COPPE/UFRJ

ANÁLISE DE DESEMPENHO DO ALGORITMO DE TRAÇADO DE RAIOS GUIADO POR FACE VISÍVEL OTIMIZADO PARA CACHE

Saulo Pereira Ribeiro

Tese de Doutorado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Doutor em Engenharia de Sistemas e Computação.

Orientadores: Antonio Alberto Fernandes de Oliveira
Ricardo Cordeiro de Farias

Rio de Janeiro
Setembro de 2010

ANÁLISE DE DESEMPENHO DO ALGORITMO DE TRAÇADO DE RAIOS GUIADO POR FACE VISÍVEL OTIMIZADO PARA CACHE

Saulo Pereira Ribeiro

TESE SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

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

Prof. Ricardo Cordeiro de Farias, Ph.D.

Prof. Ricardo Guerra Marroquim, D.Sc.

Prof. Cristiana Barbosa Bentes, D.Sc.

Prof. Luiz Marcos Garcia Gonçalves, D.Sc.

Prof. Esteban Walter Gonzalez Clua, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
SETEMBRO DE 2010

Pereira Ribeiro, Saulo

Análise de Desempenho do Algoritmo de Traçado de Raios Guiado por Face Visível Otimizado para Cache/Saulo Pereira Ribeiro. – Rio de Janeiro: UFRJ/COPPE, 2010.

XIV, 73 p.: il.; 29, 7cm.

Orientadores: Antonio Alberto Fernandes de Oliveira
Ricardo Cordeiro de Farias

Tese (doutorado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2010.

Referências Bibliográficas: p. 65 – 73.

1. Visualização Volumétrica. 2. Raycasting. 3. Desempenho. 4. Hierarquias de Memória - Cache. I. Oliveira, Antonio Alberto Fernandes de *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

À minha família.

Agradecimentos

Quero agradecer primeiramente à CAPES, cujo suporte foi indispensável durante os 4 anos de bolsa. Aos meus orientadores, os professores Ricardo Farias e Antonio Oliveira, pela grande oportunidade de compartilhar durante tantos anos não somente o conhecimento, mas também a sabedoria da vida. Aos meus professores do LCG, os professores Claudio Esperança e Paulo Roma, pelo suporte durante 8 anos e meio, desde o mestrado. Na orientação de minha tese não posso deixar de agradecer à professora Cristiana Bentes, ao D.Sc. André Maximo e ao mago da PAPI, Carlos Papaiz, por todas idéias compartilhadas, programas, correções, artigos, etc ao longo desses anos. Quero agradecer aos membros externos da banca, os professores Luiz Marcos e Esteban Clua, pela disponibilidade e participação.

Quero agradecer também aos amigos que começaram comigo o mestrado, continuaram no doutorado e já são doutores: Alvaro Cuno, Ricardo Marroquim, Disney Oliveira e Cesar Xavier. Aos meus amigos Yalmar Ponce, Guina Sotomayor, Mariela Morveli, Liliana Sanchez e Karl Apaza por compartilharmos muitos anos de nossas vidas juntos.

Não posso deixar de agradecer aos super profissionais da secretaria do PESC que resolvem os problemas que nós alunos criamos: Claudia, Solange, Sonia, Mercedes, Natália. O pessoal do suporte Adilson, Itamar e sua equipe.

Aos amigos do LCG, Felipe, Leandro, Vitor, Carlos Eduardo, Wagner, Okamoto, Alopes, Margareth, Bruno, Renan, Leandro Gazoni, Luciano de Paula, Celina, Alberto, Guilherme Cox, Leticia, Rafael, Rubens, Tiago, Aruquia, Caique, Caniato, Diego, Daniel, Flavio, Jonas, Luis, Retondaro e Zezim.

Aos amigos que me apoiaram nesta jornada: João Marcos, Flavia, Fabiane, Kris, Matheus, João Victor, Marlene, Tadeu. Às minhas amigas, Alessandra Wasilewski, Neide Angelica, Cinthia Hofacker e Nilza dos Santos por tudo.

Aos meus amigos Walter Tristão, Lourdes Tristão, Domingas Loss e companheiros de grupo por uma vida inteira compartilhada.

Ao meu super amigo e grande incentivador Ernane Ribeiro.

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

ANÁLISE DE DESEMPENHO DO ALGORITMO DE TRAÇADO DE RAIOS GUIADO POR FACE VISÍVEL OTIMIZADO PARA CACHE

Saulo Pereira Ribeiro

Setembro/2010

Orientadores: Antonio Alberto Fernandes de Oliveira
Ricardo Cordeiro de Farias

Programa: Engenharia de Sistemas e Computação

Nesta tese, propomos melhorias sobre os algoritmos tradicionais de ray-casting para CPU, visando obter um algoritmo mais eficiente, tanto em desempenho quanto em consumo de memória. Desenvolvemos um algoritmo de ray-casting guiado por face visível – VF-Ray – que otimiza o uso de memória ao explorar a coerência dos raios. Assim, mantemos na memória principal, a informação das faces percorridas pelo raio que é lançado por cada pixel sob a projeção de uma face visível. Nossos resultados mostram que ao explorarmos esta coerência, reduzimos consideravelmente o uso de memória, além de mantermos o desempenho de nosso algoritmo competitivo com os mais rápidos enfoques anteriores. Fizemos uma análise dos efeitos da hierarquia de memória para dados irregulares, com o objetivo de avaliarmos o comportamento do VF-Ray em relação ao uso da cache.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

PERFORMANCE ANALYSIS OF RAYCASTING ALGORITHM GUIDED BY VISIBLE FACE CACHE OPTIMIZED

Saulo Pereira Ribeiro

September/2010

Advisors: Antonio Alberto Fernandes de Oliveira
Ricardo Cordeiro de Farias

Department: Systems Engineering and Computer Science

In this work, we propose improvements over traditional ray-casting algorithms for CPU, aiming to get a more efficient algorithm in both performance and memory usage. We developed a ray-casting algorithm guided by visble face – VF-Ray – which optimizes the memory usage by exploring ray coherence. So, we keep in main memory the information of the faces traversed by the ray cast through every pixel under de projection of a visible face. Our results show that exploring this coherence we reduce considerably the memory usage, while keeping the performance of our algorithm competitive with the fastest previous ones. We did an analysis of the memory hierarchy effects for irregular datasets in order to evaluate VF-Ray behavior regarding the use of cache.

Sumário

Lista de Figuras **x**

Lista de Tabelas **xii**

Índice Remissivo **xiii**

1 Introdução **1**

1.1 Contribuições 5

1.2 Organização 6

2 Conceitos Básicos **7**

2.1 Visualização Volumétrica 10

2.2 Trabalhos Importantes de Visualização Volumétrica 12

2.2.1 Integral de Visualização do Volume 13

2.2.2 Ray-casting 15

2.2.3 Splatting 16

2.2.4 Shear-warp 17

2.2.5 Métodos Baseados em Mudança de Base (Domínio) 17

2.3 Organização da Memória Cache 18

2.3.1 O porquê da cache 18

2.3.2 Cache em Detalhes 20

3 Trabalhos Relacionados **23**

3.1 Trabalhos de Cache na Visualização Volumétrica 25

4 Algoritmo de Ray-Casting **28**

4.1 Enfoques Anteriores 30

4.1.1 Bunyk 30

4.1.2 ME-Ray 31

4.1.3 EME-Ray 31

5 Algoritmo de Ray-Cast Orientado por Face Visível **33**
5.1 Explorando a Coerência de Raio 33
5.2 Estruturas de Dados 34
5.3 Tratamento de Casos Degenerados 35
5.4 O Algoritmo 36

6 EVF-Ray – O VF-Ray Otimizado **38**

7 Avaliação de Desempenho e Consumo de Memória **40**
7.1 Ambiente de Testes 40
7.2 Consumo de Memória 41
7.3 Tempo de Execução 43
7.4 Discussão 44
7.5 Resultados de Desempenho do EVF-Ray 45

8 Análise de Cache para o Algoritmo de Ray-Casting **49**
8.1 Ferramentas de Avaliação de Desempenho 50
8.2 Metodologia para Avaliação das Hierarquias de Memória 51
8.3 Ambiente de Testes 53
8.4 Resultados da Influência das Hierarquias de Memória 54

9 Considerações Finais **63**
9.1 Direções Futuras 64

Referências Bibliográficas **65**

Lista de Figuras

1.1 Crânio Humano 2
1.2 Ray-casting 2
1.3 Amostra Biológica do Solo 3
1.4 Muschelkalk 5

2.1 Visão geral do processo de visualização volumétrica 7
2.2 Crânio Humano 8
2.3 Simulação de Fluido 8
2.4 Campo escalar 9
2.5 Tipos de Malhas 9
2.6 Malha curvilínea 10
2.7 Malha não estruturada 10
2.8 Visualização de superfície e volume 11
2.9 Simplificação da integral 14
2.10 Combinado células para gerar cor 15
2.11 shear-warp 17
2.12 Algritmo Shear-Warp 18
2.13 Hierarquia de Memória 19
2.14 Cache e MP 21
2.15 Cache L3 21

4.1 Composição da cor no voxel. 29
4.2 Esquema simplificado do algoritmo de ray-casting. 29

5.1 Coerência de raios da face visível. 34
5.2 Casos de interseção do raio com uma célula. 35
5.3 Pseudocódigo do *VF-Ray*. 37

6.1 Nova Estrutura de Dados do VF-Ray em CPU 39

7.1 Consumo de memória para o dado SPX. 42
7.2 Imagens dos volumes testados pelo VF-Ray 46

8.1 D1mr volume completo Alg. *EVF-Ray* 55
8.2 D2mr volume completo Alg. *EVF-Ray* 58
8.3 D2mr VF 512 a 4096 59
8.4 D1mr VF 512 a 4096 59
8.5 D2mr VF 512 32 faces 60
8.6 Tempo Renderização VF 512 a 4096 61
8.7 D1mr VF 512 32 faces 62

Lista de Tabelas

7.1 Dimensões dos dados de entrada. 41
7.2 Consumo de memória do VF-Ray versus ME-Ray, EME-Ray, Bunyk e ZSweep. 41
7.3 Resultados de tempo para o VF-Ray versus ME-Ray, EME-Ray, Bunyk e ZSweep. 43
7.4 Consumo de memória do *EVF-Ray* versus *VF-Ray*, *ME-Ray*, *EME-Ray* e *Bunyk*. 47
7.5 Resultados de tempo para o *EVF-Ray* versus *VF-Ray*, *ME-Ray*, *EME-Ray* e *Bunyk*. 48

8.1 Eventos de Cache gravados utilizando as ferramentas Cachegrind e Callgrind. 54
8.2 D1mr volume completo Alg. *EVF-Ray* 56
8.3 D2mr volume completo Alg. *EVF-Ray* 57

Índice Remissivo

[2D] duas dimensões, 7
[3D] três dimensões, 7
[CUDA] Compute Unified Device Architecture, 38
[DRAM] Dynamic Random Access Memory, 18
[GPU] Unidade de Processamento Gráfico, 3
[L1d] Level 1 data, 21
[L1i] Level 1 instruction, 21
[L2] Level 2, 22
[L3] Level 3, 22
[MIMD] Multiple Instruction Multiple Data, 25
[PAPI] The Performance API, 50
[RMI] Ressonância Magnética por Imagem, 7
[SRAM] Static Random Access Memory, 18
[TC] Tomografia Computadorizada, 7

Capítulo 1

Introdução

Com o passar dos anos, a visualização volumétrica vem sendo cada vez mais utilizada, devido à necessidade de se explorar características específicas do volume. E, devido à sua importância, foram surgindo diversas aplicações, nas mais variadas áreas do conhecimento, com o objetivo de fornecer meios que permitam os pesquisadores não somente visualizar dados de diversas natureza, mas também poderem analisar aspectos intrínsicos e assim chegarem às conclusões de que necessitam. Esta evolução utilizou-se de diversas tecnolgias como computação paralela, placas gráficas e técnicas bem elaboradas em um único processador.

Em muitos campos de estudo são gerados grandes quantidades de dados volumétricos que representam simulações ou objetos tridimensionais. Portanto é preciso de métodos que permitam um estudo sobre esses resultados. Segundo McCormick, Visualização [1] é um conjunto de técnicas utilizadas para interpretação dos dados modelados no computador e para a geração de imagens a partir destes dados multidimensionais.

Uma de suas características fundamentais é o tratamento de grandes quantidades de dados de natureza volumétrica, com o objetivo de apresentar suas interações, interligações e características que são consideradas complexas para serem percebidas em sua forma original. Algoritmos de visualização volumétrica, tratam os dados como se fossem compostos por um material semitransparente, permitindo mostrar detalhes do seu interior (Figura 1.1). Entre as grandes áreas de aplicação da visualização volumétrica encontram-se a Medicina, Meteorologia, Química, Física, Engenharia, Sistemas de Informações Geográficas, Arqueologia, Biologia (Figura 1.3) e Geologia (Figura 1.4) entre outras.

Na visualização volumétrica, quando uma imagem é gerada esta contém elementos de figura ou *pixels* [3], enquanto que os dados volumétricos contêm elementos de volume ou *voxels (volume elements)* [4].

Existem diversos algoritmos de renderização volumétrica propostos na literatura [5–9], entre eles destacam-se os seguintes paradigmas: projeção de células e

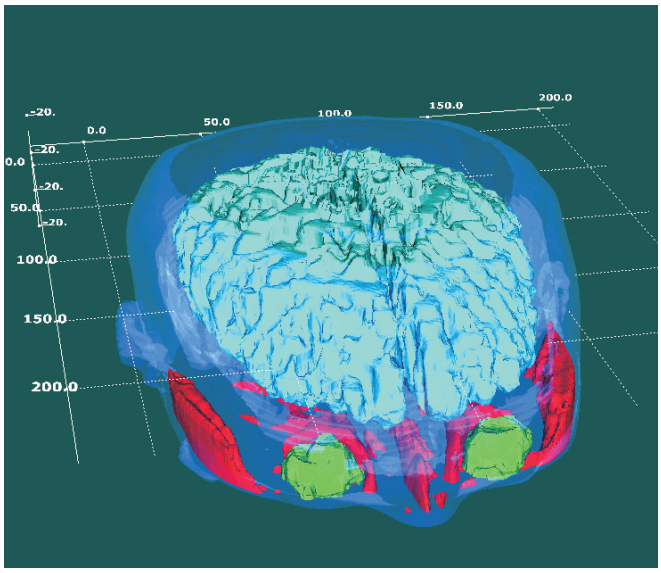

Figura 1.1: MRI 3D de um crânio humano [2].

ray-casting. Algoritmos baseados em projeção de células percorrem as células de um dado volumétrico em uma ordem pré-determinada, projetando suas faces sobre os pixels da imagem. Algoritmos baseados em ray-casting calculam a cor de cada pixel através do lançamento de raios que atravessam as células do volume a ser renderizado.

No ray-casting, um raio é lançado através de cada pixel da imagem a partir do ponto de visão (*viewpoint*), Figura 1.2. O traçado do raio determina as faces das células do volume que cada raio intersecta. Cada par de interseções é usado para computar a contribuição de cada célula para a cor e a opacidade do pixel. O raio para ao sair do volume, ou se atingir a opacidade máxima (geralmente igual a um).

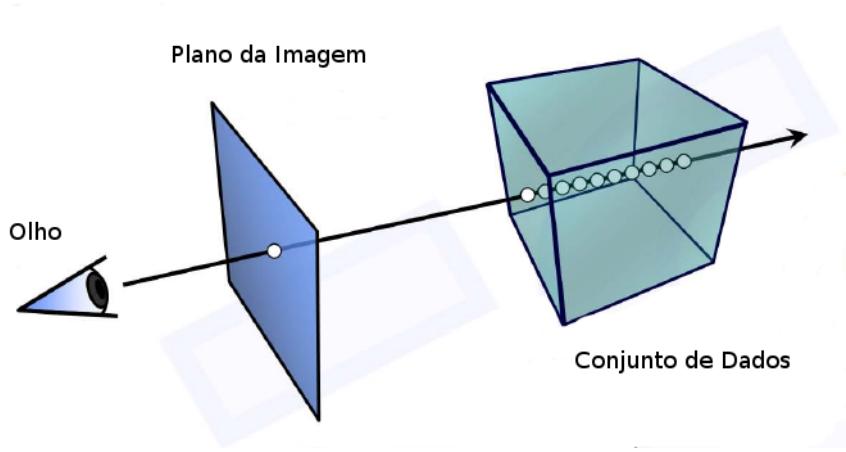

Figura 1.2: Modelo de ray-cast [2].

Comparado com outros métodos de renderização direta de volume, as grandes vantagens dos métodos de ray-casting são:

- a computação para cada pixel é independente de todos os outros pixels;
- e a passagem de um raio através da malha é guiada pelas conectividades das células da malha, evitando assim, a necessidade de se ordenar as células.

