. Cada nó da octree tem uma relação de hierarquia com oito nós de nível inferior (filhos), sendo cada um destes, correspondente a uma nova subdivisão do dado de nível superior (pai).

Ralf e Heinrich [30] também, ao introduzirem o conceito de espaço de modelagem virtual infinito baseado em (*voxels*) ilimitados, representam a superfície da malha em uma *octree* para permitir a rápida renderização. Eles utilizam um paradigma de modelagem adaptativa cuja representação é uma hierarquia de *voxels*. O objetivo é utilizar a divisão hierárquica para acelerar o processo de busca aos dados que serão renderizados.

No método de utilizado em [26], Lindstrom também utiliza uma *octree* como uma estrutura de dados *out-of-core* multi-resolução. Ainda em outro trabalho recente, Cignoni et al. [31] propõem um algoritmo de simplificação incremental *out-of-core* subdividindo o espaço em uma *octre*.

A subdivisão no espaço do objeto em formato *octree* privilegia a busca, pois a complexidade do algoritmo é *O(log(n))*. É interessante saber qual a porção do dado se deseja manipular *(bounding box)*, para que esta possa ser carregada na memória. Na renderização, os pixels de regiões da tela próximas, geralmente, acessarão as mesmas informações da folha da *octree* correspondente, o que permite aproveitar a coerência espacial do dado.

A divisão hierárquica restringe o acesso ao dado necessário, permitindo que seja feito um carregamento sob demanda. Este processo evita ao máximo as chamadas ao sistema operacional, pois restringe os acessos aos conteúdos das folhas da octree, que podem estar armazenadas sequencialmente no disco.

Com o uso eficiente em termos de gasto de memória, o custo computacional é minimizado, ainda que o algoritmo seja sequancial. Porém, o algoritmo sequancial não dá conta de uma grande massa de dados regulares, como foi explicado na Seção anterior(3.1), pelo fato desta ser maior do que a memória disponível no computador.

A fim de possibilitar a renderização do dado sob demanda e aumentar a performance, este trabalho propõe, a implementação de uma solução de processamento paralelo, interconectando estações de trabalho básicas (*Personal Computer - PC*). Esta estrutura forma um sistema multicomputador sem compartilhamento de memória, chamado *cluster*.

O próximo capítulo descreve detalhes da implementação deste trabalho, chamado PRZSweep-OOC. Operando com dados, exclusivamente, regulares, o PRZSweep in-

corpora as funcionalidades *out-of-core* e o paralelismo do processamento da renderização volumétrica, ao ambiente do RZSweep original.

Capítulo 4

O Algoritmo *PRZSweep-OOC*

O presente capítulo, descreve os detalhes da implementação do algoritmo PRZSweep-OOC usado no desenvolvimento deste trabalho. A estrutura principal do algoritmo é baseada no algoritmo RZSweep original, descrito na seção 2.3.5.

4.1 Pré-processamento

No PRZSweep-OOC, foi implementada a renderização *out-of-core*. Para tanto, este algoritmo utiliza o processo de divisão hierárquica *octree*. Esta divisão hierárquica é realizada na fase de pré-processamento, onde a caixa envolvente *(bounding box)* do dado volumétrico, é dividida ao longo dos eixos ortogonais, em oito partes iguais.

Na estrutura *octree* os *bounding boxes* de cada subdivisão do volume, são armazenados na estrutura em formato de árvore, veja a Figura 4.1. Por esta definição, os nós de menor hierarquia na *octree* (folhas) serão a referência direta a uma determinada porção do dado volumétrico em questão. Dependendo no nível da *octree*, as folhas poderão conter mais ou menos informação do dado volumétrico, pois o nível da árvore controla a quantidade de subdivisão nos três eixos.

A importância deste pré-processamento está na posterior necessidade do PRZSweep-OOC acessar determinada porção do volume quando estiver renderizando a imagem. Para isso, somente a folha da *octree*, correspondente a esta porção do volume será

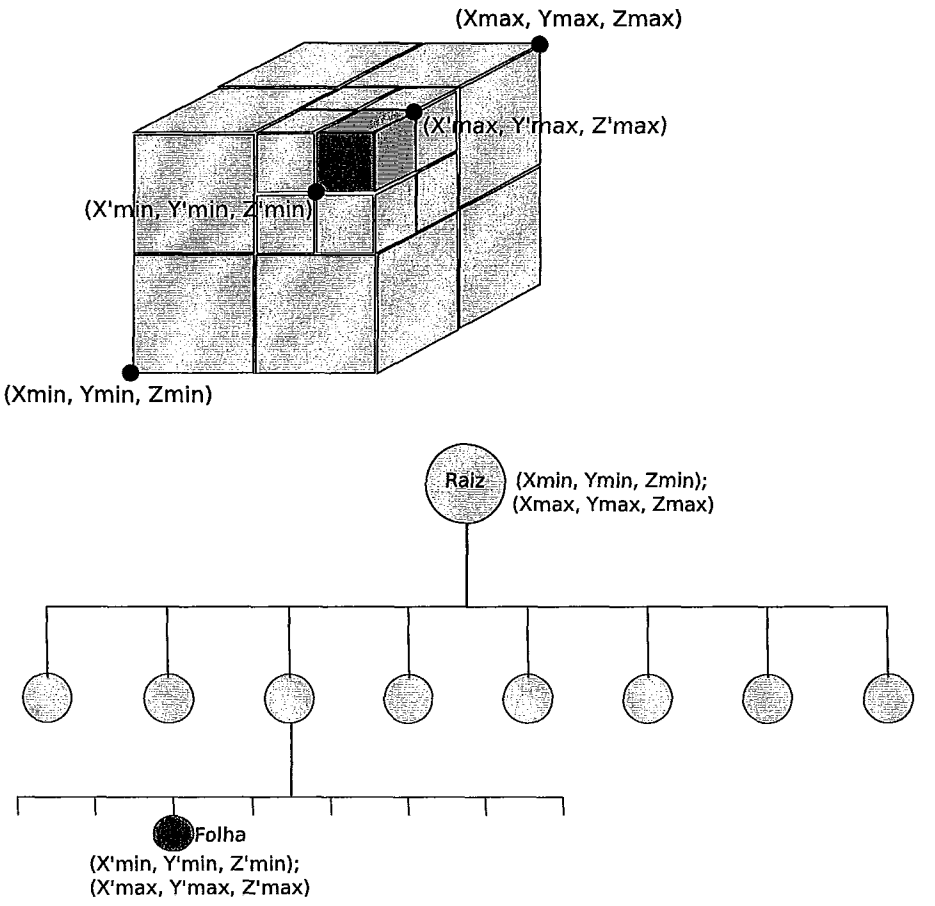


Figura 4.1: Divisão hierárquica em formato Octree

carregada na memória.

Na busca, o PRZSweep-OOC identifica o número da folha e acessa diretamente o bloco de dado, a partir do disco. A coerência espacial do dado será aproveitada, à medida que os *pixels* vizinhos, geralmente acessam as mesmas folhas da *octree.*

Observe a Figura 4.2, onde as folhas da *octree* que intersectam a região desejada estão marcadas. Repare que somente as quatro folhas marcadas contribuem com o cálculo de cor e opacidade final dos *pixels* naquela direção.

Neste modelo, somente as folhas da octree precisam ser carregadas na memória para permitir o acesso ao dado, pois estas contêm o *bounding box* daquela região. Desta forma, a divisão do dado em octree permite o acesso ao dado sob demanda. As folhas da octree são numeradas de acordo com a região a que representam, observe a Figura 4.3.

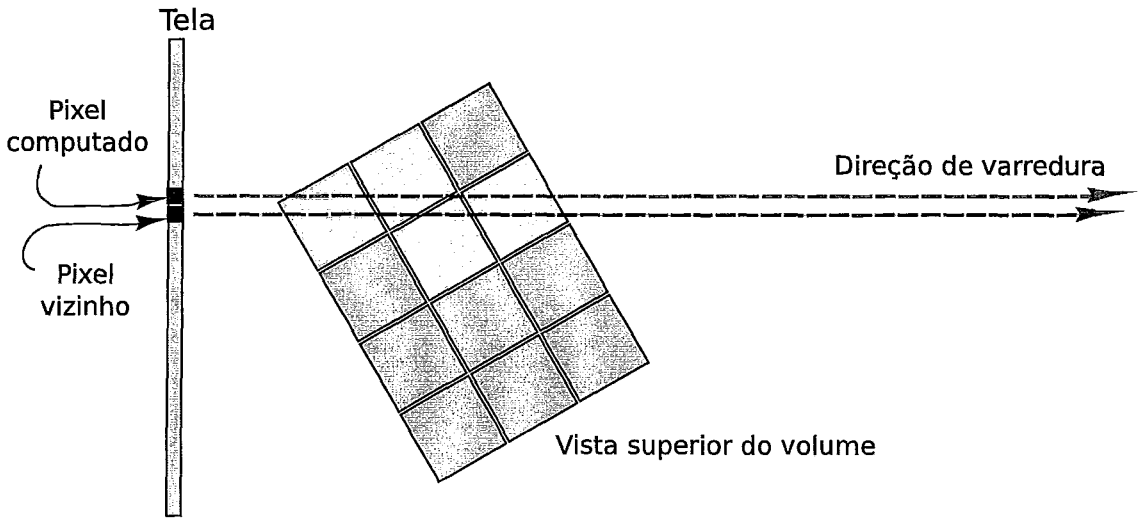


Figura 4.2: Vista superior de um dado volumétrico. Na renderização dos pixels vizinhos, as mesmas folhas da Octree serão acessadas

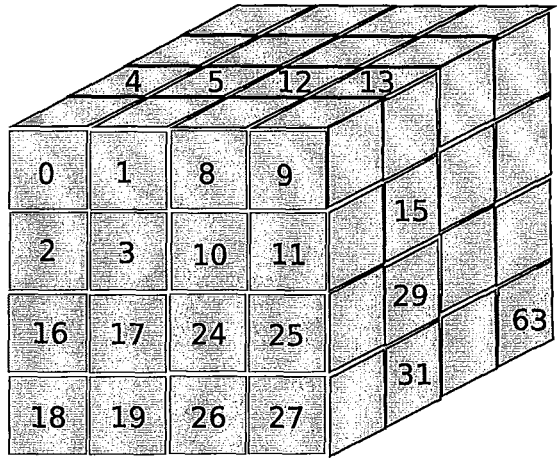


Figura 4.3: A Numeração das 64 folhas de uma Octree com 2 níveis de subdivisão.

Após identificar as folhas, é possível acessá-las realizando uma busca rápida na estrutura da *octree*, pois o algoritmo de busca para as folhas da *octree* tem complexidade $O\ (log(n))$.

A estrutura da *octree* é mantida durante todo o processo de renderização para facilitar a busca do dado a ser carregado na memória. Entretanto, ainda na fase de pré-processamento esta estrutura é utilizada para gerar uma cópia do arquivo binário que contém o dado volumétrico. Esta cópia deve ser gerada para permitir a organização do dado volumétrico na ordem exata das folhas da *octree*.

Com a reorganização do arquivo segundo a *octree*, cada folha fica armazenada sequencialmente e, consequentemente, as informações referentes ao *bounding box* correspondente.

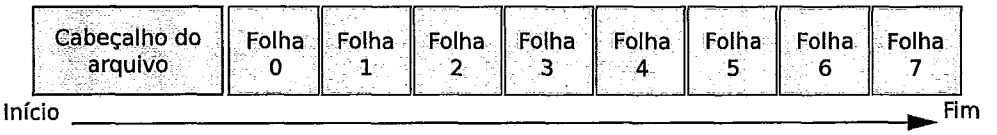


Figura 4.4: Organização do arquivo binário por ordem das folhas da Octree. Após o acesso rápido, somente a porção do dado referente à folha, é carregada na memória do sistema

Seguindo este princípio, o acesso ao conteúdo referente à folha no disco é facilitado, apenas calculando o deslocamento do ponteiro no arquivo. Este deslocamento é calculado, naturalmente, observando o número da folha e o tamanho em *bytes* do seu conteúdo. Como se trata de um dado regular, todas as folhas têm o mesmo tamanho. Com isto, os dados referentes às regiões do volume armazenadas nas folhas da *octree* serão gravados sequencialmente armazenadas no disco; veja a Figura 4.4.

4.2 Inicialização de Estruturas Básicas

Na fase de inicialização, o PRZSweep-OOC prepara as estrutudas de dados auxiliares que serão utilizadas na renderização volumétrica *out-of-core*. O funcionamento do algoritmo é baseado na abordagem do plano de varredura. O uso destas estruturas são necessários ao controle do percurso do plano de varredura, bem como da computação da informação de cada vértice percorrido.

Assim como em outras abordagens, o plano de varredura avança em direção à profundidade do volume varrendo um vértice de cada vez. Vértices já ultrapassados pelo plano de varredura e projetados são marcados como varridos (*swept*).

Quando o plano de varredura toca um vértice, o algoritmo projeta todas as faces incidentes no vértice. Os vértices destas faces incidentes só podem ser definidos por vértices ainda não-varridos (*non-swept*) pelo plano de varredura; veja a Figura 4.5.

Inicialmente, o algoritmo verifica as faces incidentes e determina as que podem ser projetadas. Cada vértice interno, tem doze faces incidentes; veja a Figura 4.6. Entre

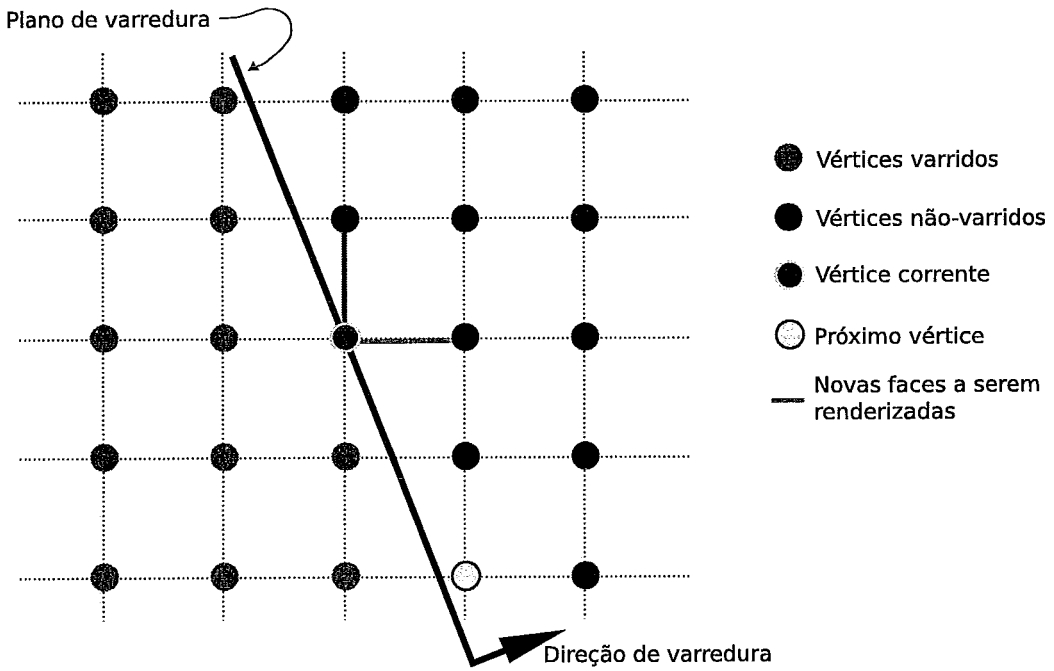


Figura 4.5: O plano de varredura avançando em direção à profundidade

estas faces, somente as que estão localizadas além do plano de varredura devem ser projetadas. Estas faces, que serão correntemente renderizadas, são chamadas novas faces; compare as figuras 4.6 e 4.7.