A desvantagem é que a conectividade das células tem de ser computada explicitamente e mantida em memória. Em outras palavras, a quantidade de memória usada por algoritmos de ray-casting é um grande obstáculo quando se deseja manipular modelos muito grandes.

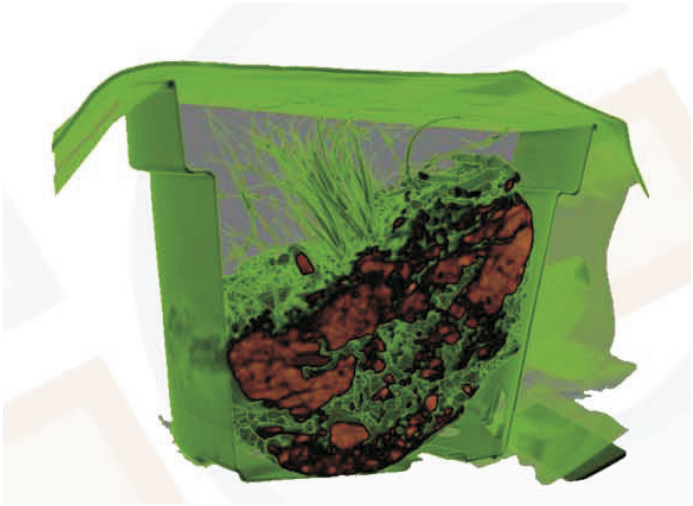

Figura 1.3: TC de uma amostra biológica do solo. Grupo de Realidade Virtual, Universidade de Erlangen, Alemanha [10].

Independente do algoritmo de renderização utilizado, a visualização volumétrica de grandes massas de dados é um problema conhecidamente custoso em termos computacionais. A crescente demanda por desempenho tem sido o foco de uma série de pesquisas. Atualmente, boa parte das soluções com tempos interativos de renderização estão sendo obtidas com ajuda da GPU e de Sistemas Paralelos. Entretanto, à medida que os dados crescem, a capacidade da memória principal de cada computador ou da placa gráfica, acaba se tornando o grande gargalo de desempenho, tanto para sistemas paralelos, como para implementações em hardware gráfico.

Consequentemente, o problema de renderizar grandes conjuntos de dados (que não cabem na memória da GPU) deve ser tratado através de soluções de software [11–14]. Na literatura, entretanto, poucas soluções de ray-casting em software foram propostas para tratar malhas irregulares. Garrity [15] propôs o primeiro método para Ray-Casting de malhas irregulares usando a conectividade das células. Bunyk *et al.* [16] posteriormente aperfeiçoam o trabalho de Garrity provendo um algoritmo mais rápido. Pina *et al.* [17] melhoraram o enfoque de Bunyk em:

- consumo de memória;
- tratamento completo de casos degenerados;
- tratamento de malhas tetraedrais e hexaedrais.

Através do uso de novas estruturas de dados, Pina *et al.* [17] reduzem significativamente os requisitos de memória do enfoque de Bunyk, mas os algoritmos propostos ficam bem mais lentos comparados ao Bunyk *et al.* [16].

Neste trabalho, avaliamos o desempenho e uso da memória de sistemas de renderização volumétrica baseados no algoritmo de ray-casting. Mais especificamente, estamos interessados em avaliar detalhadamente a relação entre custo de computação e gasto de memória do algoritmo de ray-casting implementado. A partir desta avaliação, pretendemos propor novas implementações para o algoritmo ray-casting que permitam não só uma execução eficiente com baixo uso de memória, mas também que seja *leve*, portável e de fácil utilização.

Para resolver o problema de gasto de memória, nossa idéia é utilizar estruturas de dados mais compactas, focando no espaço gasto para armazenar os dados das faces das células. Para resolver o problema de custo computacional, nossa idéia é atacar duas frentes diferentes: melhorar o uso da cache, explorando a coerência entre raios, e paralelizar o algoritmo explorando os ganhos de uma arquitetura paralela, como por exemplo uma unidade de placa gráfica (GPU).

Como uma primeira proposta, implementamos um novo algoritmo de ray-casting baseado nos algoritmos propostos por Pina *et al.* [17], que explora a coerência dos raios, mantendo na memória somente a informação das faces dos raios percorridos mais recentes. Nossa idéia é usar cada face visível computada na etapa de pré-processamento, para guiar a criação e a destruição dos dados das faces internas na memória. Nosso algoritmo, chamado Visible Faces Ray-Cast — *VF-Ray* [18], obtém ganhos consistentes e significativos no consumo de memória comparado com os enfoques anteriores.

A partir da implementação de *VF-Ray*, existe uma série de questões que precisam ser investigadas em detalhe no desenvolvimento de um algoritmo de renderização eficiente e de baixo custo. Será preciso analisar o uso de cache do *VF-Ray*. Raios de uma mesma face visível tendem a apresentar localidade temporal, entretanto, a escolha da próxima face visível a ser computada pode ter grande influência na localidade do novo grupo de raios.

Para dados irregulares, a literatura não apresenta trabalhos sobre o estudo de cache (CPU) de Ray-casting em um único processador. Já para dados regulares, poucos trabalhos apresentam este estudo. A maioria se concentra na melhoria do desempenho. E, de fato isto é importante, pois impacta no tempo desejado para

se obter um resultado, principalmente, quando o modelo utilizado é muito grande. Palmer *et al.* [19] apresentam um estudo de como o uso do cache (CPU) em arquitetura de memória compartilhada impacta no desempenho de algoritmos de ray-casting para dados regulares, mostrando que o bom uso das hierarquias de memória aumentam o desempenho, não somente em um único processador, mas em sistemas paralelos.

O estudo de cache para dados irregulares é importante, pois estes não apresentam um padrão, como o existente em dados regulares. Os dados irregulares podem conter buracos, ter inúmeras variações de forma, comprimento entre outros, o que torna complexa a renderização de suas estruturas internas. A forma como os dados serão processados impactará no desempenho do algoritmo. Portanto é preciso que este processamento seja feito de tal forma que, para a maioria dos modelos irregulares, se obtenha um ótimo desempenho. Desta forma, tem-se um método que não depende das características do modelo. Já o ray-casting sobre dados regulares foca no melhor uso dos blocos a serem processados a fim de se obter um bom desempenho.

Com relação à paralelização do *VF-Ray*, esta é feita de forma direta, já que o elemento principal são as faces visíveis. Existem duas extensões do *VF-Ray* para arquiteturas diferentes, GPU e CELL (processador Cell Broadband Engine [20]), mas não fazem parte do escopo desta tese. Um maior detalhamento poderá ser visto em [21, 22].

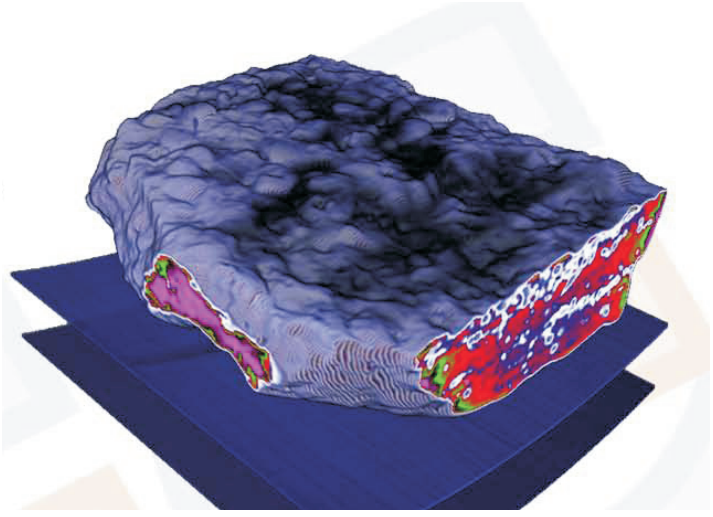

Figura 1.4: Muschelkalk (um tipo de concha): sedimentos a base de calcário do período triássico alemão. Paleontologia. Grupo de Realidade Virtual, Universidade de Erlangen, Alemanha. [10].

1.1 Contribuições

Em resumo, as contribuições desta tese são as seguintes:

1. Avaliação detalhada de diferentes implementações do algoritmo de ray-casting no que se refere a:

   - uso da cache - a localidade temporal na computação dos raios tem grande influência na quantidade de *cache miss* e *cache hit* do algoritmo;
   - relação uso de memória e tempo de execução - quanto maior for a quantidade de informação armazenada na memória, mais rápido o algoritmo computará a próxima interseção do raio;
   - gasto de memória de cada estrutura de dados utilizada.

2. Desenvolvimento e avaliação de uma nova implementação do algoritmo de ray-casting, *VF-Ray*, baseada na coerência de raios, utilizando a computação das faces visíveis para guiar a criação e a destruição de dados na memória;

3. Otimização do *VF-Ray* através da reestrutaçao de sua estrutura de dados e da determinação mais eficiente da face da saída, ou seja, a próxima face a ser cortada pelo raio. Esta versão otimizada é chamada de *EVF-Ray*;

4. Análise dos efeitos da hierarquia de memória para dados irregulares, com o objetivo de avaliarmos o comportamento do *VF-Ray* em relação ao uso da cache.

Dessa forma, estaremos contribuindo significativamente para o desenvolvimento de uma ferramenta de visualização volumétrica eficiente, escalável e com baixo custo.

1.2 Organização

O restante desta tese está organizada como segue. O capítulo 2 descreve os conceitos básicos na área de visualização científica, explicando sobre os dados volumétricos e sobre os algoritmos de visualização e também os conceitos de memória cache, na área de arquitetura de computadores. O capítulo 3 apresenta os trabalhados relacionados com a nossa proposta. O capítulo 4 descreve o algoritmo de ray-casting. No Capítulo 5 apresentamos o algoritmo *VF-Ray* que reduz o consumo de memória, explorando a coerência entre raios. Já no Capítulo 6 temos a versão otimizada do *VF-Ray*, o *EVF-Ray*. O capítulo 7 apresenta os resultados do *VF-Ray* e do *EVF-Ray*, mostrando o potencial na redução do consumo de memória e tempo de execução, quando comparado a propostas anteriores. No capítulo 8 estudamos os efeitos da hierarquia de memória para o algoritmo de ray-casting para dados irregulares. Finalmente, no Capítulo 9 apresentamos as considerações finais e trabalhos futuros.

Capítulo 2

Conceitos Básicos

Como já vimos na introdução, o objetivo da visualização volumétrica é a extração de informações relevantes dos dados que representam o volume, veja a Figura 2.1. Para tal estes dados devem conter informações espaciais referentes ao seu interior. Quando essas informações são obtidas através da discretização do espaço, então temos dados volumétricos. Estes estão em 3D e as informações podem ser arestas ou superfícies.

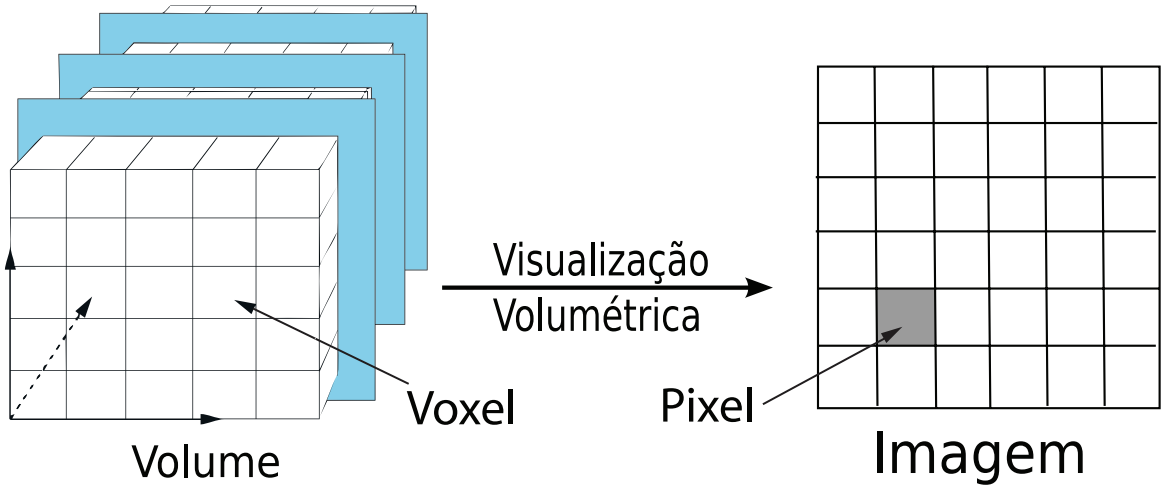

Figura 2.1: O processo de visualização volumétrica é utilizado na geração de imagens a partir de informações relevantes contidas no volume.

Os dados volumétricos têm como principal origem, os dados amostrados de fenômenos naturais ou objetos reais. Também podem ter origem em técnicas de modelagem, de resultados numéricos que são gerados por experimentos empíricos ou simulações. Como exemplo de dados amostrados podemos ter uma sequência de fatias em 2D obtidas de uma tomografia computadorizada (TC) , a Figura 2.2, ou uma ressonância magnética por imagem (RMI) que é reconstruída em três dimensões num volume e visualizada a fim de se obter um diagnóstico, planejar um tratamento etc.

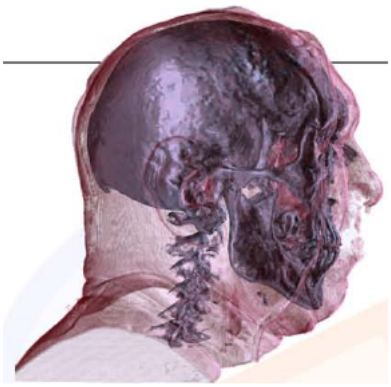

Figura 2.2: TC Crânio Humano. Projeto Homem Visível. Biblioteca Nacional de Medicina dos EUA. Maryland, EUA. [10].

Figura 2.3: Simulações das chamas de explosões [23].

Em muitas áreas da computação, como por exemplo a área de dinâmica dos fluídos, a Figura 2.3, os resultados das simulações geralmente são visualizados como dados volumétricos para fins de verificações e análises.

Como os dados volumétricos consistem de informações de posições no espaço, estas informações podem representar campos escalares, vetoriais ou uma combinação de ambos. Assim, x, y, z, w representa um elemento do conjunto de dados nos quais x, y, z caracteriza a posição e w o valor associado a mesma. Por um campo escalar, queremos dizer uma quantidade que depende da posição no espaço ou simplesmente um campo que é caracterizado em cada ponto por um simples número–um escalar. Como exemplo podemos ter temperatura, densidade, etc. Na Figura 2.4 temos o exemplo da temperatura, na qual em cada ponto x, y, z do espaço está associado um número $T(x, y, z) = w$. Assim todos os pontos sobre a superfície com $T = 40^o$ (mostrada como uma curva em $z = 0$) possuem a mesma temperatura. As setas são exemplos de vetores de fluxo de calor $\mathbf{h}$.

Como exemplo de campos vetoriais, temos a velocidade, fluxos (de calor) etc. Considere o fluxo de calor num bloco. Se a temperatura no bloco é alta em um local e baixa em outro, então haverá fluxo de calor dos locais mais quentes para os mais frios. O calor fluirá em diferentes direções das diferentes partes do bloco. O fluxo

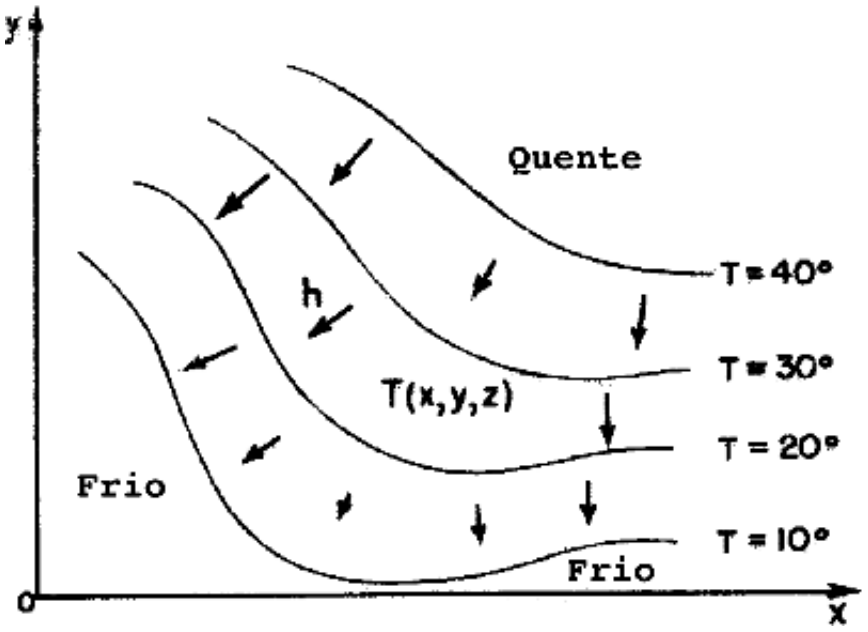

Figura 2.4: Temperatura é um campo escalar [24].

de calor é uma grandeza vetorial que denominada **h**. Sua magnitude é uma medida de quanto calor está fluindo.