O uso da estrutura *heap* visa controlar a seleção dos vértices pela sua ordenação e relação ao eixo z, que servirá de caminho para o plano de varredura.

Cada vértice a ser percorrido pelo plano de varredura é recuperado da heap até que ela esteja vazia. No pior caso, a heap tem complexidade $O(n\ log(n))$.

O algoritmo começa inserindo na heap somente o vértice mais próximo (com menor coordenada z), entre os oito vértices da caixa envolvente da grade implícita (*bounding box*). Os demais vértices seguintes são inseridos sob demanda, minimizando o tamanho necessário da heap.

Além dos dados em si, o algoritmo requer espaço em memória para manter a heap e algumas variáveis de controle, chamadas *flags*. Estes flags são utilizados para controlar o processo de varredura. Além disso, é mantida em memória a *octree* que irá controlar o acesso ao dado no disco, sob demanda.

O tamanho da *heap* pode ser previsto considerando a maior diagonal do volume

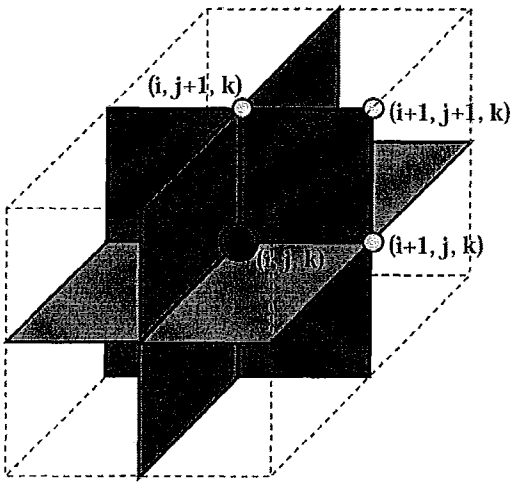


Figura 4.6: Doze faces incidentes ao vértice interno

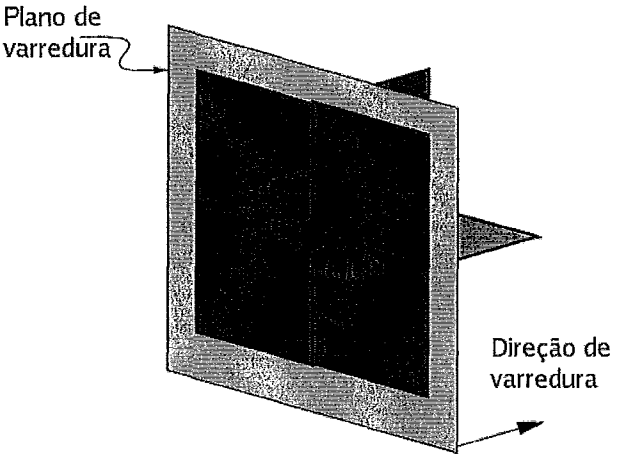


Figura 4.7: As faces que estão à frente do plano de varredura

de dados, pois esta pode ser a maior distância a ser percorrida pelo plano de varredura, no pior caso. Isto depende da orientação do volume em relação ao plano de varredura.

Este tamanho pode ser considerado apenas como mais uma fatia de dados do volume, o que não é significativo. Isto vem do fato que apenas os vértices adjacentes que estão à frente do plano de varredura, estarão na *heap*.

Quanto aos flags, foi tomado o cuidado de utilizar somente bits de controle, a fim de não desperdiçar memória para desnecessária alocação.

Antes de iniciar o processo de renderização, quatro etapas de inicialização ainda

são realizadas. A primeira preocupação do algoritmo é criar uma representação implícita de uma grade (*grid*) para a malha regular do dado volumétrico. Um sistema de coordenadas do mundo é construído, cuja representação é inicialmente feita por três vetores unitários u_0, u_1 e u_2 correspondente às direções dos eixos x, y e z.

Em seguida, a matriz de transformação afim contendo as rotações solicitadas, é criada e aplicada aos vetores do sistema de coordenadas do mundo. Deste ponto em diante, qualquer vértice (i,j,k) da grade implícita pode ser representado no espaço do mundo pela projeção, usando o produto escalar, das coordenadas implícitas do vértice com cada um dos vetores unitários.

Em terceiro lugar, o volume é totalmente analisado e cada vértice que se encontra dentro da faixa de valores escalares solicitados (*threshold*) são marcados com um flag para acelerar o processo de classificação. A quarta etapa, detalhada a seguir, determina quais faces incidentes em um vértice podem ser projetadas.

Os vértices internos tem doze faces incidentes, conforme pode ser visto na Figura 4.6. Considere que o dado tem dimensões Dim_X, Dim_Y e Dim_Z, e o vértice interno v tem índices *(i,j,k)*. Ao adicionar ou subtrair uma unidade dos índices de v, é possível determinar todos os seus 26 vértices vizinhos, o que juntamente com v, determinam as doze faces incidentes. Dentre estas faces, as que serão projetadas, estão localizadas à frente do plano de varedura. No pior caso, as oito faces adjacentes a v podem ser projetadas, conforme a Figura 4.7. No melhor caso, somente três faces serão projetadas, conforme ilustrado na Figura 4.8.

Considerando as coordenadas do vértice que está sendo analisado, os quatro vértices de cada face que poderá ser projetada, deverão ter coordenadas maiores ou iguais a esta referência. Tais faces farão parte de uma lista, a fim de permitir rápida verificação no processo de varredura.

4.3 O Processo de Renderização

A Figura 4.5 mostra o plano de varredura tocando o vértice corrente (*current vertex*). Os dois vértices não-varridos ainda pelo plano devem ser enviados para a heap e

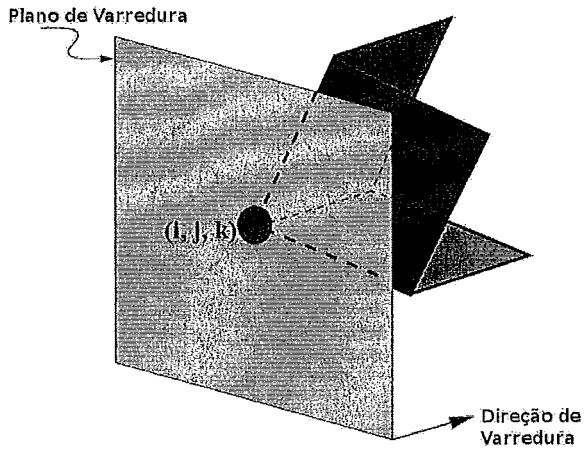


Figura 4.8: No melhor caso, somente três faces incidentes são projetadas

marcados como não-varridos. Isto permite o plano continuar caminhando. As duas novas faces são incidentes ao vértice corrente e estão localizadas à frente do plano de varredura.

A fim de evitar várias inserções na heap, o algoritmo envia somente os vértices adjacentes que ainda não tenham sido enviados antes. Além disso, para envitar múltiplas projeções de determinada face, é verificado se um outro vértice que define a face (exceto o vértice corrente), já foi projetado, isto é, varrido (*swept vertex*).

A classificação sob demanda é realizada à medida que, somente os vértices que estão dentro do *threshold* solicitado, são enviados para a heap. Ao marcar os vértices na etapa de inicialização, a classificação pode ser feita de forma mais eficiente.

4.3.1 Projeção das Faces e Cálculo de Iluminação

A projeção das faces só pode ser realizada, se todos os vértices das faces adjacentes ao vértice corrente, estiverem marcados como não enviados *(not sweept)*. Neste momento são efetuados os cálculos de cor e opacidade para o pixel. Valores diferentes para cor e opacidade, produzem um certo grau de realismo no resultado final. Esta seção trata das abordagens utilizadas para implementação das funções de transferência destes atributos.

Cada elemento do volume de dados, representa um único valor escalar. Não

há informação de cor e opacidade. Esta informação é explicitamente associada, implementando um algoritmo que calcula a integral de iluminação baseado em um modelo simplificado [32, 33, 34].

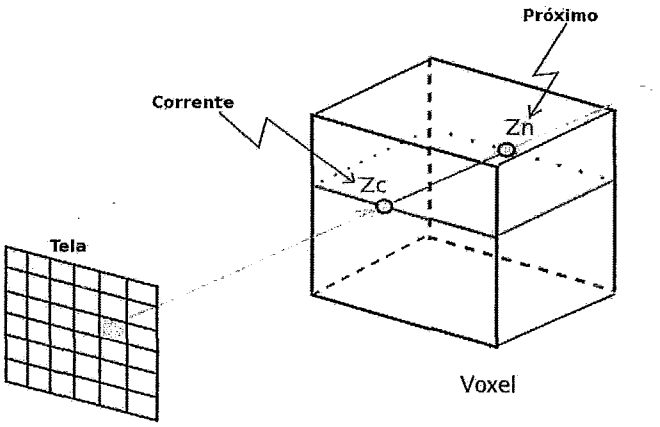


Figura 4.9: Modelo simplificado da integral de iluminação.

Nesse modelo simplificado, o raio de visão (*viewing ray*) é considerado, e a integral depende apenas da distância percorrida dentro da célula l (*length*), chamada de espessura (*thickness*) da célula, e os valores de entrada Z_c (*current Z*) e saída Z_n (*next Z*) do raio. Como a malha é regular, todas as células tem uma unidade de espessura. O resultado da integral é então a parcela de contribuição da iluminação no pixel pela célula.

No modelo simplificado, a integral é reduzida a duas simples equações, como expressado por Shirley e Tuchman [32]:

$$C = \frac{C(Z_c) + C(Z_n)}{2} \tag{4.1}$$

$$\alpha = 1 - e^{-\frac{\tau(Z_c)+\tau(Z_n)}{2} l} \tag{4.2}$$

Na Equação 4.1 a cor C é computada pela média das cores de entrada $C(Z_c)$ e saída $C(Z_n)$. Na Equação 4.2 o cálculo da opacidade α é baseado no coeficiente de extinção (*extinction coefficient*) τ, que mede quanto de luz é absorvida pela célula. Da mesma forma que as cores, o coeficiente de extinção depende da entrada $\tau(Z_c)$ e saída $\tau(Z_n)$ da célula. As cores e os coeficientes de extinção são associados aos

valores escalares de entrada Z_c e saída Z_n por uma função chamada: **função de transferência** (*transfer function*) [15].

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

$$C_{i+1} = \alpha_i \, C_i + (1 - \alpha_i) \, C_{i-1} \tag{4.3}$$

$$\alpha_{i+1} = \alpha_i + \alpha_{i-1} \tag{4.4}$$

Note que, somente a opacidade da célula corrente α_i é usada na regra da Equação 4.3. A opacidade da célula já computada α_{i-1} é usada na Equação 4.4 para o cálculo da opacidade resultante α_{i+1}. O par (C_{i-1}, α_{i-1}) corresponde ao valor $RGBA$ do fragmento da célula $i-1$, que combinado ao fragmento da célula i gera a cor do pixel.

4.3.2 Rasterização

Após estabelecer a cor final do pixel, o plano de visão é atualizado com estas informações. Este processo ocorre a cada interação do processo de varredura, e é conhecido como *rasterização*. Na rasterização é importante notar que o volume que está sendo renderizado está centralizado no plano de visão.

Cada folha da *octree* será renderizada em separado, conforme a demanda, mas o posicionamento final desta parte do volume deverá ser ajustado com a posição da folha em relação ao volume. Isto significa que, ao informar a porção do volume que será renderizada, o PRZSweep-OOC não irá centralizá-la no plano de visão, sob pena de obter uma imagem desfigurada, cujas dimensões serão referentes à largura e altura do volume da folha da *octree*. O exemplo ilustrado na parte esquerda da Figura 4.10 indica que todas as folhas foram renderizadas na mesma posição.

Para rasterizar a folha na posição correta relativa ao volume, o algoritmo implementa um deslocamento do centro desta porção do volume. O fator de deslocamento é dependente da quantidade de níveis da *octree*. A partir desta coordenada, deslocada do centro do volume, a folha será rasterizada. Desta maneira pode-se garantir

que a rasterização será realizada na posição correta, pois aquela coordenada refere-se ao centro do dado relativo à folha da *octree*, conforme a parte direita da Figura 4.10.

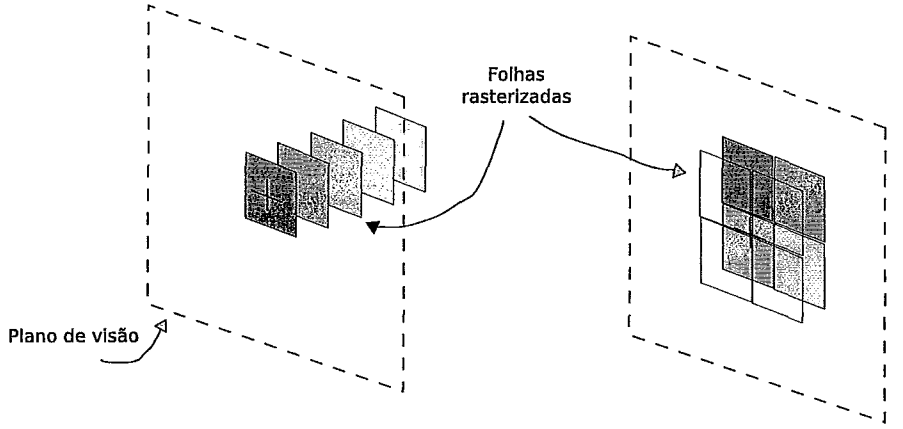


Figura 4.10: Rasterização das folhas na posição central (esquerda) e na posição correta do plano de visão (direita)

4.4 Paralelização do Algoritmo

Com a renderização *out-of-core*, o PRZSweep-OOC faz a renderização de grandes massas de dados, resolvendo o problema de limitação de tamanho de memória em cada PC. A proposta deste trabalho, prevê uma solução de baixo custo e que seja ainda bastante eficiente. Sendo assim, a melhor opção foi a utilização de um *cluster* de PCs como arquitetura alvo do sistema. Esta implementação, permite a exploração da capacidade de processamento entre diversos PCs.