De acordo com o tipo de dado volumétrico teremos a utilização de diferentes técnicas de visualização. Frequentemente, os dados volumétricos são representados por grades retilíneas em $3D$, nas quais cada elemento de volume que a compõe é chamado de *voxel* (volume element) [4]. Em contrapartida, a imagem gerada contém elementos de figura ou *pixels* [3], como representado na Figura 2.1.

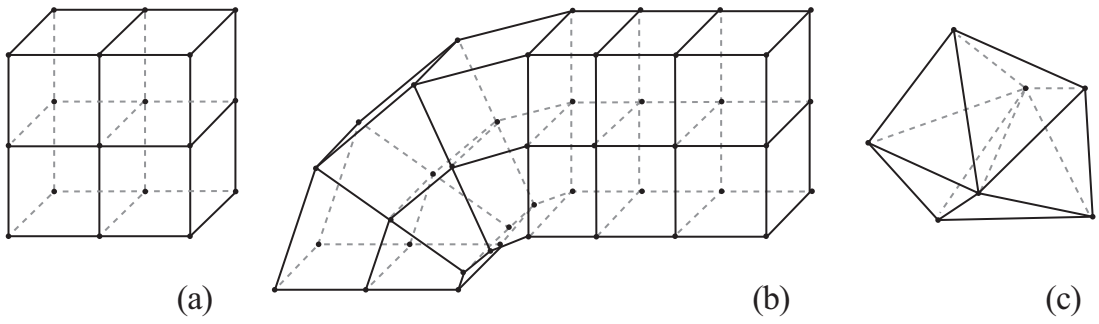

Figura 2.5: Tipos de representação de dados volumétricos: (a) Grade retilínea regular; (b) Grade Curvilínea; (c) Grade irregular [25].

A cada voxel está associada alguma característica do objeto ou fenômeno em questão. Quando o conjunto de dados é formado por todos os voxels como cubos idênticos, então este é classificado como regular, a Figura 2.5(a). Quando a grade retilínea sofre uma transformação não-linear, mas preserva a sua topologia, esta passa a ser uma grade curvilínea ou grade estrutrurada, as Figuras 2.6 e 2.5(b).

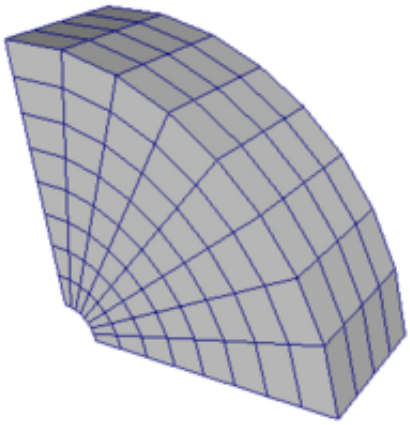

Figura 2.6: Malha estruturada curvilínea [26].

O arqueamento sofrido pela grade tem o objetivo de criar uma amostragem mais densa dos pontos de uma região peculiar do espaço. Quando os dados volumétricos são formados por células poliedrais (tetraedros, hexaedros) e necessitam que a conectividade destas seja fornecida, estes são ditos irregulares ou não-estruturados, as Figuras 2.5(c) e 2.7.

Os dados regulares são muito utilizados na modelagem de imagens médicas, já as grades curvilíneas são utilizadas em simulações de fluído. Os dados irregulares podem ser usados, por exemplo, para simular a combustão no interior de um reator.

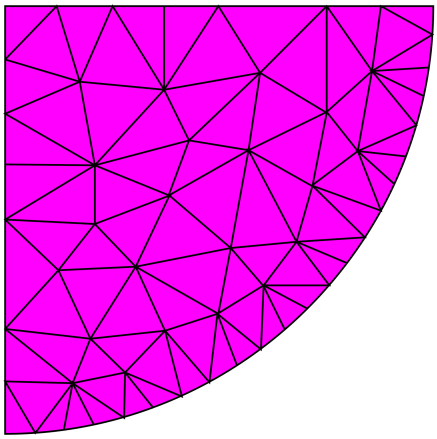

Figura 2.7: Malha não estruturada (irregular) [27].

2.1 Visualização Volumétrica

Com o passar do tempo, surgiram diferentes modalidades de aquisição de dados volumétricos e, assim, vários métodos de visualização foram desenvolvidos. Uma vez que os métodos para exibição de primitivas geométricas já estão bem estabele-

cidos, a maioria dos métodos primitivos implica em aproximar uma superfície contida nos dados usando primitivas geométricas. Estes algoritmos utilizam primitivas geométricas, que estão implementadas em hardware, como polígonos, a fim de exibir os dados, viabilizando assim a visualização. Em geral, esses métodos precisam saber, para cada amostra de dados, se a superfície passa ou não por ele. Isto gera falsos positivos (superfícies errôneas) ou falsos negativos (buracos que não pertencem à superfície). E, como a informação sobre o interior dos objetos geralmente não é guardada, esses métodos perdem a informação referente a uma das dimensões. A fim de resolver esse problema da visualização de superfície, técnicas de renderização de volume direta(*direct volume rendering*) [28, 29] foram desenvolvidas na tentativa de capturar integralmente o dado em $3D$ em uma imagem $2D$. Essas técnicas carregam mais informações do que as de visualização de superfície na imagem gerada, conforme podemos observar na Figura 2.8, mas a um determinado custo. Este aumenta a complexidade do algoritmo e, por conseguinte, gera um aumento do tempo de visualização. A fim de diminuir este custo computacional para se poder obter uma maior interatividade na visualização volumétrica, muitos algoritmos são desenvolvidos, bem como, hardwares específicos para este propósito.

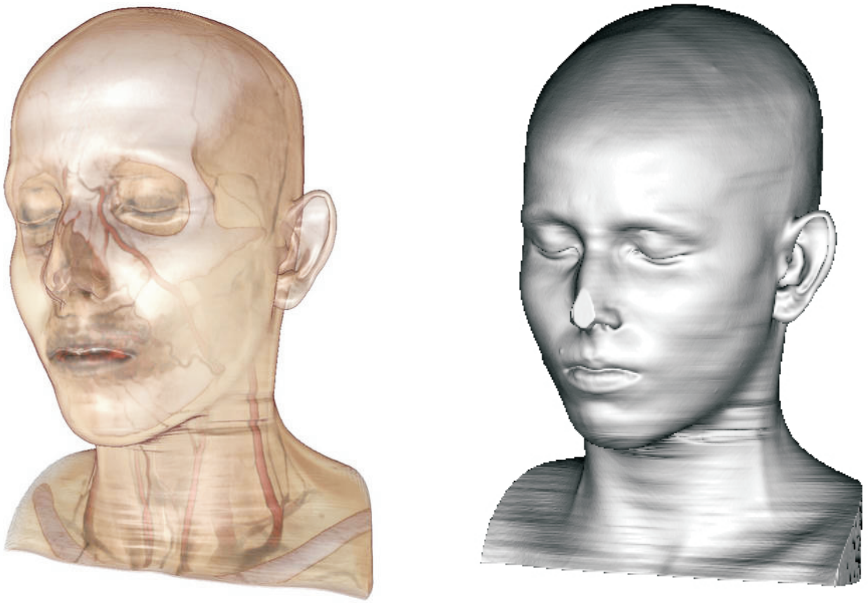

Figura 2.8: Diferença entre a visualização volumétrica (à esquerda) e superficial (à direita) do mesmo objeto volumétrico.

Algoritmos de visualização volumétrica incluem enfoques como ray-casting [30], splatting [31] e shear-warp [32]. Ao invés de extrair uma representação intermediária, a visualização volumétrica provê um método para exibir diretamente os dados volumétricos. As amostras originais são projetadas no plano da imagem através de um processo que interpreta os dados com uma nuvem de partículas amorfas.

Segundo Kaufman [33], a visualização volumétrica pode ser feita de três formas diferentes, mas pode-se acrescentar uma quarta que é um hibridismo de duas outras.

1. Espaço do objeto;
2. Espaço da imagem;
3. Baseado em domínio;
4. Híbrido (Espaço do objeto + Espaço da imagem).

As técnicas com enfoque no espaço do objeto fazem o mapeamento das amostras dos dados $3D$, ou seja, percorrem as células do volume e computam suas contribuições para a geração da imagem. Como exemplo, temos as técnicas:

- Rasterização que projeta primitivas planas (triângulos) no plano da imagem [34];
- Projeção de Célula que projeta primitivas volumétricas no plano da imagem [35];
- Projeção de Vértice ou Splatting [31] que projeta primitivas de vértices no plano da imagem.

Já as técnicas baseadas no espaço da imagem, partem dos pixels da imagem do plano e computam a contribuição das células apropriadas para estes pixels. Como exemplo, temos o ray-casting [30]. No ray-casting a cor do pixel é avaliada utilizando-se um modelo de iluminação local, enquanto que o raytracing suporta modelos de iluminação global. Nas técnicas baseadas em domínio, os dados volumétricos $3D$ sofrem a transformação para um outro domínio que poder ser, por exemplo, o domínio da frequência. Os métodos híbridos procuram combinar as vantagens dos métodos no espaço do objeto e da imagem. Como exemplo, temos o algoritmo shear-warp [32]. Este é restrito às grades cartesianas da quais as fatias da grade que sofrem cisalhamento (shear) são projetadas num eixo alinhado com o plano e posteriormente sofrem uma transformação de dobra (warp) no plano da imagem.

2.2 Trabalhos Importantes de Visualização Volumétrica

Nesta seção, são apresentados importantes trabalhos na área de visualização volumétrica. Iremos destacar os trabalhos abaixo relacionados:

- Integral de Visualização do Volume;

- Ray-casting;
- Splatting;
- Shear-warp.

2.2.1 Integral de Visualização do Volume

Antes mesmo que possamos visualizar um volume transparente, se faz necessário o entendimento de como a luz interage com o volume, ou seja, de como o volume faz o transporte da luz. Os modelos ópticos descrevem esse transporte da luz no interior do volume. A integral de visualização de volume ou integral de iluminação é uma equação que computa a cor da luz que passa através do volume.

Nas técnicas de visualização de volume direta ou visualização de volume, os modelos óticos vêem o volume como uma nuvem de partículas [36, 37]. A luz emitida por uma fonte pode ser espalhada ou absorvida por partículas. Modelos que levam em conta todo o fenômeno de interação da luz são muito complicados. Por esse motivo, modelos práticos utilizam várias simplificações. Uma aproximação comum para a integral de visualização de volume é dada por [38]:

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

Na Equação 2.1, $I(D)$ é a quantidade de luz proveniente de uma direção de raio que parte do fim do volume ($s = 0$) até o ponto de visão ($s = D$). O primeiro termo desta equação computa a quantidade de luz (I_0), que chega até o final do volume. Essa luz de entrada (intensidade inicial) é atenuada exponencialmente com o comprimento D, ou seja, esta intensidade é absorvida na medida em que o raio atravessa o volume. Como analogia, o mesmo acontece quando observamos objetos situados a várias distâncias num dia enevoado. O segundo termo desta equação adiciona à quantidade de luz, que é emitida por cada ponto ao longo do raio, considerando quanto foi atenuado em cada ponto até o final deste.

Em Max [38], podemos ver diferentes modelos para interação da luz com o volume. A equação 2.1 leva em consideração os efeitos de absorção e emissão, mas descarta efeitos mais avançados como espalhamento (a iluminação de uma partícula por raios de luz refletidos sobre outras partículas) e sombreamento (atenuação da luz entre uma partícula e a fonte de luz) [38]. Essas aproximações tornam o cálculo mais simples, considerando somente a luz que passa diretamente entre uma partícula e o ponto de visão.

Como vimos no início, a modelagem integral deste fenômeno tem um alto custo computacional e como consequência, o desempenho diminui. Visando obter uma

melhor relação custo-benefício, os trabalhos [39–41] propõem simplificações na integral de visualização de volume. A simplificação consiste na integração da função de transferência [28] durante o pré-processamento, antes que seja feita a visualização em si. Desta forma, esta integração gera valores de cor associada e opacidade, que são guardados em uma tabela $3D$. Considerando que um raio de luz tem origem no ponto de visão e atravessa uma célula, a Figura 2.9, tendo uma função de transferência com domínio escalar, a integral de iluminação dependerá somente da distância percorrida no interior da célula l, e dos valores escalares da posição de entrada s_f (front scalar) e de sáida s_b (back scalar) do raio. Os escalares s_f e s_b são interpolados levando em conta os escalares dos vértices da célula.

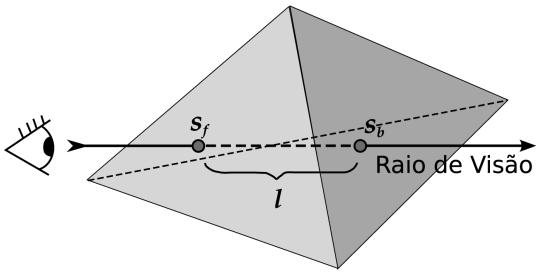

Figura 2.9: Simplificação da Integral de Visualização Volumétrica. O raio tem origem no observador e atravessa a célula tetraedral.

Quando a célula é um tetraedro, o campo escalar tem uma variação linear com relação à distância l que foi percorrida pelo raio, entre s_f e s_b. Assim a opacidade do tetraedro e a cor associada podem ser reescritos em função das três variáveis (l, s_f, s_b) [39]:

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

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

Onde a cor C é a média das cores de entrada $C(s_f)$ e saída $C(s_b)$. A opacidade α tem como base o coeficiente de extinção τ, que representa a quantidade de luz que é absorvida pela célula C_i. O coeficiente de extinção também depende dos valores escalares na posição de entrada e saída da célula, $\tau(s_f)$ e $\tau(s_b)$, respectivamente. Este mapeamento é feito através da função de transferência. Esta deve ser integrada célula a célula, assim a combinação dos resultados resultará na cor final do pixel. A

combinação linear das cores anteriores C_i e C_{i-1}, que resulta no valor atualizado da cor C_{i+1}, é dada por:

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

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

Onde α_i é a opacidade computada ao atravessar à celula atual, α_{i-1} a opacidade computada até a célula anterior e α_{i+1} a opacidade final da composição. Podemos ver na Figura 2.10 como o valor final da cor de um pixel foi gerado. A combinação dos fragmentos de cor $RGBA$ até a célula anterior $i-1$ (que é dado por (C_{i-1}, α_{i-1})) com a célula corrente i, resulta na cor do pixel.

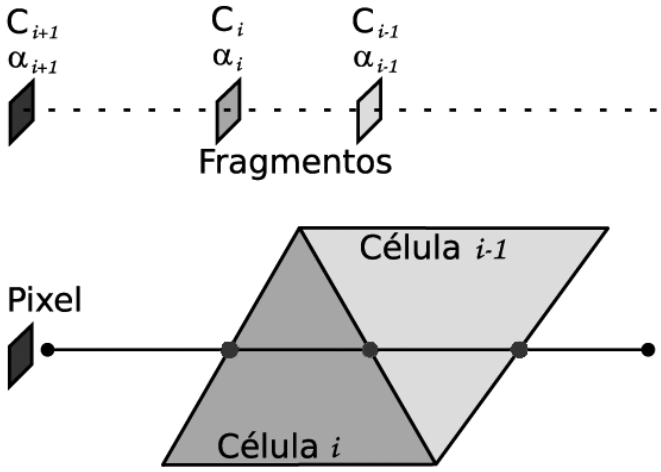

Figura 2.10: A cor do pixel é obtida através da combinação das células.

Existem outras formas de simplificação como feita em [40]. Resumindo, vimos que a computação exata tem um alto custo e que é preciso utilizar outros métodos que fazem uso de simplificações a fim de se obter um melhor desempenho.

2.2.2 Ray-casting

No capítulo 5 veremos o ray-casting mais detalhadamente, sendo assim faremos apenas uma rápida passagem por este conceito.

Esta técnica de visualização volumétrica, foi inspirada na técnica de renderização de superfícies, **traçado de raios** (*ray tracing*), que é antiga em computação gráfica e seus conceitos foram introduzidos por Jim Blinn [37]. A técnica de traçado de raios, também conhecida como *ray-shooting*, é atualmente definida como *ray-casting*. Segundo Blinn [37], um raio de luz é lançado por cada pixel da tela, quando este

incide em um objeto, computa-se a quantidade de luz incidente em um ponto deste objeto.

Posteriormente, Garrity [15] apresenta uma abordagem conceitualmente diferente na qual os raios lançados atravessam um volume não-estruturado com transparência, levando em conta somente a entrada destes raios no volume por faces externas ou faces da borda. Estas pertencem a apenas uma célula. Geralmente, o número de faces externas é muito menor que o total de faces. Assim, ao testar apenas as faces externas, o número de testes de interseção fica bastante reduzido. Bunyk et al. [16] aperfeiçoam esta parte do algoritmo, através do cômputo de todas as interseções dos raios com cada face externa frontal, somente uma vez. Desta forma, o algoritmo fica mais eficiente.