O objetivo na utilização do *cluster* é enviar mensagens em um tempo na escala de microssegundos – em vez de acessar a memória compartilhada em um tempo na escala de nanossegundo -, sendo assim mais simples, barato e fácil de implementar. Neste ambiente, o algoritmo foi implementado para tirar proveito do ambiente em cluster, não sendo transparente como nos sistemas multiprocessadores. Para tanto, existem pacotes voltados para facilitar o desenvolvimento de aplicações paralelas em *clusters*, como o *MPI (Message Passing Interface)* que foi utilizado na implementação deste trabalho.

MPI é uma biblioteca de funções que estabele um padrão para troca de mensa-

gens entre os processadores, e está disponível virtualmente em todas as plataformas. Isto faz com que *MPI* tenha alta portabilidade. Outro fator preponderante na escolha desta biblioteca, é a sua utilização em larga escala, e também por ter um bom desempenho em *clusters*.

Para a divisão de tarefas entre os processadores, o algoritmo implementa duas abordagens de paralelização. A primeira abordagem, explora a subdivisão da imagem em retângulos de tamanho fixo, denominados *tiles*. A segunda abordagem, se utiliza da subdivisão no espaço do objeto, através das folhas da *octree*. As duas abordagens são referenciadas neste trabalho, como: renderização por *tiles* e renderização por *folhas*, respectivamente.

A renderização por *tiles*, permite não só a divisão das tarefas a serem realizadas em paralelo, mas também a redução da quantidade de dados a serem trazidos para a memória por cada processador, motivando a implementação *out-of-core*.

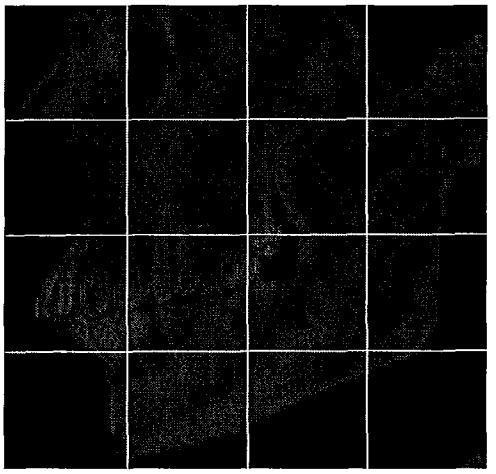


Figura 4.11: Imagem referente ao dado CT-Skull rotacionado em 90^o no eixo X, subdividida em 16 tiles iguais, dispostos em uma grade 4 x 4

Cada *tile*, recebe uma identificação única e representa uma porção da imagem que pode ser computada independentemente. Portanto, pela identificação de cada *tile* é possível determinar qual porção da imagem deverá ser computada.

A idéia da paralelização é dividir os *tiles* para os diversos processadores do *cluster*. A computação de um *cluster* gera uma subimagem que deve ser unida às outras subimagens geradas, para formar a imagem final. A Figura 4.11 mostra a subdivisão

no espaço da imagem.

As mensagens entre os processadores contêm a estrutura atualizada do plano de visão. Entretanto, como cada processador renderiza a imagem contida em um *tile*, somente aquela região do plano de visão será interessante de compartilhar. O PRZSweep utiliza o processo 0 como mestre, no processo de renderização. O processo mestre é o responsável por reunir as imagens que forem renderizadas individualmente para cada *tile*, por cada processador.

A implementação com *MPI* exige que seja feito um sincronismo por parte do algoritmo, já que a sua estrutura é assíncrona. No momento em que os processos terminam a renderização, cada um envia uma mensagem para o processo mestre. O processo mestre por sua vez, ativa uma primitiva de recebimento que bloqueia o seu próprio processamento, até que tenha recebido todas as mensagens. Na sequência, o processo mestre salva em disco o conteúdo do plano de visão, composto pelos *tiles* renderizados pelos demais processos.

Na renderização por *folhas*, cada folha da *octree* é renderizada por um processo distinto, independente da divisão da imagem em *tiles*. Com a divisão em *tiles*, é possível fazer a renderização do volume de maneira eficiente. Porém, existe a possibilidade de uma folha da *octree* ser intersectada por mais de um *tile*. Neste caso, mesmo que somente a porção do volume pertencente ao *tile* seja renderizada, toda a folha deve ser carregada na memória. Com a divisão de tarefas no espaço do objeto, onde cada processador é responsável pela renderização de uma ou mais folhas específicas da *octree*, este problema não ocorrerá. Desta forma, a minimização do esforço e a velocidade de processamento podem ser garantidas.

Cada folha recebe uma identificação que está associada ao processo que irá renderizá-la. Quando o processo inicia o a renderização, cada folha é carregada uma única vez, e a combinação do resultado final também é realizada pelo processo mestre.

No propósito de reduzir o tempo de execução e melhorar a performance, a combinação das duas abordagens são necessárias. Os resultados obtidos com estes experimentos estão descritos no próximo capítulo.

Capítulo 5

Resultados

No capítulo anterior foi descrito a implementação do algoritmo PRZSweep-OOC. O código-fonte foi escrito em C++ usando OpenGL 2.0 [35, 36, 37] em Linux. A implementação em paralelo foi executada em um cluster com 8 máquinas *dual core* Intel Pentium IV 3.6 GHz com 2 GB RAM. O paralelismo foi implementado a partir do uso da biblioteca de passagem de mensagens *MPI*.

Os dados volumétricos utilizados nos testes do PRZSweep-OOC foram: *ctLobster*, *ctLobster* +, *mriHead*, *ctSkull*, *ctBonsai* + e *ctBrain* do site Volvis [38], todos com malhas regulares dispostos em grades retilíneas. O dado com nomenclatura iniciada por *mri* foram adquiridos por Ressonância Magnética e os dados iniciados por *ct*, foram adquiridos por Tomografia Computadorizada. Os dados marcados com sinal (+) foram sinteticamente modificados por subdivisão de amostragem, com o objetivo de aumentar o tamanho e melhor condicionar os testes de renderização em cluster.

A Tabela 5.1, registra os detalhes de cada dado volumétrico utilizado na obtenção dos resultados. Todos os dados foram renderizados no intervalo de valores escalares (*threshold*) entre 40 e 80. Todos os tempos de renderização foram obtidos de imagens de tamanhos iguais a 800×800 pixels, exceto quando especificado outro tamanho. Os tempos são dados para cada transformação geométrica aplicada ao volume. Portanto, inicialmente é aplicada uma rotação no dado, para depois ser

realizada a renderização do volume. O desempenho do algoritmo PZSWeep-OOC é descrito em segundos (s) de acordo com a quantidade de processadores utilizada: 1 (sequencial), 2, 4 e 8. Todos os resultados foram obtidos considerando apenas dois níveis de subdivisão da *octree*.

Dados	Tamanhos	# Vértices (M)
ctLobster	301 x 324 x 56	5,21
mriHead	256 x 256 x 111	6,94
ctLobster +	602 x 324 x 56	10,42
ctSkull	256 x 256 x 256	16
ctBonsai +	512 x 256 x 256	32
ctBrain	512 x 256 x 512	64

Tabela 5.1: Tamanho e número de vértices do volumes

As duas abordagens de renderização volumétrica implementados pelo PZSweep-OOC dependem de uma configuração inicial. As configurações sempre privilegiam o número de processadores. Portanto, na renderização por *tiles*, o número máximo de *tiles* é igual ao número de processadores. Na abordagem por *folhas*, cada processador recebe uma ou mais folhas da *octree* para renderizar.

A Figura 5.1 ilustra os tempos obtidos na renderização inicial por *tiles*, considerando o primeiro nível de subdivisão da *octree*. O gráfico demonstra que há um gasto maior quando a tarefa é dividida em 8 processos, devido ao processamento redundante de folhas intersectadas por mais de um *Tile*. Entretanto foi observada um aumento gradativo da performance, quando do aumento dos processadores.

Utilizando o mesmo critério de avaliação, os dados foram submetidos à uma renderização por folhas. A Figura 5.2 mostra o gráfico de tempo decrescente quando o número de folhas a ser renderizada foi distribuída pelos processos. Como cada folha só é carregada uma vez, o aumento de performance é substancial, o que faz o algoritmo executar a renderização do volume com mais eficiência.

A Figura 5.3 mostra um gráfico comparativo para cada dado volumétrico renderizado por *tile* e por *folha* no primeiro nível de divisão espacial.

Ainda que o método de renderização por folhas obtenha melhor resultado final,

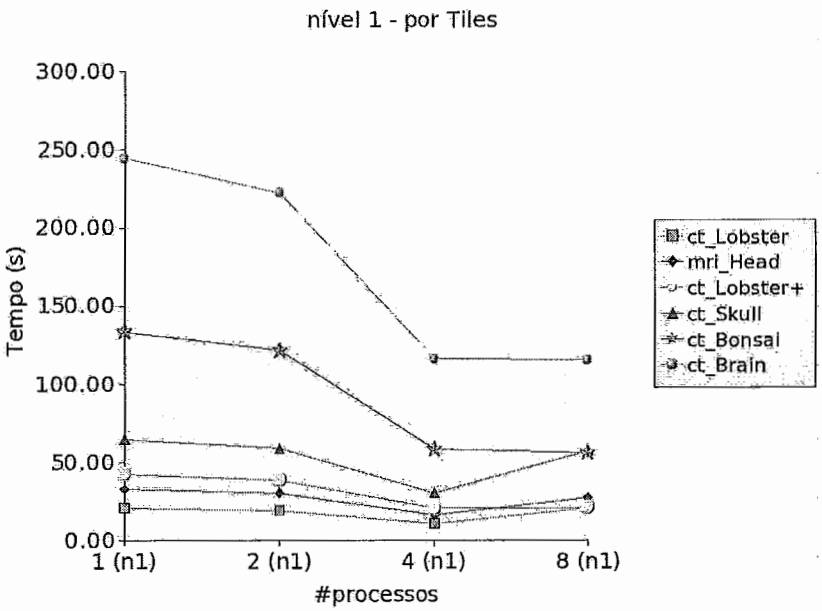


Figura 5.1: Renderização por Tiles com subdivisão de 1° Nível

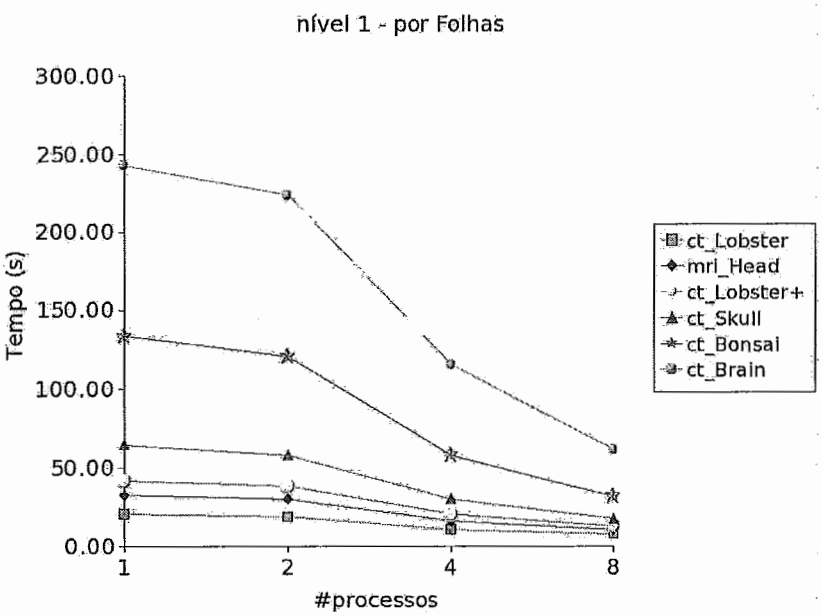


Figura 5.2: Renderização por Tiles com subdivisão de 1° Nível

para níveis de divisão espacial maior, esta tendência pode não ser uma hegemonia. Quando o número de subdivisões atinge níveis mais altos, o número de folhas para cada processo pode aumentar bastante. Neste caso, a renderização por *tiles* pode ser boa opção, desde que seja feita uma configuração adequada da distribuição de *Tiles*.

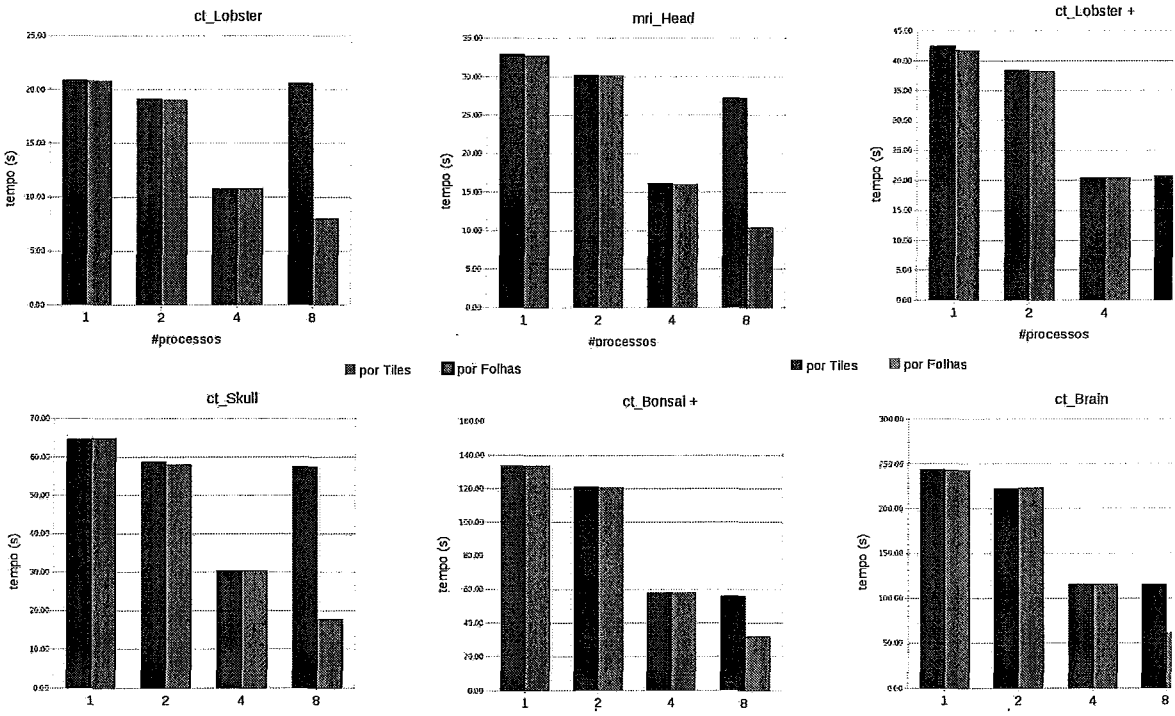


Figura 5.3: Comparação entre os diversos volumes renderizados por *Tiles* e Folhas.