2.2.3 Splatting

Este método foi proposto por Westover [31, 42] com o objetivo de aumentar o desempenho dos métodos de visualização volumétrica até então existentes.

Consiste no mapeamento de cada voxel (elemento de dados do volume) no plano da imagem, ou seja, todos os voxels serão projetados neste plano. Feito isto, por meio de um processo de acumulação, a contribuição de todos os pixels que estão sob esta projeção é somada para obtenção da imagem.

Na verdade o volume de dados é representado por um vetor de funções de bases que se sobrepõem, geralmente, núcleos Gausianos (kernels) com amplitudes escaladas pelo valor dos voxels. Então a imagem é gerada pela projeção destas funções de base no plano da imagem. Para tornar este passo mais eficiente, faz-se a rasterização utilizando uma tabela de footprint pré-computada. Cada entrada na tabela de footprint armazena a função kernel integrada analiticamente ao longo de um raio percorrido. A maior vantagem do splatting é que somente os voxels que são relevantes para formação da imagem devem ser projetados e rasterizados. Isto oferece uma grande redução nos dados do volume que precisam ser processados e armazenados [43].

O enfoque tradicional de splatting [31] adiciona os kernels do voxel dentro de fatias do volume quase paralelas ao plano da imagem. Este tende a varições no brilho. Para contornar essa desvantagem Mueller *et al.* [44] propõem um método que elimina o problema acima através do processamento dos núcleos do voxel dentro de slabs (um conjunto de planos), de largura Δs, paralelamente alinhadas ao plano da imagem. Portanto, este enfoque é denominado splatting alinhado com a imagem (image-aligned splatting).

2.2.4 Shear-warp

Este método foi proposto por Lacroute e Levoy [32, 45] e era um dos mais rápidos na época. Shear-Warp faz uso do conceito de fatoração da matriz de visualização. O volume sofre uma transformação na qual todos os voxels ficam alinhados com os pixels, ou seja, cada fatia do volume fica paralela ao plano da imagem.

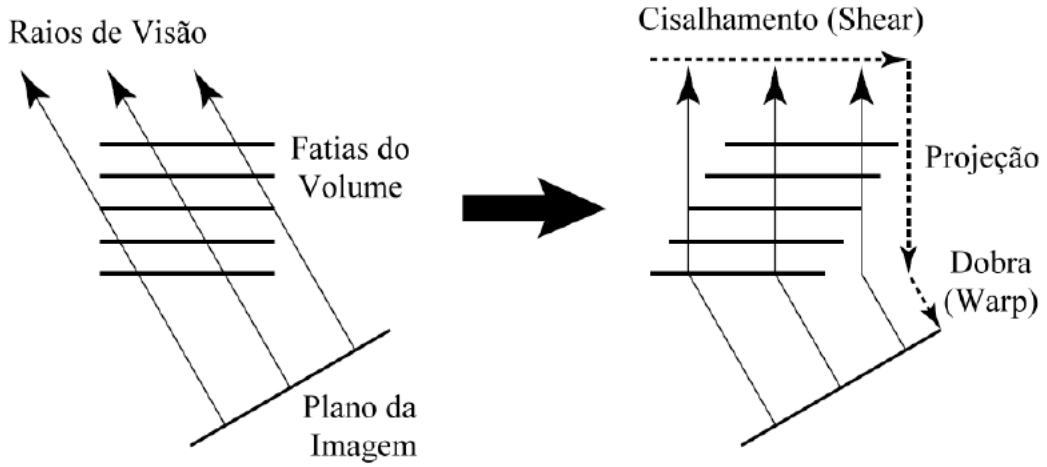

Figura 2.11: A fim de transformar um volume no espaço do objeto cisalhado por uma projeção paralela, cada fatia sofre uma translação. No espaço do objeto cisalhado pode-se projetar as fatias (slices) do voxel numa imagem de modo simples e eficiente [32].

Esta primeira etapa é o shear ou cisalhamento, descrito anteriormente. Desta forma, os raios de visão lançados atravessam as fatias do volume facilmente e uma imagem intermediária com distorção é gerada. Estas fatias são percorridas seguindo a ordem da frente para o fundo e os voxels são projetados nos pixels correspondentes. Na segunda etapa, para corrigir a imagem, aplica-se uma transformação $2D$ e então tem-se a imagem final correta. Esta estapa é o warping. As duas etapas podem ser vistas na Figura 2.11 e o esquema de composição na Figura 2.12.

2.2.5 Métodos Baseados em Mudança de Base (Domínio)

Neste método, o volume de dados passa por uma transformação para um outro domínio, onde a visualização é diretamente gerada. Estes domínios podem ser wavelets, projeção de Fourier, bases de cosseno, etc. Quando o domínio é a frequência, utiliza-se a projeção de Fourier. A projeção do volume $3D$ é obtida pelo cômputo de integrais de linha do volume ao longo de raios perpendiculares ao plano da imagem, ou seja, uma fatia perpendicular à direção de projeção é obtida aplicando-se após a transformada inversa de Fourier. Uma vantagem do método é

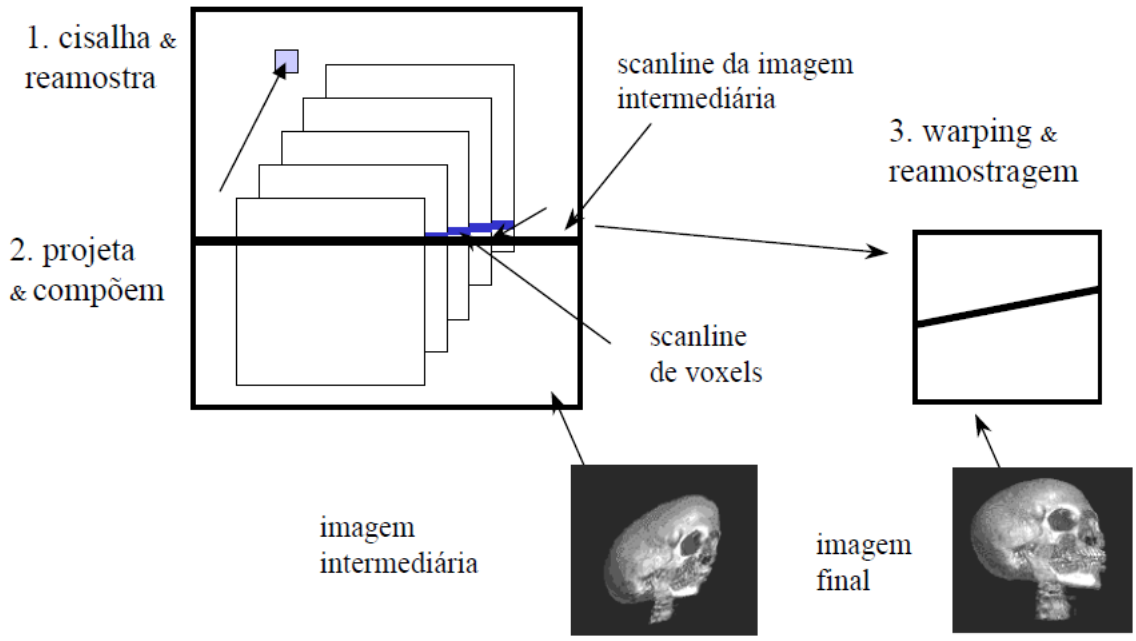

Figura 2.12: Algoritmo Shear-Warp [32].

sua aplicação para volumes muito grandes devido às características de compressão das transformadas de Fourier.

2.3 Organização da Memória Cache

Nesta seção apresentamos como a memória cache está organizada e os conceitos básicos relacionados a ela para fins de um melhor entendimento da análise de cache feita para o algoritmo de ray-casting.

2.3.1 O porquê da cache

Os pioneiros da computação corretamente predizeram que os programadores desejariam quantidades ilimitadas de memória rápida. Uma solução econômica para isso é uma *hierarquia de memória*, que tira vantagem da localidade e do custo/desempenho das tecnologias de memória. O *princípio de localidade* diz que a maioria dos programas não acessa todo o código ou dados de forma uniforme. Este princípio, juntamente com a diretiva de que hardware pequeno é mais rápido, levaram a hierarquias baseadas em memórias de diferentes tamanhos e velocidades [46].

Uma vez que a memória rápida é cara, uma hierarquia de memória é organizada em vários níveis — cada um é menor, mais rápido e mais caro por byte do que o próximo nível inferior. O objetivo é prover um sistema de memória com custo quase

tão baixo como o nível mais barato de memória e velocidade quase tão rápida como o nível mais rápido [46].

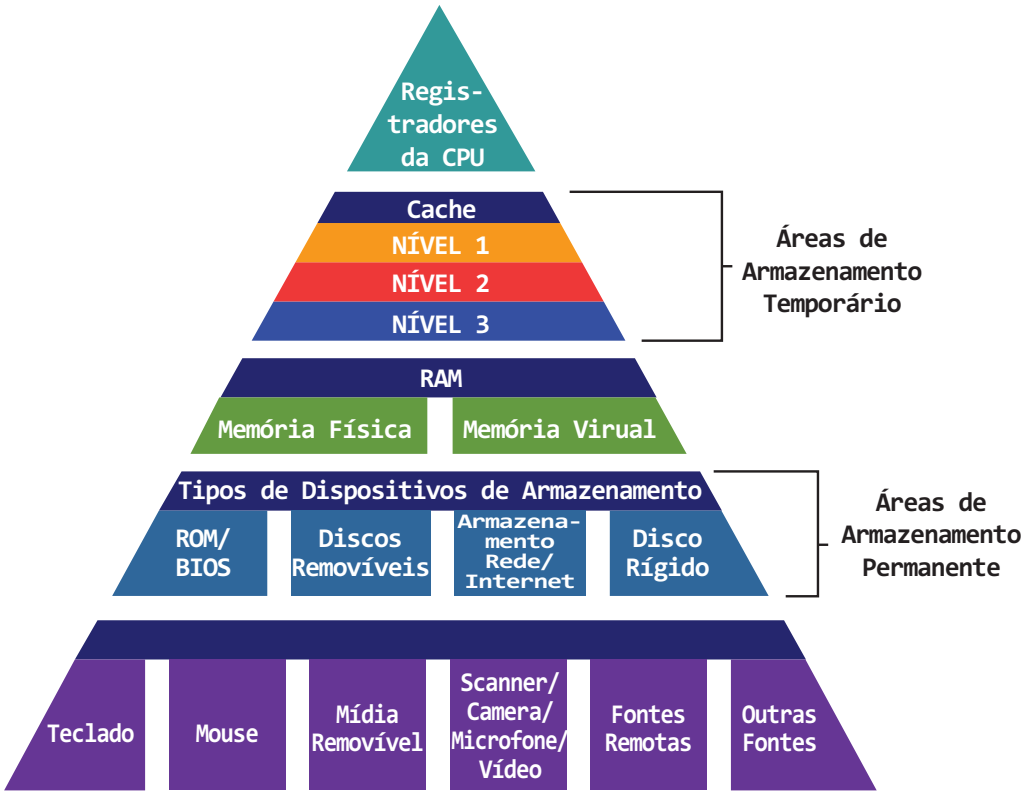

Figura 2.13: Hierarquia de Memória.

As memórias SRAM são extremamente rápidas e de alto custo, usadas para cache. Já as DRAM são mais baratas e mais lentas, usadas para a memória principal.

Com a hierarquia de memória (Figura 2.13), podemos diminuir a latência do acesso a memória principal mantendo cópias dos dados na memória rápida, tirando assim, vantagem da localidade.

O princípio de localidade [47] diz que a maioria dos programas não acessam o código ou os dados de modo uniforme, ou seja, o código do programa e os dados têm localidade temporal e espacial. Assim pode-se explorar a localidade dos acessos à memória:

- Localidade espacial: um endereço de memória tende a ser acessado se um dos seus vizinhos o for.
- Localidade temporal: um endereço de memória tende a ser acessado novamente em um curto intervalo de tempo. Desta forma, mantêm-se cópias dos dados recentemente usados. Por exemplo, num loop uma chamada a uma função é feita e esta está localizada em algum lugar no espaço de endereçamento. A função pode estar distante na memória, mas as chamadas a ela serão próximas no tempo.

Estes princípios são explorados para se definir as políticas de substituição de dados na cache, a busca em blocos e o próprio princípio de reuso.

Então, cache é o nome que se dá ao nível da hierarquia de memória que é encontrado, uma vez que o endereço sai da CPU [46]. O termo cache atualmente é aplicado sempre que um buffer é empregado a fim de se fazer o reuso dos dados. Isto acontece porque o princípio de localidade se aplica nos vários níveis, desta forma, se pode aproveitá-lo para se obter um melhor desempenho.

2.3.2 Cache em Detalhes

Apresentamos agora algumas terminologias que estamos adotando [46, 48]:

- Linha: a divisão da cache é por linhas. Cada linha tem seu índice ou endereço, bem como o tamanho de um bloco.
- Bloco: um conjunto de dados de tamanho fixo. É a quantidade de informação que é transferida por vez da memória principal para a cache. O bloco tem o mesmo tamanho da linha.
- Hit: o dado é encontrado no local desejado.
- Miss: o dado não foi encontrado no local desejado.
- Taxa de Miss: percentagem do dado que não foi encontrada na cache.
- Taxa de Hit: percentagem do dado que foi encontrada na cache.
- Latência: o tempo para a memória responder a uma requisição de leitura ou escrita.
- Largura de Banda: número de bytes que podem ser lidos ou escritos por segundo.
- Prefetch: ocorre quando o processador se antecipa a uma requisição e busca os dados que serão utilizados.
- Tag: rótulo ou etiqueta que é parte do endereço real de memória.
- Set/bloco: conjunto/bloco de linhas da cache.
- Word/Offset: palavra/deslocamento que indexa os dados na linha da cache.

O conceito de cache pode ser visto na Figura 2.14 abaixo [48]:

Na Figura 2.14 temos uma quantidade grande e lenta de memória (memória principal) e uma memória cache bem menor e mais rápida. Quando o processador

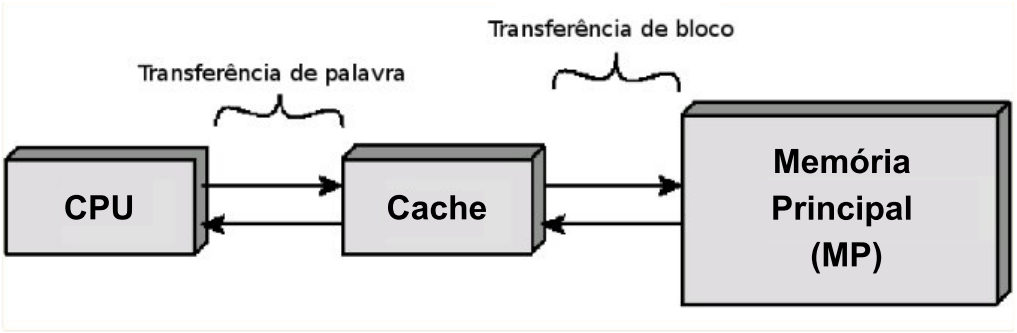

Figura 2.14: Cache e Memória Principal.

requisita a leitura de uma palavra da memória, primeiro este verifica se a palavra está na cache. Se estiver (cache hit), o processador a recebe. Do contrário (cache miss), será lido para a cache um bloco da memória e, somente então, o processador recebe a palavra.

Temos um cache hit, quando a CPU requisita um dado e este já está na cache. Do contrário, temos um cache miss. A largura de banda da memória e a latência determinam o tempo gasto no cache miss. O custo para recuperar a primeira palavra do bloco é determinado pela latência e a largura de banda que, por sua vez, determina o custo para recuperar o resto deste bloco. O hardware é responsável por tratar o cache miss, e como consequência, o processador tem de esperar até que o dado esteja disponível [46].

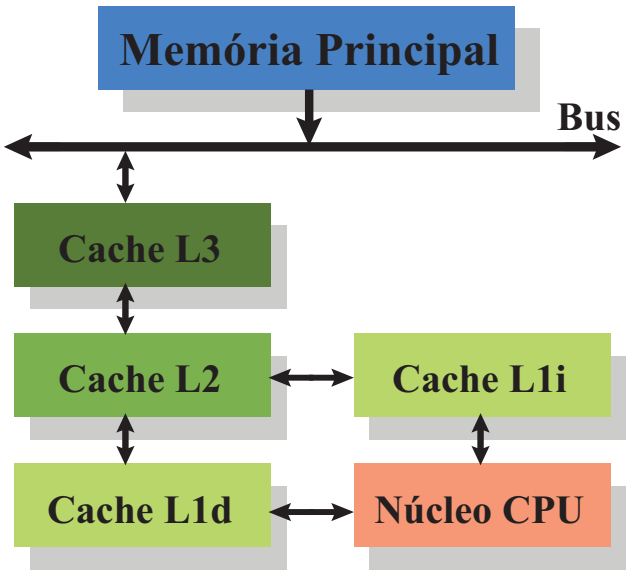

Figura 2.15: Processador com cache L3 [49].