A Figura 5.4 demonstra a evolução quando oito processos renderizam o volume, com nível de subdivisão 2, em velocidade próxima ao tempo obtido com nível 1.

A ilustração da Figura 5.5 revela a comparação do aumento de performance entre os dois tipos de renderização, com o aumento de níveis da *octree*. Note que a renderização por *tiles* tem um ganho de performance maior, proporcionalmente, do que a renderização por *folhas*. Isto se deve ao fato de que a redundância no processo de carga de folhas da *octree* é minimizado.

É interessante informar que a disposição dos *tiles* não afetou o tempo de renderização significativamente. Os tempos foram diferenciados em apenas centésimos de segundos. Os melhores tempos foram alcançados com as configurações adotadas que estão relacionadas na Tabela 5.2.

Algumas imagens geradas pelo PZSweep-OOC podem ser vistas nas Figuras a seguir:

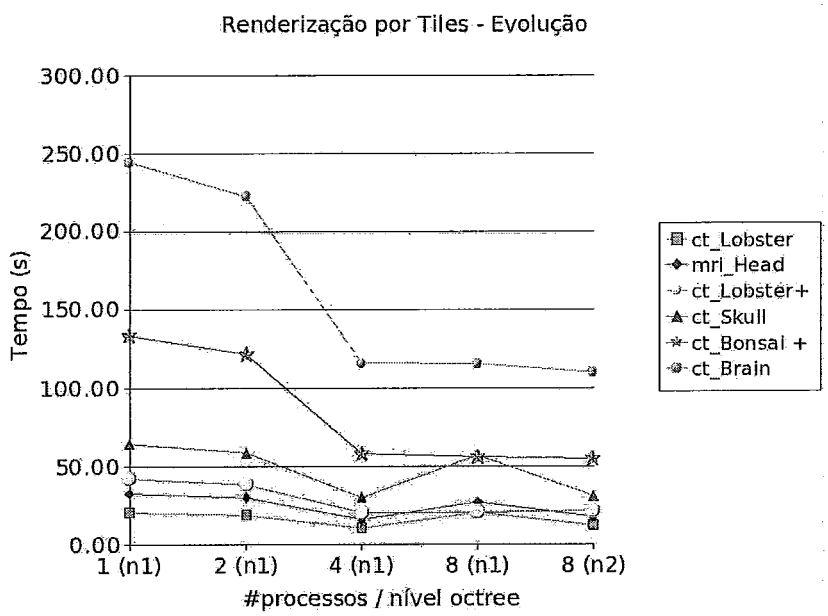


Figura 5.4: Comparação do aumento de performance na renderização por *tiles* com o nível 2 de subdivisão da *octree*.

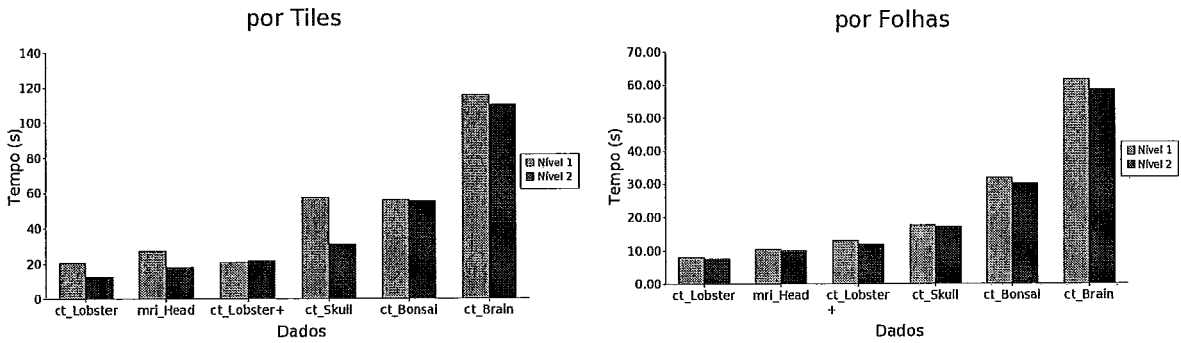


Figura 5.5: Com o aumento da Subdivisão da *octree*, a performance aumenta em ambos os métodos de renderização.

# Processos	Configuração dos Tiles
2	2×1
4	2×2
8	4×2

Tabela 5.2: Configuração Adotada para a Disposição dos Tiles.

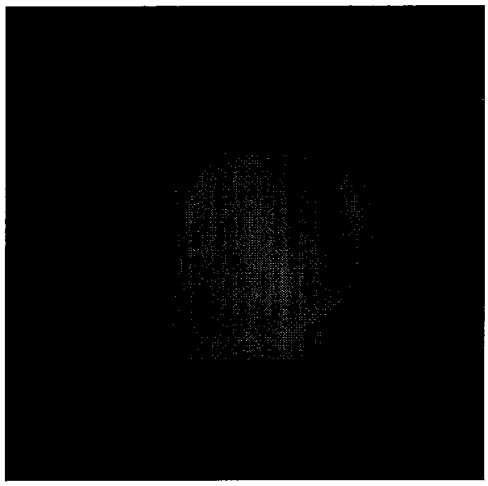


Figura 5.6: MRI Head

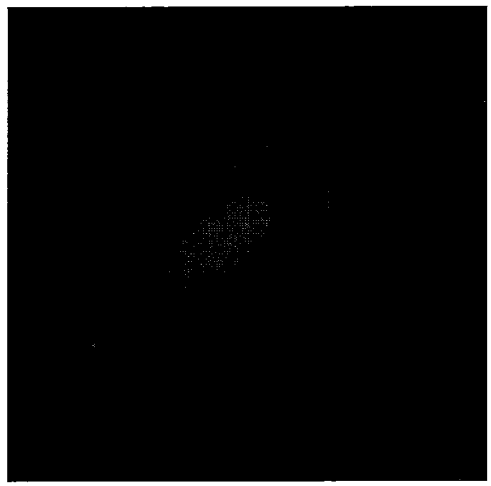


Figura 5.7: CT Lobster

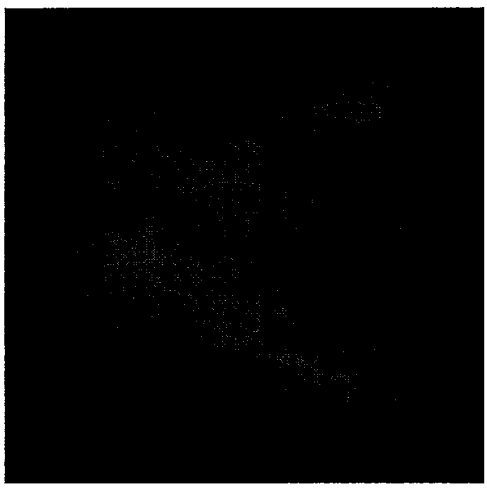


Figura 5.8: CT Skull

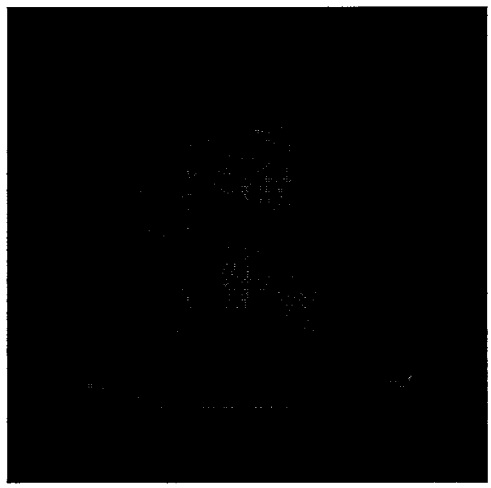


Figura 5.9: CT Bonsai

Capítulo 6

Conclusões

Neste trabalho propomos uma versao paralela e *out-of-core* para o algoritmo de renderização de volume RZSweep. O algoritmo RZSweep é um algoritmo de renderização volumétrica baseado no algoritmo ZSweep. O ZSweep atingiu excelentes resultados para solucionar tarefas de renderização volumétrica para malhas irregulares. Porém, não funciona para malhas regulares. Quem atinge este objetivo é o algoritmo RZSweep.