Na Figura 2.15, temos uma visão mais atual do sistema de memória, com três níveis de cache. A terminologia utilizada na literatura é:

- L1d é o nível 1 da cache de dados.
- L1i é o nível 1 da cache de instrução.
- L2 é o nível 2 da cache. A mesma nomenclatura para dados e instrução.
- L3 é o nível 3 da cache. Idem L2.

Capítulo 3

Trabalhos Relacionados

Dentro da área de pesquisa de métodos de aceleração de *ray-casting*, surgiram duas linhas principais de pesquisa:

- enfoques em software (CPU);
- enfoques em hardware gráfico.

A primeira implementação do algoritmo de ray-casting para renderizar malhas irregulares foi proposto em software por Garrity [15]. Seu enfoque faz uso da conectividade das células a fim de computar o ponto de entrada e de saída de cada raio. Levando-se em conta que os datasets (dados volumétricos) podem não ser convexos, e assim, um determinado raio poderia entrar e sair várias vezes do volume, Garrity usa uma estrutura de dados espacial a fim de guardar as células que determinam as fronteiras do volume. Este trabalho foi aperfeiçoado posteriormente por Bunyk *et al* [16] que propoem uma técnica diferente para tratar dados irregulares. Sua solução explora a natureza discreta da imagem. Sendo assim, determinou-se que para cada pixel da tela, os fragmentos das faces frontais limítrofes do volume que são projetados, sendo armazenados numa lista ordenada, ou seja, seu aprimoramento é a computação para cada pixel de uma lista de intersecções de faces visíveis e a fácil determinação da ordem correta dos pontos de entrada para cada raio. O processo de renderização segue o método de Garrity, mas quando um raio sai da malha, o algoritmo facilmente pode determinar em qual célula o raio entrará novamente nela. Este enfoque se torna mais simples e mais eficiente do que o proposto por Garrity, entretanto, ele mantém algumas estruturas de dados auxiliares bem grandes. O trabalho recente de Pina *et al* [17] propoem dois novos algoritmos de ray-cast, ME-Raycast e EME-Raycast, que apresentam ganhos significantes e consistentes no uso de memória e na *c*orretude da imagem final sobre o enfoque de Bunyk *et al*. Seus algoritmos, entretanto, são mais lentos do que o de Bunyk. A fim de diminuir o consumo de memória que as implementações de ray-casting requerem ao ter de

manter em memória as células que são computadas, Ribeiro *et al* [18] tratam este problema otimizando o uso de memória, através da exploração da coerência de raios, que é um dos enfoques desta tese. A solução é manter, em memória, a informação das faces percorridas pelo raio lançado através de cada pixel sob a projeção de uma face visível. Desta forma, obtém-se uma redução considerável do consumo de memória e mantém-se um ótimo desempenho. Posteriormente, foram feitas duas implementações paralelas do algoritmo proposto por Ribeiro *et al* [18]. A primeira, um trabalho de Maximo *et al* [21], explora as vantagens da arquitetura de GPU, este trabalho faz parte da tese de doutorado de André de Almeida Maximo. Em seguida, teremos uma seção sobre isso falando rapidamente sobre os ganhos obtidos, e como parte das estruturas de dados são reutilizadas no aprimoramento do algoritmo proposto por Ribeiro *et al* [18]. A segunda, um trabalho de Cox *et al* [22], que fez parte da dissertação de mestrado de Guilherme Cox, explora o alto paralelismo da arquitetura do processador Cell Broadband Engine, que é uma arquitetura multicore (com vários núcleos de processamento) que foi desenvolvida pela IBM, Sony e Toshiba [20].

Recentemente, vários algoritmos de renderização de volume fazem uso da GPU a fim de alcançar altos desempenhos. Geralmente, soluções baseadas em hardware gráfico buscam atingir desempenhos em tempo real e imagens de alta qualidade. Weiler *et al* [50] apresentam um algoritmo de ray-casting baseado em hardware com base no trabalho de Garrity. Eles calculam o ponto de entrada do raio inicial através da renderização das faces frontais e depois percorrem as células usando o programa de fragmentos para armazenar estas células e o grafo de conectividade em texturas. Entretanto, o método proposto funciona somente sobre dados convexos não estruturados. Bernardon *et al* [51] melhoram o enfoque de Weiler *et al*, usando o algoritmo de Bunyk como base para a implementação em hardware e o método *depth peeling* para corrigir a renderização de malhas irregulares não convexas. Espinha e Celes [52] propõem um melhoramento no trabalho de Weiler *et al*. Os autores usam pré-integração parcial e provêem modificações interativas da função de transferência. Eles também usam uma estrutura de dados alternativa, mas como seus trabalhos são baseados nos de Weiler *et al*, o consumo de memória ainda é alto.

Em termos de outros algoritmos de renderização direta de volume, diferentes do ray-casting, é importante mencionar o enfoque de projeção de células (*cell projection*). Neste enfoque, cada célula poliedral dos dados volumétricos é projetada na tela, evitando assim, a necessidade de se manter a conectividade dos dados em memória, mas requer que as células sejam primeiramente ordenadas por ordem de visibilidade e depois compostas a fim de gerar sua cor e opacidade na imagem final. Neste escopo, existem várias implementações em CPU, tais como, Farias *et al* [6] e Max [38], que trazem flexibilidade e fácil paralelização. As implementações em GPU

de projeção de algoritmos, entretanto, são mais notórias em Marroquim *et al* [53], Wylie [41] e Weiler *et al* [50].

3.1 Trabalhos de Cache na Visualização Volumétrica

Challinger [54] faz a paralelização dos algoritmos de ray-casting e cell projection (projeção de célula), usando uma arquitetura MIMD , com memória compartilhada. Para o método de ray-casting são usados dois enfoques para a divisão de tarefas:

1. Um pixel por tarefa;
2. Uma linha de varredura (scan line) por tarefa.

Neste seu trabalho, para a paralelização do método de projeção, chega-se a conclusão de que este não tem um bom escalonamento com o número de processadores. Na medida que estes aumentam, o overhead (trabalho extra) tem um aumento significativo devido aos requisitos de sincronização para a ordenação apropriada da renderização da célula. Para o método de ray-casting usando o enfoque de scan line, obtém-se como maiores obstáculos, um desbalanceamento de carga quando o número de processadores é muito grande e a penalidade por usar memória global compartilhada.

Observa-se que ao se usar a dimensão da tarefa de um pixel como unidade de paralelismo, obtém-se um desempenho pior do que ao se usar uma cujo tamanho da tarefa é de uma scan line. Challinger supõe que isto é devido ao uso pouco eficiente do cache do processador no primeiro caso, mas não o explora em profundidade. O número de tarefas (logo, dimensão da tarefa) tem grande influência na escalibilidade, principalmente na geração de tarefas para um pixel. Apesar disso, seus trabalhos mostram que a paralelização do método ray-casting tem algumas vantagens na arquitetura utilizada. Os raios lançados pelos pixels podem ser processados de forma independente, sem restrições de ordenação. Porém é de suma importância que estes sejam agrupados a fim de se obter uma maior eficiência do processador ao se gerar uma tarefa. No enfoque scan line, para o dataset (volume de dados) utilizado, a renderização serial que é de 9 minutos cai para 12 segundos ao se usar 100 processadores. Challinger conclui que um algoritmo que faça uso melhor da memória local pode obter um desempenho que chegue a ser interativo.

Lacroute [45], Nieh e Levoy [55], Singh *et* al. [56] também não aproveitam o potencial da cache para renderização de volume.

Nieh e Levoy [55] fazem um algoritmo paralelo para renderização de volume, baseado no algoritmo de raytracing [30] usando uma arquitetura de memória com-

partilhada.Apesar de obterem ótimos resultados de speedup (quanto um algoritmo paralelo é mais rápido do que seu correspondente sequencial), não consideram, como uma das componentes de maior custo, as penalidades derivadas dos cache miss. Eles consideram que o overhead de memória não é a percentagem dominante da execução, apesar de reconheceram que o cache no DASH foi importante para reduzir o overhead de memória. Não foi feita uma análise do cache $L1$ por não terem uma ferramenta para fazer uma medição direta.

Singh *et* al. [56] também não consideram as penalidades de cache miss como foi dito acima. Suas medições de cache identificam que o working set de $4K$ de dados reduz consideravelmente a taxa de miss e que para um dataset de 256^3, o working set dever ser de $16K$. Não explora o cache $L2$ de $4MB$ do Challenge e nem faz uma avaliação dos efeitos de cache no desempenho, ao se usar um com tamanho suficiente para conter um bloco do volume.

Lacroute [45] identificam duas causas para os cache miss. A primeira, devido à comunicação no paralelismo, quando da redistruibuição dos dados e a segunda devido ao tamanho do working set ser maior que o tamanho do cache. No primeiro caso, é difícil reduzir este custo uma vez que o warp requer a mudança de pixels para diferentes partes da imagem. No segundo caso, ao aumentar o tamanho do cache num simulador para $1MB$ por processador, obtém-se uma diminuição significativa dos cache miss. Sua conclusão é de que o custo ligado à hierarquia de memória é um custo de suma importância, mas não consegue melhorar o algoritmo a fim de aumentar a localidade.

Outros trabalhos pioneiros, agora nos níveis mais altos da hierarquia de memória, foram feitos [57–61].

Em Palmer [19, 62] são estudados métodos para otimizar o desempenho de todos os níveis da hierarquia de memória, no contexto da renderização de volume usando ray-casting para dados regulares.

Foram analisados os efeitos da hierarquia de memória nos seguintes casos:

- em um único processador;
- usando memória compartilhada (shared-memory);
- usando memória distribuída (distributed-memory).

Em um único processador, o caso que é de nosso maior interesse, Palmer *et al* [19, 62] identificam como o maior custo no kernel (núcleo) do ray-casting, as penalidades de cache miss devido às leituras dos voxels do dataset. Mostram também que este alto custo é totalmente dependente da direção de visão e que esta dependência pode ser controlada através de uma blocagem apropriada dos dados.

Foram isoladas, separadamente, as componentes de cache miss $L1$ e $L2$ que contribuem para este custo. Efeitos complexos inter-pixels e intra-pixels contribuem com maior peso para a dependência das altas taxas de cache miss ligadas à direção de visão. Os efeitos de cache inter-bloco e inter-frame são ínfimos, devido às altas taxas das quais o cache é completamente liberado.

Ao agruparem os dados do volume em blocos cúbicos (de maneira bruta) e pixels em tiles (divisões quadrangulares ou retangulares da imagem), obtém uma maior localidade de memória e, portanto, um desempenho superior na hierarquia de memória em todos os níveis. O tamanho ideal do bloco para vários datasets está entre 0.5 e 1.0 vez o tamanho do cache $L2$.

Trabalhos mais recentes, Grimm *e*t al. [63], Grimm e Bruckner [64], Bruckner [65], sobre dados regulares, continuam ressaltando um melhor o uso da memória cache a fim de se obter uma melhora do desempenho.

De acordo com Grimm *e*t al. [63], os sistemas de renderização que utilizam ray-casting ainda têm, como pendente, a utilização ineficiente da CPU e a má coerência de cache. Para tratar o primeiro caso utilizam a tecnologia de hyper-threading, e para o segundo, um layout de memória em blocos. Bricking (divisão por blocos) requer um esquema eficiente de endereçamento dos dados intra-blocos e inter-blocos. A solução foi o uso de duas tabelas de consulta de endereço avançado.

Capítulo 4

Algoritmo de Ray-Casting

O algoritmo Ray-Casting [30] é um algoritmo exato de renderização volumétrica muito utilizado quando se quer obter imagens de alta qualidade. Ele opera no espaço da imagem.

Este algoritmo foi originalmente proposto por Levoy [30] e Drebin *et al* [66], em 1988, como uma técnica que permitia a visualização de pequenos detalhes internos ao volume, através do controle de transparência dos voxels, removendo trivialmente as partes escondidas atrás de partes definidas como opacas e visualizando o volume a partir de qualquer direção.

A idéia básica do algoritmo é lançar um raio através de cada pixel, a partir do observador em direção ao volume. A cor final de cada pixel da imagem é obtida integrando as contribuições de cor $C(x)$ e opacidade $\alpha(x)$ de cada voxel x interceptado pelo raio.

Na Figura 4.1, a cor do pixel correspondente ao raio, antes do cálculo da contribuição do voxel em questão, é C_{in}. Computada a contribuição de um voxel x, a cor passa a ser C_{out}:

$$C_{out} = C_{in}(1 - \alpha(x)) + C(x)\alpha(x) \tag{4.1}$$

Levoy [30] implementou este algoritmo através de dois pipelines independentes: um para iluminação e um para classificação do material, concluindo com uma fase final de composição dos dois pipelines.

Na etapa de iluminação, as componentes RGB da cor $C(x)$ para cada voxel x são calculadas a partir de uma estimativa do gradiente da função densidade $D(x)$ e da intensidade de luz, usando o algoritmo de Phong [3]. Na etapa de classificação, uma opacidade $\alpha(x)$, baseada na densidade dos materiais, é associada a cada voxel. A visualização final da imagem é feita pela projeção bi-dimensional de $C(x)$ e $\alpha(x)$ no plano de visualização.

Suponha que desejamos renderizar, por exemplo, uma imagem a partir de um

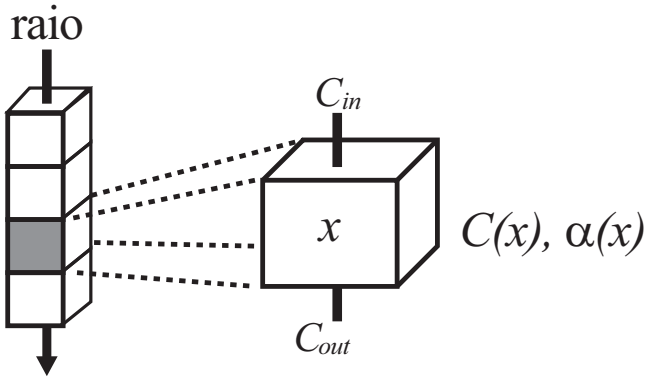

Figura 4.1: Composição da cor no voxel.

volume composto por células tetraedrais. Um raio é disparado a partir do centro de um pixel através desse volume, conforme a Figura 4.2.

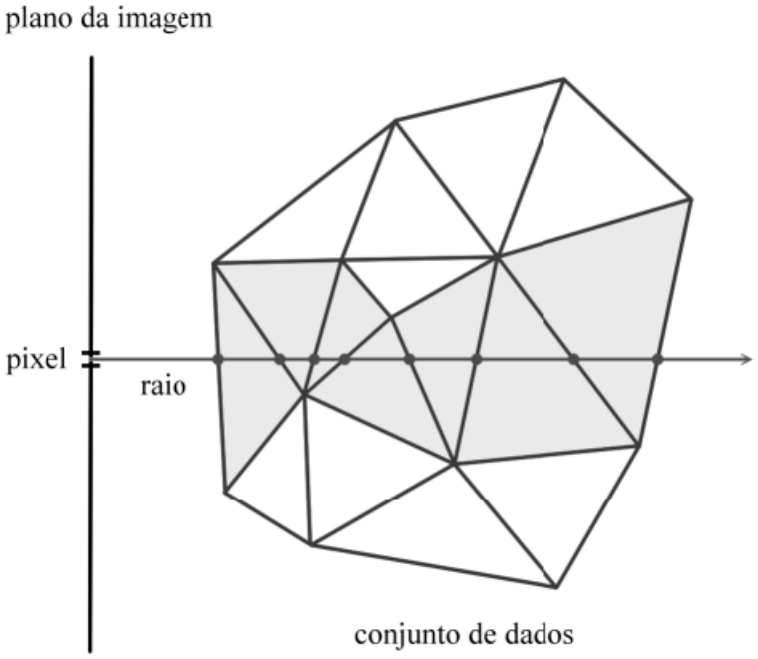

Figura 4.2: Esquema simplificado do algoritmo de ray-casting.

O primeiro passo do algoritmo é encontrar a primeira face intersectada pelo raio lançado através do pixel. Esta face é dita ser uma face externa e, a partir dela, são obtidos os valores iniciais de cor e opacidade.

Depois de encontrar a primeira face intersectada pelo raio, caminha-se através do conjunto de dados por meio de informações sobre a conectividade das células. A cada nova face intersectada pelo raio são acumulados os valores de cor e opacidade por meio da aplicação da integral de iluminação, até que o raio saia inteiramente do volume ou que o valor da opacidade acumulada atinja o valor máximo (1). O

resultado final da composição dos valores de cor e opacidade das faces por onde o raio passa é a cor do pixel.

Em geral, são utilizadas no algoritmo de Ray-Casting estruturas que contenham as células e, junto com elas, informações que permitam determinar suas células vizinhas para que, após encontrar a primeira face intersectada pelo raio, as demais sejam encontradas caminhando-se através da tetraedração por meio das adjacências. A interseção do raio com uma célula do volume pode ocorrer em uma face, em um vértice ou em uma aresta.

No algoritmo de ray-casting ocorrem casos onde é necessário ter um tratamento especial que podem ser chamados de casos degenerados. Estes casos ocorrem quando um raio intersecta a célula em um vértice ou em uma aresta. Desta forma, pode ser muito custoso descobrir quem será a próxima célula. Caso esta não seja encontrada, haverá a interrupção da composição da cor e da opacidade e, portanto, um erro será gerado na cor final do pixel em questão. No capítulo 5 discutiremos este assunto.

4.1 Enfoques Anteriores

Nesta seção descreveremos o algoritmo de ray-casting proposto por Bunyk *et al* [16], que chamaremos simplesmente de *Bunyk* e os algoritmos propostos por Pina *et al* [17], chamados *ME-Ray* (Memory Efficient Ray-Cast) e *EME-Ray* (Enhanced Memory Efficient Ray-Cast).

As estruturas de dados usadas por esses algoritmos para armazenar os dados da face guiaram seus consumos de memória e seus desempenhos. Os dados da face correspondem:

- à informação a respeito da geometria;
- aos parâmetros da face.

A geometria da face representa as coordenadas dos pontos que a definem. Os parâmetros da face são as constantes para a equação do plano, definida pela face, que serão usadas para computar a interseção entre o raio e a face. O dado da face é armazenado em uma estrutura que será chamada de *face*. Os parâmetros da face têm um alto custo para serem computados e seu armazenamento é responsável pelo maior consumo de memória nos algoritmos de ray-casting.

4.1.1 Bunyk

Na etapa de pré-processamento todos os pontos e tetraedros do conjunto de dados de entrada são lidos. Uma lista com todos os vértices e uma lista com todas as células são criadas. Além disso, para cada tetraedro de entrada, cada uma de suas

faces é armazenada em uma lista. Bunyk também mantém, para cada vértice, uma lista de todas as faces que usam este vértice, chamada lista `referredBy`. Depois de ler todo o conjunto de dados, Bunyk determina quais faces pertencem ao lado visível da borda da cena. Estas são as faces visíveis. O algoritmo projeta essas faces visíveis na tela criando, para cada pixel, uma lista de interseções, que será usada como ponto de entrada de seus raios. Depois que os pontos de entrada são armazenados, realmente começa o ray-casting.

Para cada pixel a primeira interseção é o ponto de entrada nos dados e cada próxima interseção é obtida inspecionando-se a interseção entre o raio e todas as outras faces da célula corrente. Cada par de interseções é usado para computar a contribuição da célula para a cor e a opacidade do pixel. A busca pela próxima interseção no método de Bunyk é bem rápida, uma vez que esta busca é executada entre as outras três faces do tetraedro. Entretanto, a lista de todas as faces e as listas `referredBy` resultam em um consumo muito alto de memória.

4.1.2 ME-Ray

O método *ME-Ray* também inicia com uma etapa de pré-processamento, lendo o conjunto de entrada e criando algumas estruturas de dados. Ele cria uma lista de todos os vértices, uma lista de todas as células e a lista `Use-Set` para cada vértice. As listas `Use-Set` substituem as listas `referredBy` usadas por Bunyk. Cada `Use-Set` contém uma lista de todas as células incidentes no vértice.

Outro passo na fase de pré-processamento é criar, para cada célula, a lista de todas as células que compartilham uma face com ela. No algoritmo de renderização, *ME-Ray* também cria uma lista de faces visíveis e determina o ponto de entrada de cada raio, da mesma forma que Bunyk o faz. O algoritmo prossegue procurando pela próxima intereseção nas células vizinhas. *ME-Ray* cria uma lista de faces sob demanda na medida em que a face é intersectada pelo raio. As faces que nunca são intersectadas por qualquer raio não serão criadas. Depois que cada par de interseções é encontrado, o *ME-Ray* computa sua contribuição para a cor final e a opacidade. O *ME-Ray* pode economizar aproximadamente 40% da memória usada por Bunyk, mas é mais lento, uma vez que este já criou as faces logo no início do processo, antes da renderização.

4.1.3 EME-Ray

O *EME-Ray* foi desenvolvido como uma otimização do *ME-Ray* em termos de uso de memória, pois possuem um comportamento similar. Entretanto, o *EME-Ray* não armazena as faces na memória, uma vez que esta foi uma das estruturas com o maior custo de memória no *ME-Ray*. Consequentemente, o *EME-Ray* tem de recalcular

os parâmetros das faces para a verificação da interseção do raio a cada vez que uma nova face é criada. Com esta otimização, o *EME-Ray* usa somente em torno de 1/4 da memória usada por *Bunyk*. As reduções significantes no uso de memória vêm com o aumento no tempo de execução. O *EME-Ray* geralmente dobra o tempo de execução do Bunyk. É importante notar que os ganhos do *ME-Ray* e do *EME-Ray* sobre Bunyk não são somente em uso de memória, mas também, na corretude da imagem final. Suas estruturas de dados lhes permitem lidar com todos os casos degenerados (seção 5.3) que Bunyk não foi capaz de resolver.

Capítulo 5

Algoritmo de Ray-Cast Orientado por Face Visível

Comparando os três enfoques de Ray-Casting descritos no capítulo anterior, podemos observar que: quanto maior for o número de faces armazenadas na memória mais rápido o algoritmo computa a interseção do raio. Por esta razão, Bunyk ainda é a implementação de software mais rápida do algoritmo de Ray-Casting. Não obstante, o consumo de memória do Bunyk torna a sua implementação em hardware gráfico proibitivo devido a quatidade limitada de memória das GPUs atuais.

Nosso algoritmo, chamado *Ray-Cast Dirigido por Face Visível*, **VF-Ray**, trata o problema de manter a informação sobre as faces em memória, degradando minimamente o desempenho de renderização. A idéia básica por trás do *VF-Ray* é manter na memória principal somente as faces dos percorridos mais recentes. Para tal, nós exploramos a coerência de raio, usando a informação da face visível para guiar a criação e a destruição das faces na memória.

O artigo do algoritmo do *VF-Ray* foi publicado no SIBGRAPI 2007 [18].

5.1 Explorando a Coerência de Raio

O *VF-Ray* é baseado na coerência dos raios. Esta coerência vem do fato de que o conjunto de faces intersectadas por dois raios, que tem como origem pixels vizinhos, conterá quase as mesmas faces. Nosso algoritmo tira vantagem da coerência do raio mantendo em memória somente as faces de raios próximos.

Um fato importante na exploração da coerência do raio é a determinação do conjunto de raios que provavelmente tem o mesmo conjunto de faces intersectadas. Nós usamos a informação das faces visíveis para fazer isso. O conjunto de raios que pode reutilizar a mesma lista de faces são os que correspondem ao conjunto de pixels sob a projeção de certa face visível. Esse conjunto de pixels será chamado

aqui de conjunto visível, *visible set*. A Figura 5.1 mostra um exemplo em $2D$ desta idéia. Os raios que são lançados através da face visível, apresentados na Figura 5.1, tendem a intersectar as mesmas faces internas. Consequentemente, as faces internas são criadas e armazenadas uma vez para a primeira interseção de raio. Depois disso, elas são reusadas até que todos os pixels do *visible set* sejam computados. Este processo é repetido para todas as faces visíveis. A fim de garantir a corretude, todas as faces visíveis são ordenadas em profundidade.

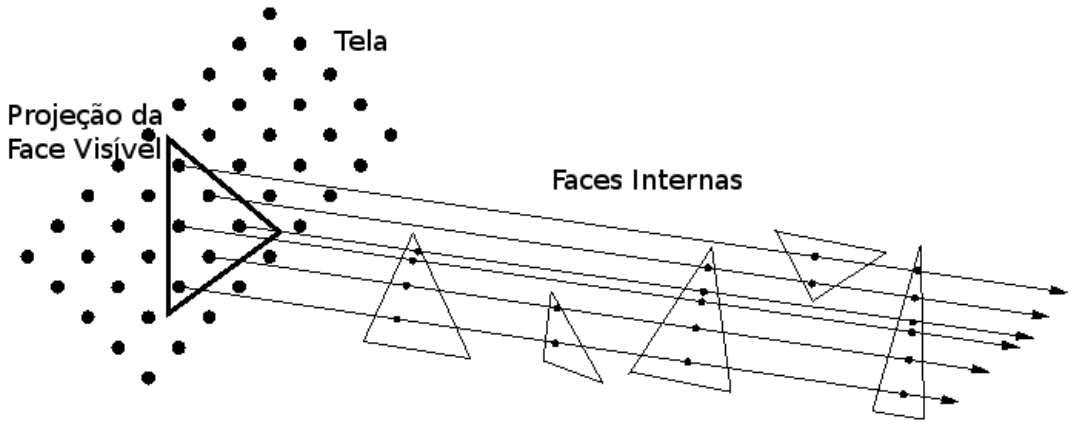

Figura 5.1: Coerência de raios da face visível.

Nossos resultados mostram que explorar a coerência de pixels reduz consideravelmente o uso de memória, enquanto mantém competitivo o desempenho do VF-Ray com os mais rápidos algoritmos anteriores, pois a reutilização dos dados das faces melhora o desempenho da cache.

5.2 Estruturas de Dados

As estruturas de dados usadas pelo *VF-Ray* são similares às do EME-Ray:

- um array de vértices;
- um array de células;
- o `Use-set` de cada vértice.

Além destas estruturas o *VF-Ray* também tem uma lista chamada *computedFaces* que armazena as faces internas através do modelo. Observe que esta lista é esvaziada depois do cômputo do ray-Casting de todos os pixels de um *visible set*. Dessa forma, não precisamos armazenar todas as faces internas, como é feito nos algoritmos de Bunyk e do *ME-Ray*.

Além disso, da mesma forma que o *ME-Ray* e o *EME-Ray*, nosso algoritmo consegue tratar qualquer tipo de célula convexa. Para malhas tetraedrais, as interseções

são computadas entre a linha definida pelo caminho do raio e o plano definido pelos três vértices de uma face triangular. Para malhas hexaedrais, em que cada face é quadrangular, a interseção é encontrada através da divisão dessas faces quadrangulares em duas faces triangulares e o mesmo cálculo é executado para cada uma.

5.3 Tratamento de Casos Degenerados

As situações degeneradas, que podem ser encontradas durante o caminho percorrido por um raio, são tratadas da mesma forma que foi executada em Pina *et al* [17], através da investigação do `Use-set` de cada vértice. Assim, quando um raio acerta um vértice ou aresta, o algoritmo consegue determinar a próxima interseção corretamente, como veremos a seguir:

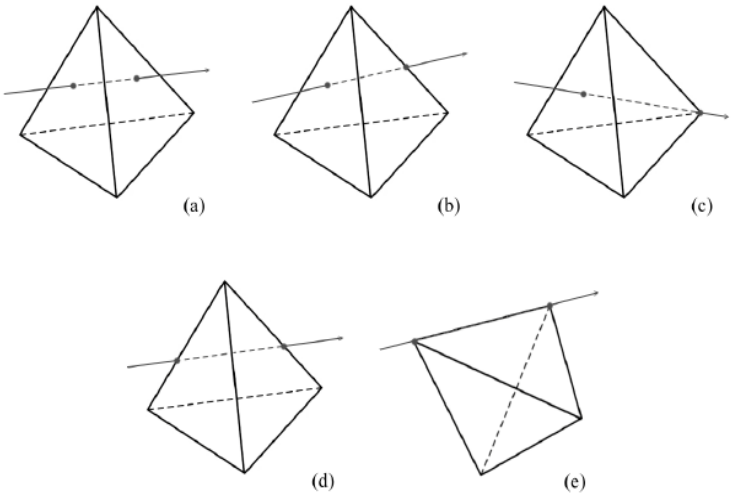

Figura 5.2: Casos de interseção do raio com uma célula.

Um problema que ocorre no algoritmo de Ray-Casting é que, quando o raio intersecta a célula em um vértice ou em uma aresta (Figura 5.2), pode ser muito difícil determinar qual será a próxima célula e, se ela não for encontrada, a composição de cor e opacidade será interrompida, causando um erro na cor final do pixel.

Existem 5 casos de interseção do raio com uma célula (Figura 5.2). O raio intersecta:

1. duas faces (5.2-a);
2. uma face e uma aresta (5.2-b);
3. uma face e um vértice (5.2-c);
4. duas arestas da mesma face (5.2-d);
5. dois vértices da mesma aresta (5.2-e).

Para verificar se o raio lançado pelo pixel intersecta uma face, foi utilizado um procedimento que faz a transformação da face para o plano xy e verifica se o ponto lançado através do pixel (x,y) está nela, simplificando o problema de interseção do raio com uma face 3D. No entanto, quando é necessário encontrar a próxima face intersectada pelo raio, duas situações podem ocorrer:

- encontrar mais de uma face;
- não encontrar uma face cuja coordenada z de profundidade seja maior que a atual.

Isso acontece se um raio intersecta um vértice ou uma aresta. Nesse caso, o programa tenta encontrar a próxima face entre todas as faces de todas as células adjacentes à atual.

5.4 O Algoritmo

O algoritmo *VF-Ray* inicia com a leitura do conjunto de dados e cria três listas:

- uma lista de vértices;
- uma lista de células;
- uma lista `Use-Set` para cada vértice.

No processo de renderização *VF-Ray* cria uma lista de todas as faces visíveis, que são as faces cujas normais fazem um ângulo menor do que 90° com a direção de visão. Para cada face visível, o algoritmo segue os passos apresentados na Figura 5.3.

Inicialmente, a face visível, $vface_i$, é projetada na tela, gerando o *visible set* de $vface_i$. Para cada pixel p no *visible set*, o algoritmo computa primeiro a interseção entre o raio que sai de p e a face $vface_i$. Depois que esta primeira interseção é achada, o caminhamento do raio se inicia, quando o algoritmo computa as interseções do raio com as faces internas. A próxima face intersectada, $face_j$, é encontrada em uma outra face da célula corrente. Se $face_j$ não foi criada anteriormente (esta é a primeira interseção na $face_j$), a face é criada e inserida na lista `computedFaces`. Do contrário, os dados da $face_j$ são carregados a partir desta lista. As operações de inserção e carga na lista são feitas em tempo constante, uma vez que nós armazenamos na célula o índice da posição de cada face na lista. O cálculo das inteseções é feito usando-se os parâmetros da face computados quando esta foi criada. Para cada interseção o valor escalar é interpolado entre os três vértices de cada face, os quais são usados na integral de iluminação proposta por Max [38].

```
For cada face visível vface_i Do
  Projetar vface_i; // Determina conjunto visível
  For cada pixel p do conjunto visível Do
    Intersectar vface_i; // Interpola valor escalar
    Do
      Achar próxima face interna face_j; // Interseção
      If face_j not in computedFaces then
        Criar face_j e Inserir em computedFaces;
      else
        Carreguar face_j de computedFaces;
      Integração do Raio; // Modelo óptico
    While existir próxima face face_j;
  End For
  Limpar computedFaces; // Limpar lista
End For
```

Figura 5.3: Pseudocódigo do *VF-Ray*.

Capítulo 6

EVF-Ray – O VF-Ray Otimizado

Como já vimos, o *VF-Ray* tem um bom desempenho com um baixo custo de memória. A fim de adaptar o *VF-Ray* em GPU [21], devido às restrições de memória da arquitetura CUDA [67], foi preciso reestruturar a estrutura de dados original. A nova versão do *VF-Ray*, chamada **EVF-Ray** (Enhanced VF-Ray), utiliza a nova reestruturação, mas com certas restrições conforme veremos.

Na etapa de pré-processamento é feita a leitura dos tetraedros e seus respectivos vértices. Ambos são armazenados na lista de vértices (vertList) e na lista de tetraedros (tetList). A partir destas listas, é construída a conectividade dos tetraedros (conTet), similarmente como é feito nos trabalhos de Garrity [15] e Bunyk *et al.* [16]. Nesta lista, para cada uma das faces, é armazenado o índice do tetraedro que compartillha a mesma face. Quando a face a ser armazenada for uma face externa, simplesmente armazena-se o próprio índice do tetraedro.

Podemos ter uma visão da estrutura de dados na Figura 6.1, para um tetraedro. Cada vértice do tetraedro é representado por um quarteto (x, y, z, s) do qual o trio (x, y, z) são as coordenadas e s seu escalar. Como cada tetraedro é composto por 4 vértices, a lista *tetList* armazena seus respectivos índices. Da mesma forma que o *VF-Ray*, continua-se tendo a lista de faces externas que é pré-computada. Sua determinação é feita de forma semelhante ao feito por Bunyk *et al.* [16].