As grandes massas de dados e o seu crescente aumento, são pivôs do alto custo computacional na execução destes algoritmos. Com o uso eficiente em termos de gasto de memória, o custo computacional pode ser minimizado, ainda que o algoritmo seja sequencial. Porém, o algoritmo sequencial não dá conta de uma grande massa de dados regulares, pelo fato desta ser maior do que a memória disponível no computador.

A implementação *out-of-core* do PRZSweep-OOC permite renderizar estes grandes dados volumétricos, com eficiência e baixo custo. A divisão espacial do volume possibilitou o carregamento sob demanda de partes menores do volume na memória do computador. Esta divisão permite ainda, uma busca eficiente desta porção do volume no disco, devido a sua estrutura hierárquica.

Além disso, a paralelização do algoritmo permitiu o ganho de performance na renderização destes dados, na medida que a divisão espacial foi combinada com

métodos de divisão de tarefas. Foi observado que com a maior subdivisão espacial, e a equilibrada divisão de tarefas no espaço da imagem (*tiles*) e no espaço do objeto (*octree*), ganhos de performance são mais notórios, o que satisfaz o objetivo desta implentação. Ao ser comparado com a implemtação sequencial, o PRZSweep-OOC reduziu o tempo de processamento em média a 75%.

As dificuldades mais evidentes estão relacionadas com a melhor configuração do renderizador. Os valores de *threshold* são particulares a cada dado volumétrico, e alguns deles não vem com indicação da faixa ideal, fazendo com que estes valores tenham que ser pesquisados a partir do método de repetição. A quantidade de faces que serão projetadas são dependentes destes valores. Além disso, a configuração de números de *tiles*, e do nível de divisão espacial está intimamente vinculada à quantidade de processadores disponíveis no *cluster*, inibindo alguns experimentos extras. Em resumo, há diversas melhorias e extensões possíveis. Trabalhos decorrentes desta dissertação podem melhorar bastante a performance e a qualidade final das imagens renderizadas.

Esperamos com este trabalho incrementar a família de algoritmos baseados no ZSweep, e contribuir com a possibilidade de renderização de grandes massas de dados regulares com maior eficiência. Concluimos que a contribuição do PRZSweep-OOC foi significativa. Pois, não só melhorou muito a performance do algoritmo sequencial, mas também, permitiu a renderização de volumes de quaisquer tamanhos.

Referências Bibliográficas

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

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

[3] Y.-J. CHIANG, R. FARIAS, C. S., AND WEI, B. A unified infrastructure for parallel out-of-core isosurface and volume rendering of unstructured grids. In *IEEE Parallel and Large-Data Visualization and Graphics Symposium* (2001).

[4] FARIAS, R., MITCHELL, J. S. B., AND SILVA, C. T. Parallelizing the zsweep algorithm for distributed-shared memory architectures. In *International Workshop on Volume Graphics* (New York, NY, USA, 2001), pp. 91–99.

[5] CHAUDHARY, G., R. L. F. R. Rzsweep: A new hardware-assisted volume-rendering technique for regular datasets. In *International Workshop on Volume Graphics* (2003).

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

[7] LICHTENBELT, B., CRANE, R., AND NAQVI, S. *Introduction to Volume Rendering*, first ed. Hewlett-Packard, 1998.

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

[9] CRAWFIS, R., MAX, N., BECKER, B., AND CABRAL, B. Volume rendering of 3d scalar and vector fields at llnl. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing* (New York, NY, USA, 1993), ACM Press, pp. 570–576.

[10] FOLEY, VAN DAM, FEINER, AND HUGHES. *Computer Graphics Principles and Practice*, second ed. Addison-Wesley, 1996.

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

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

[13] WEILER, M., KRAUS, M., MERZ, M., AND ERTL, T. Hardware-based ray casting for tetrahedral meshes. In *VIS '03: Proceedings of the 14th IEEE conference on Visualization '03* (2003), pp. 333–340.

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

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

[16] DE BERG, M., VAN KREVELD, M., OVERMARS, M., AND SCHWARZKOPF, O. *Computational Geometry: Algorithms and Applications*, second ed. Springer, 2000.

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

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

[19] SILVA, C., MITCHELL, J. S. B., AND KAUFMAN, A. E. Fast rendering of irregular grids. In *VVS '96: Proceedings of the 1996 symposium on Volume visualization* (Piscataway, NJ, USA, 1996), IEEE Press, pp. 15–ff.

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

[21] CHAUDHARY, G. Rzsweep: A new volume-rendering technique for uniform rectlinear datasets. Dissertação de Mestrado, Mississippi State University, May 2003.

[22] DAWSON R. ENGLER, M. F. K., AND JR., J. O. Exokernel: an operating system architecture for application-level resource management.

[23] HAND, S. M. Self-paging in the nemesis operating system. 73–86.

[24] A. ABUTALEB, V. O. K. L. Paging strategy optimization in personal communication systems wireless networks. 195–204.

[25] LINDSTROM, P. Out-of-core simplification of large polygonal models. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2000), ACM Press, pp. 259–262.

[26] LINDSTROM, P. Out-of-core construction and visualization of multiresolution surfaces. In *I3D '03: Proceedings of the 2003 symposium on Interactive 3D graphics* (New York, NY, USA, 2003), ACM, pp. 93–102.

[27] GAO, Y., DENG, B., AND WU, L. Efficient view-dependent out-of-core rendering of large-scale and complex scenes. In *VRCIA '06: Proceedings of the 2006*

ACM international conference on Virtual reality continuum and its applications (New York, NY, USA, 2006), ACM, pp. 297–303.

[28] STAADT, O. G., WALKER, J., NUBER, C., AND HAMANN, B. A survey and performance analysis of software platforms for interactive cluster-based multi-screen rendering. In *EGVE '03: Proceedings of the workshop on Virtual environments 2003* (New York, NY, USA, 2003), ACM, pp. 261–270.

[29] LIAO, H.-S., CHUANG, J.-H., AND LIN, C.-C. Efficient rendering of dynamic clouds. In *VRCAI '04: Proceedings of the 2004 ACM SIGGRAPH international conference on Virtual Reality continuum and its applications in industry* (New York, NY, USA, 2004), ACM, pp. 19–25.

[30] BÖONNING, R., AND MÜULLER, H. Interactive sculpturing and visualization of unbounded voxel volumes. In *SMA '02: Proceedings of the seventh ACM symposium on Solid modeling and applications* (New York, NY, USA, 2002), ACM, pp. 212–219.

[31] P. CIGNONI, C. ROCCHINI, C. M. R. S. External memory management and simplification of huge meshes. In *IEEE Trans. on Visualization and Comp. Graph* (2003), IEEE Press, pp. 525–537.

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

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

[34] WYLIE, B., MORELAND, K., FISK, L. A., AND CROSSNO, P. Tetrahedral projection using vertex shaders. In *VVS '02: Proceedings of the 2002 IEEE Symposium on Volume visualization and graphics* (Piscataway, NJ, USA, 2002), IEEE Press, pp. 7–12.

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

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

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

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