No algoritmo original cada célula armazena a posição na lista de faces já computadas, *computedFaces*, de cada uma de suas faces. Este armazenamento continua a existir, agora numa lista chamada *compFaceTet*. Os parâmetros das faces, pertencentes à lista de faces externas, são igualmente computados. O valor escalar de um dado ponto é obtido através de interpolação. A partir do conjunto de faces externas são determinadas as faces que são visíveis. Uma face externa é dita visível, se o valor interpolado da coordenada z da projeção do vértice oposto for menor que o valor da coordenada z deste vértice.

O *EVF-Ray* não faz a indexação da lista de faces já computadas,*computedFaces*, através de uma tabela de dispersão como é feito por sua versão em GPU. A indexação

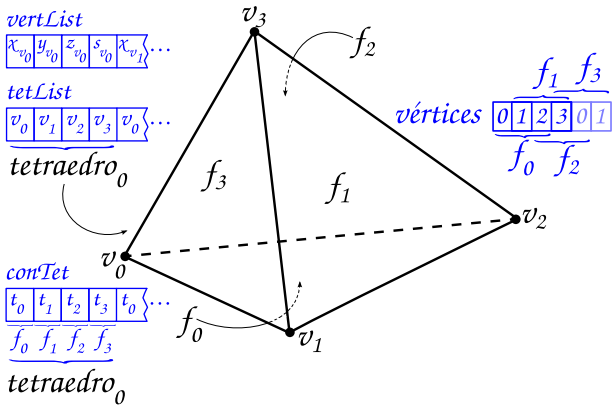

Figura 6.1: Estruturas de dados base do EVF-Ray em CPU. Fonte [21]

está armazenada na lista *compFaceTet* e é feita da mesma forma que o *VF-Ray*. Desta forma, as operações de inserção e leitura de uma face na lista *computedFaces* é feita em tempo constante.

O tratamento de casos degenerados não é similar ao feito no *VF-Ray*. No algoritmo original é preciso armazenar para cada vértice, a lista de células incidentes chamada *Use_set*. Assim, quando um raio intercepta uma aresta ou um vértice, percorre-se seu *Use_set* a fim de determinar qual o tetraedro seguinte. O *EVF-Ray* utiliza o mesmo enfoque de sua versão em GPU. Quando um caso degenerado ocorre, o raio sofre uma pequena perturbação e na iteração seguinte, este prossegue na mesma direção. A perturbação não é levada em conta nesta iteração.

O *EVF-Ray* determina a face de saída (próxima face), através da projeção do vértice oposto sobre a face de entrada. Esta solução é mais eficiente do que a empregada pelo *VF-Ray*, que computa os parâmetros de cada uma das possíveis faces de saída, uma por vez.

Resumindo, o *EVF-Ray* ficou mais *leve*, pois consome menos memória do que sua versão original e mantém um ótimo desempenho. O *VF-Ray* utiliza de 27% a 133% mais memória do que o *EVF-Ray*.

Capítulo 7

Avaliação de Desempenho e Consumo de Memória

Neste capítulo será avaliado o desempenho e o consumo de memória do *EVF-Ray* quando comparado ao *VF-Ray*, *ME-Ray*, *EME-Ray* e *Bunyk*. Também será avaliado o desempenho e o consumo de memória do *VF-Ray* quando comparado ao *ME-Ray*, *EME-Ray* e *Bunyk*. Será apresentado também uma comparação entre *VF-Ray* e um algoritmo de projeção de célula, *ZSweep* [6]. Esta última comparação foi incluída a fim de colocar nossos resultados em perspectiva, com respeito a outro paradigma de renderização. A idéia principal do algoritmo *ZSweep* é a varredura dos dados com um plano paralelo ao plano de visão xy, na direção positiva do eixo z. O processo de varredura é feito ordenando-se os vértices pelos valores das suas coordenadas z, em ordem crescente. Eles são recuperados, um a um, a partir desta estrutura de dados. Para cada vértice varrido pelo plano de varredura, o algoritmo projeta na tela, todas as faces a ele incidentes.

7.1 Ambiente de Testes

O algoritmo *VF-Ray* foi escrito em C++ ANSI sem usar qualquer biblioteca gráfica particular. Nossos experimentos foram conduzidos em um computador com processador Intel Pentium IV de 3.6 GHz com 2 GB de RAM e 1 MB de cache L2, rodando sobre o Linux Fedora Core 5.

Nós usamos quatro conjuntos de dados diferentes: Blunt Fin, Liquid Oxygen Post, Delta Wing e SPX. A tabela 7.1 mostra o número de vértices, faces, faces sobre a superfície e células para cada conjunto de dados. Os modelos renderizados com o *VF-Ray* podem ser vistos na Figura 7.2. Nós também variamos os tamanhos das imagens de 512×512 até 8192×8192.

Dataset	# Verts	# Faces	# Tets	# Boundary
Blunt	41 K	381 K	187 K	13 K
SPX	149 K	1.6 M	827 K	44 K
Post	109 K	1 M	513 K	27 K
Delta	211 K	2 M	1 M	41 K

Tabela 7.1: Dimensões dos dados de entrada.

7.2 Consumo de Memória

A tabela 7.2 apresenta a memória utilizada pelo *VF-Ray* para renderizar um frame (em MBytes) para os modelos Blunt, SPX, Post, Delta e os resultados do uso da memória relativa do *ME-Ray*, *EME-Ray*, *Bunyk* e *ZSweep* quando comparados com o uso do *VF-Ray*.

As porcentagens apresentadas nesta tabela correspondem à razão do uso de memória dos outros algoritmos sobre o consumo de memória do *VF-Ray*. Em outras palavras, nós consideramos os resultados do *VF-Ray* como sendo 100% e estamos apresentando quanto os outros algoritmos crescem ou decrescem este resultado.

Dataset	Resolução	Memory (MB)	ME-Ray	EME-Ray	Bunyk	ZSweep
Blunt	512×512	14	404%	166%	528%	208%
	1024×1024	30	240%	129%	297%	177%
	2048×2048	39	333%	251%	377%	369%
	4096×4096	75	495%	451%	517%	689%
	8192×8192	219	610%	655%	617%	917%
SPX	512×512	96	215%	74%	299%	87%
	1024×1024	99	234%	88%	305%	107%
	2048×2148	108	271%	141%	337%	188%
	4096×4096	144	383%	284%	428%	407%
	8192×8192	288	533%	498%	569%	711%
Post	512×512	63	165%	74%	290%	97%
	1024×1024	66	203%	95%	301%	129%
	2048×2048	74	281%	172%	353%	238%
	4096×4096	110	424%	349%	470%	499%
	8192×8192	256	601%	560%	603%	800%
Delta	512×512	116	167%	74%	303%	97%
	1024×1024	119	200%	84%	308%	116%
	2048×2048	129	246%	124%	330%	177%
	4096×4096	165	346%	247%	403%	375%
	8192×8192	307	500%	467%	534%	704%

Tabela 7.2: Consumo de memória do VF-Ray versus ME-Ray, EME-Ray, Bunyk e ZSweep.

Como podemos observar na tabela 7.2, na maioria dos casos, *VF-Ray* utiliza menos memória do que os outros três algoritmos. Quando comparado a *Bunyk*, *VF-Ray* usa de 1/3 a 1/6 da memória utilizada por este. Estes ganhos são consideráveis. Para o modelo Blunt, por exemplo, *VF-Ray* renderiza uma imagem de 4096×4096

usando a mesma quantidade de memória que *Bunyk* utiliza para renderizar uma imagem de 512×512. Para o maior conjunto de dados, Delta, *Bunyk* usa mais memória para renderizar uma imagem de 512×512 do que o *VF-Ray* usa para renderizar uma imagem de 8192×8192.

Quando comparado com os dois algoritmos sensíveis à memória, *ME-Ray* e *EME-Ray*, podemos observar que o *ME-Ray* usa de 1.5 a 6 vezes mais memória do que o *VF-Ray*. Para o *EME-Ray*, nós percebemos um resultado interessante. Embora o *EME-Ray* não armazene as faces na memória, para imagens com alta resolução – maiores do que 1024×1024 – *EME-Ray* usa mais memória do que *VF-Ray*. Este resultado pode ser explicado pelo fato de que o *EME-Ray* mantém, exatamente como *Bunyk* e *ME-Ray*, para cada pixel, uma lista contendo as faces de entrada para este. O número de listas cresce com o aumento da resolução da imagem. *VF-Ray*, por outro lado, não requer este tipo de lista, uma vez que este computa o ponto de entrada do raio sob demanda, para cada face visível.

Quando comparado ao *ZSweep*, podemos observar que, exceto para o modelo SPX com uma imagem de 512×512, o *VF-Ray* usa muito menos memória do que o *ZSweep*. Para as imagens com resolução mais alta, o *ZSweep* consegue utilizar de 8 a 9 vezes mais memória que o *VF-Ray*. Para renderizar o SPX com uma imagem de 8192×8192, por exemplo, o *ZSweep* gasta aproximadamente $2GB$ de memória. O consumo de memória do *ZSweep* é diretamente proporcional a resolução da imagem final, já que este cria listas de pixels para armazenar as interseções encontradas para cada um.

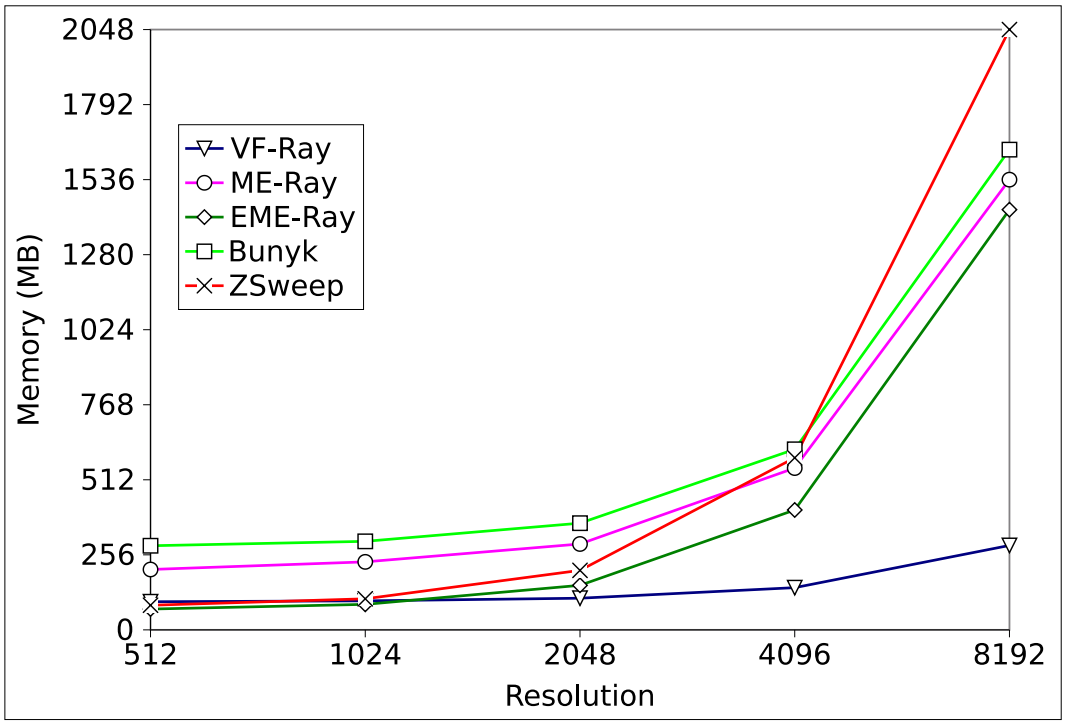

Figura 7.1: Consumo de memória para o dado SPX.

7.3 Tempo de Execução

É importante notar que enquanto que os outros algoritmos têm um crescimento linear do consumo de memória, com o aumento da resolução da imagem, o *VF-Ray* aumenta linearmente. A Figura 7.1 mostra o aumento do uso de memória de todos os algoritmos testados para as diferentes resoluções de imagens. Embora o eixo X do gráfico esteja fora de escala, devido às grandes diferenças de resoluções utilizadas, podemos notar a grande diferença no crescimento do uso da memória com o aumento da resolução da imagem de *VF-Ray* em relação aos outros algoritmos.

A tabela 7.3 apresenta o tempo gasto pelo *VF-Ray* para renderizar um frame (em segundos) para os modelos Blunt, SPX, Post, Delta e os resultados dos tempos relativos do *ME-Ray*, *EME-Ray*, *Bunyk* e *ZSweep* quando comparados com o tempo de execução do *VF-Ray*. Da mesma forma que a tabela anterior, as porcentagens apresentadas nesta tabela correspondem à razão do tempo de execução dos outros algoritmos sobre o tempo de execução do *VF-Ray*.

Dataset	Resolução	Time (sec)	ME-Ray	EME-Ray	Bunyk	ZSweep
Blunt	512 × 512	1.9	139%	296%	105%	253%
	1024 × 1024	7.0	143%	317%	94%	431%
	2048 × 2048	27.2	146%	337%	88%	481%
	4096 × 4096	107.4	147%	328%	88%	516%
	8192 × 8192	426.7	154%	341%	91%	382%
SPX	512 × 512	4.1	111%	161%	76%	287%
	1024 × 1024	13.0	103%	203%	81%	202%
	2048 × 2048	46.6	103%	225%	83%	219%
	4096 × 4096	177.1	105%	234%	82%	226%
	8192 × 8192	696.3	105%	238%	81%	260%
Post	512 × 512	5.0	138%	251%	80%	201%
	1024 × 1024	19.5	136%	254%	76%	167%
	2048 × 2048	78.6	133%	258%	74%	194%
	4096 × 4096	309.9	135%	257%	74%	227%
	8192 × 8192	1246.3	135%	263%	72%	246%
Delta	512 × 512	3.1	130%	242%	91%	465%
	1024 × 1024	11.2	122%	270%	88%	271%
	2048 × 2048	41.9	121%	280%	87%	312%
	4096 × 4096	162.7	120%	290%	84%	341%
	8192 × 8192	640.3	123%	290%	83%	369%

Tabela 7.3: Resultados de tempo para o VF-Ray versus ME-Ray, EME-Ray, Bunyk e ZSweep.

Como podemos observar na tabela 7.3, o *VF-Ray* supera em desempenho o *ME-Ray*, *EME-Ray* e o *ZSweep* para todos os conjuntos de dados e para todas as resoluções de imagens. Para os modelos Blunt, Post e Delta, o *VF-Ray* é cerca de 3 vezes mais rápido do que o *EME-Ray* e o *ZSweep*, e usa em torno de 3/4 do tempo utilizado pelo *ME-Ray* para renderizar a imagem. Para o modelo SPX, os

ganhos do *VF-Ray* contra o *EME-Ray* e o *ZSweep* são menores, mas ainda elevados. Quando comparado ao *Bunyk*, para os modelos Blunt e Delta, o *VF-Ray* apresenta a diferença no tempo de execução de somente 10%. Para os modelos SPX e Post, o *VF-Ray* é mais lento somente em torno de 22% comparado ao *Bunyk*.

Observando o aumento no tempo de renderização do VF-Ray para diferentes resoluções de imagens, podemos notar que o tempo de renderização aumenta na mesma proporção que o aumento da resolução da imagem. Os ganhos de VF-Ray sobre os outros algoritmos são, entretanto, praticamente os mesmos para diferentes resoluções.

7.4 Discussão

O VF-Ray obteve ganhos consistentes e significativos no uso de memória sobre Bunyk, provendo quase a mesma performance (a diferença entre os seus tempos de execução é em torno de 16% somente). Na medida em que a resolução da imagem aumenta, nossos ganhos em uso de memória sobre Bunyk são ainda mais pronunciados, mas a diferença no tempo de renderização se mantém. Para imagens de 8192×8192, Bunyk usa aproximadamente 6 vezes mais memória do que o VF-Ray, com uma diferença de performance em torno de 18% somente. Em nossos experimentos, entretanto, estamos comparando execuções somente onde todo o conjunto de dados cabe na memória principal, para todos os algoritmos. Nossa plataforma experimental tem 2GB de memória principal que pode armazenar todas as estruturas de dados usadas na nossa carga de trabalho. Na medida em que o consumo de memória aumenta, a renderização precisará utilizar os mecanismos de memória virtual do sistema operacional, que teria grande influência sobre o tempo de execução total.

O pequeno aumento no tempo de execução, quando comparado com Bunyk, vem do fato de que Bunyk computa todas as faces externas na etapa de pré-processamento, antes do loop de ray casting, enquanto que VF-Ray o faz sob demanda, na medida em que a face é intersectada pela primeira vez. Além disso, no VF-Ray, se uma face for intersectada por raios de duas faces visíveis diferentes, este terá seus dados computados duas vezes.

A comparação do VF-Ray com o ME-Ray e o EME-Ray proveu resultados interessantes. ME-Ray e EME-Ray são algoritmos sensíveis à memória, projetados para reduzir o grande consumo de memória de Bunyk. VF-Ray foi projetado para reduzir o consumo de memória do ME-Ray, próximo ao usado pelo EME-Ray, mas executando mais rápido do que o EME-Ray. Nossos resultados mostraram, porém, que para imagens com maior resolução, nosso algoritmo não é somente mais rápido que o ME-Ray, mas também usa menos memória do que o EME-Ray. Os ganhos em

consumo de memória do VF-Ray sobre o EME-Ray vêm da eliminação das listas de faces de entrada por cada pixel. Os ganhos em tempo de execução do VF-Ray sobre o EME-Ray vêm da boa performance de cache do VF-Ray. Nós fizemos alguns experimentos para avaliar os cache misses do VF-Ray e do ME-Ray e obtivemos para o VF-Ray uma taxa de cache miss muito baixa, próxima a 0%, enquanto que a taxa de cache miss do ME-Ray é em torno de 1.9% para imagens com tamanho significativo. Este resultado foi obtido porque nosso algoritmo reutiliza as faces para os raios na mesma face visível. Dado ao fato de que os raios são lançados um após o outro, quanto mais próximo os raios vizinhos estiverem uns dos outros, maior será a probabilidade de que eles reutilizem os dados no cache.

A comparação do VF-Ray com enfoque de projeção de célula, implementado pelo ZSweep, mostrou que o VF-Ray é mais rápido e gasta menos memória que o ZSweep. Os ganhos são significativos. Para o conjunto de dado Delta, o ZSweep usa 7 vezes mais memória, executando em torno de 3.5 vezes mais lento que o VF-Ray. A lista de pixel mantida pelo enfoque do ZSweep aumenta enquanto um *alvo-Z* não foi alcançado. Na medida em que esta lista cresce, a operação de inserção é mais custosa, devido à cache misses, uma vez que ela deve ser mantida ordenada. Por outro lado, o algoritmo VF-Ray não depende de ordenação. A diferença no uso de memória do ZSweep é devido ao aumento nestas listas de pixels. Para o ZSweep na medida em que o tamanho da imagem aumenta, cada face projetada irá inserir unidades de interseção em mais listas de pixels. Se o *alvo-Z* não é apropriado, as listas de pixels aumentarão consideravelmente.

Em termos das características do conjunto de dados, o desempenho do VF-Ray aumenta na medida em que o número de pixels em cada face aumenta. Um conjunto de dados com número menor de faces tem uma quantidade maior de pixels por face visível. Consequentemente, conjunto de dados, como Blunt, que tem um número menor de faces ($381K$), veja tabela 7.1 tendem a apresentar melhores resultados de performance para o VF-Ray.

Finalmente, é importante ter em mente que as reduções no consumo de memória permitiram a implementação do algoritmo no hardware gráfico [21]. Além disso, essas reduções também permitirão o uso de precisão dupla nos parâmetros para calcular a interseção entre um raio e uma face, podendo obter imagens mais precisas.

7.5 Resultados de Desempenho do EVF-Ray

O algoritmo *EVF-Ray*, uma otimização do *VF-Ray*, também foi escrito em C++ ANSI sem usar qualquer biblioteca gráfica particular. Nossos experimentos foram conduzidos em um computador com processador Intel Core 2 Duo, de 2.0 GHz, com 4 GB de RAM, 32 KB de cache L1 e 4096 KB de cache L2, rodando sobre o Linux

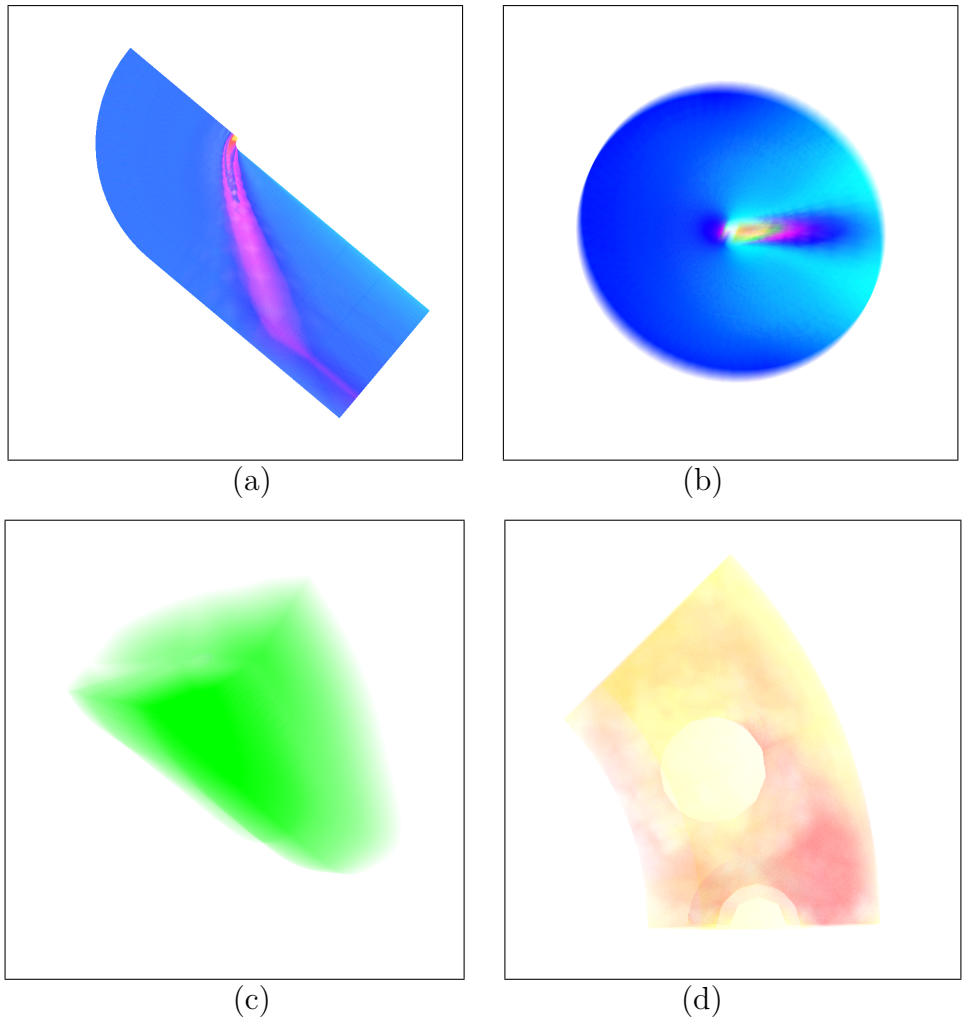

Figura 7.2: Visualização dos seguintes volumes: *Blunt Fin* (a), *Oxygen Post* (b), *Delta Wing* (c), *SPX* (d).

Ubuntu 10.04.

A Tabela 7.4 apresenta a memória consumida pelo *EVF-Ray* para renderizar um frame (em MBytes) para os modelos SPX, Post, Delta e Blunt, bem como, os resultados do uso da memória relativa do *VF-Ray*, *ME-Ray*, *EME-Ray* e *Bunyk* quando comparados com o uso do *EVF-Ray*. Similarmente ao feito na seção 7.2, as porcentagens apresentadas nesta Tabela correspondem à razão do uso de memória dos outros algoritmos sobre o consumo de memória do *EVF-Ray*. Sendo assim, este equivale a 100%.

Conforme podemos ver na Tabela que indica o consumo de memória 7.4, o *EVF-Ray* obteve uma redução considerável no consumo de memória em relação a sua versão original, o *VF-Ray*, devida a reestruração que foi feita na estrutura de dados. O *VF-Ray* utiliza de 27% a 133% mais memória do que o *EVF-Ray*.

A Tabela 7.5 apresenta o tempo gasto pelo *EVF-Ray* para renderizar um frame

(em segundos) para os modelos SPX, Post, Delta e Blunt, bem como, os resultados dos tempos relativos do *VF-Ray*, *ME-Ray*, *EME-Ray* e *Bunyk* quando comparados com o tempo de execução do *EVF-Ray*. Similarmente à Tabela anterior, as percentagens apresentadas correspondem à razão do tempo de execução dos outros algoritmos sobre o tempo de execução do *EVF-Ray*.

Conforme podemos observar na Tabela 7.5, o *EVF-Ray* supera em desempenho o *VF-Ray*, *ME-Ray*, *EME-Ray* e *Bunyk* para todos os conjuntos de dados e para todas as resoluções de imagens. O desempenho do *VF-Ray* já era excelente conforme descrito na seção 7.2 e este só perdia para o *Bunyk*. Com a reestruturação do *VF-Ray*, obteve-se uma estrutura de dados mais *leve*. Associado a isso, a simplificação feita no tratamento dos casos degenerados bem como a forma de determinar a face de saída utilizando-se a projeção do vértice oposto sobre a face de entrada fez do *EVF-Ray* um algoritmo mais competitivo. E, conforme veremos na seção 8.4, o *EVF-Ray* faz um ótimo uso das hierarquias de memória.

Dataset	**Res.**	**Memory (MB)**	**VF-Ray**	**ME-Ray**	**EME-Ray**	**Bunyk**
SPX	512^2	42	229%	512%	176%	683%
	1024^2	44	225%	520%	198%	686%
	2048^2	53	204%	530%	260%	687%
	4096^2	89	162%	553%	392%	692%
POST	512^2	27	233%	359%	185%	678%
	1024^2	29	228%	379%	228%	686%
	2048^2	38	195%	542%	337%	687%
	4096^2	74	149%	622%	516%	699%
DELTA	512^2	52	223%	404%	173%	675%
	1024^2	54	220%	417%	189%	678%
	2048^2	64	202%	523%	252%	666%
	4096^2	100	165%	570%	396%	666%
BLUNT	512^2	11	127%	400%	173%	673%
	1024^2	13	231%	454%	189%	692%
	2048^2	22	177%	523%	252%	668%
	4096^2	58	129%	621%	396%	669%

Tabela 7.4: Consumo de memória do *EVF-Ray* versus *VF-Ray*, *ME-Ray*, *EME-Ray* e *Bunyk*.

Dataset	Res.	Time (sec)	VF-Ray	ME-Ray	EME-Ray	Bunyk
SPX	512^2	1,85	120,31%	148,19%	238,51%	107,68%
	1024^2	6,22	112,69%	125,26%	273,68%	105,93%
	2048^2	22,44	106,82%	124,29%	324,88%	106,14%
	4096^2	85,91	104,02%	125,76%	315,85%	104,82%
POST	512^2	1,85	136,41%	223,36%	449,97%	127,28%
	1024^2	7,28	135,72%	214,73%	442,92%	117,33%
	2048^2	28,95	135,48%	213,55%	442,68%	115,06%
	4096^2	116,30	134,75%	211,64%	441,51%	112,66%
DELTA	512^2	1,60	117,96%	170,13%	324,00%	121,41%
	1024^2	5,53	114,58%	162,37%	367,96%	121,10%
	2048^2	20,53	112,51%	158,92%	394,09%	118,96%
	4096^2	80,05	114,31%	156,21%	407,58%	114,83%
BLUNT	512^2	0,63	183,56%	416,53%	691,57%	143,09%
	1024^2	2,47	181,39%	293,64%	694,21%	142,32%
	2048^2	9,86	179,97%	291,51%	697,23%	141,79%
	4096^2	39,10	180,97%	293,59%	718,71%	141,69%

Tabela 7.5: Resultados de tempo para o *EVF-Ray* versus *VF-Ray*, *ME-Ray*, *EME-Ray* e *Bunyk*.

Capítulo 8

Análise de Cache para o Algoritmo de Ray-Casting

Este capítulo tem por objetivo estudar os efeitos da hierarquia de memória sobre o algoritmo de ray-casting para dados irregulares. A versão otimizada do *VF-Ray*, chamada *EVF-Ray*, será utilizada para este fim. Veremos como as penalidades de cache ocorrem.

A primeira parte desta tese foi a implementação do algoritmo de ray-casting em CPU com um novo enfoque, o *VF-Ray*. Este sofreu otimizações em sua estrutura de dados gerando o *EVF-Ray*. Agora neste capítulo, veremos a segunda parte desta tese.

Até o momento foram feitos poucos trabalhos relativos a este tema. Todos esses avaliam o ray-casting sobre dados regulares, conforme descrito no capítulo de trabalhos relacionados. O maior destaque é o estudo feito por Palmer *et al* [19, 62].

De acordo com Palmer, o maior custo no núcleo do ray-casting para dados regulares são as penalidades de cache miss devido às leituras dos voxels do dataset.

Os dados regulares têm um padrão, pois, são fatias de mesmas dimensões do volume. Já os dados irregulares, muitas vezes gerados por simulações, não apresentam um padrão que possa ser explorado. E na maioria das vezes não são convexos, e ainda podem possuir buracos em seu interior. Isto requer um tratamento especial para que o raio possa continuar a partir do próximo ponto de entrada no volume.

Nosso objetivo é identificar o comportamento do algoritmo de ray-casting para dados irregulares sob a ótica da hierarquia de memória. Assim, podemos propor novas melhorias a fim de se explorar cada nível da hierarquia de memória.

8.1 Ferramentas de Avaliação de Desempenho

Com o objetivo de se avaliar o algoritmo de ray-casting para dados irregulares, usamos as ferramentas Cachegrind e Callgrind do framework Valgrind [68], bem como, a ferramenta PAPI [69]. Inúmeros trabalhos já utilizaram as ferramentas do framework Valgrind [70–77], assim como, a ferramenta PAPI [78–81] para este fim.

Valgrind [68] é um framework de instrumentação para construção dinâmica de ferramentas de análise. Este provê ferramentas para debug e análise de desempenho, como Memcheck, Cachegrind, Callgrind, Hellgrind e Massif. Valgrind é uma ferramenta ganhadora de prêmios a saber:

- Maio de 2008: Valgrind ganhou *TrollTech's inaugural Qt Open Source Development Award* pela melhor ferramenta de desenvolvimento de código aberto (open source);
- Julho de 2006: Julian Seward ganhou um *Google-O'Reilly Open Source Award* como *Best Toolmaker* por seu trabalho no framework Valgrind;
- Janeiro de 2004: Valgrind ganhou a medalha de bronze por mérito do Open Source Award.

A ferramenta Cachegrind [68] produz resultados detalhados sobre cache miss e falhas na predição de cláusulas branch (if then else). As estatísticas são geradas para o programa inteiro, para cada função e para cada linha de código. As seguintes estatísticas são coletadas:

- instrução de cache L1 para leitura e escrita;
- cache de dados L1 para leitura (reads), leituras perdidas (read misses), escritas (writes) e escritas perdidas(writes misses);
- cache unificado L2 para leitura (reads), leituras perdidas (read misses), escritas (writes) e escritas perdidas(writes misses);
- cláusulas condicionais (conditional branches) e falha na predição destas (mispredicted conditional branches);
- branches indiretos e falha na predição indireta. Um branch indireto é um *jump* ou uma chamada para um destino que somente se sabe em tempo de execução.

Nas arquiteturas mais modernas um miss no nível L1 custará em torno de 10 ciclos, um miss no nível L2 pode custar 200 ciclos e uma falha na predição de um branch custa de 10 a 30 ciclos.

Já a ferramenta Callgrind mostra os relacionamentos de custo entre as chamadas das funções, com simulação de cache opcional. Esta constrói um grafo de chamadas para uma execução do programa. Os dados coletados podem ser:

- número de instruções executadas;
- o relacionamento destas instruções com as linhas do código;
- o relacionamento caller/callee (quem chama/quem é chamado) entre as funções e a quantidade de tais chamadas;
- opcionalmente também tem um simulador de cache, similar ao Cachegrind.

Os resultados produzidos pelo framework Valgrind podem ser vizualizados num editor de texto comum ou através de uma interface visual, KCachegrind, que faz parte do Ambiente de Desktop KDE.

A PAPI [69], The Performance API, como o nome já diz, é uma biblioteca para avaliação de desempenho. É preciso programar, utilizando-se de chamadas específicas nos trechos de programa os quais se deseja avaliar. Dispõe de uma infinidade de contadores, mas que precisam estar habilitados pelo fabricante do processador. O uso desta biblioteca foi feito para corroborar as estatísticas geradas pelas ferramentas Cachegrind e Callgrind.

Utilizando as ferramentas Cachegrind, Callgrind e PAPI, estudamos o comportamento da memória cache nos primeiros níveis em um único processador. Estamos interessados nos valores de cache miss tanto no nível 1 quanto no nível 2.

8.2 Metodologia para Avaliação das Hierarquias de Memória

Todos os testes foram executados em um único processador. Não fizemos uso de threads a fim de garantir que somente um core (núcleo) fosse utilizado. O dataset utilizado, bem como as estruturas de dados cabem na memória principal, descartando assim, a ocorrência de swaps em disco.

Os testes foram feitos para um pequeno subconjunto de faces visíveis e para o conjunto como um todo. Esse subconjunto de faces foi escolhido de tal forma que utilizasse pouquíssima memória. Foram escolhidas 32 faces, tal que, um raio lançado por um de seus pixels corte o volume no máximo 200 vezes (valor ótimo conseguido via testes). Como cada face tem 32 bytes, então o tamanho da lista de faces já computadas será no máximo 6400 bytes ou 6.25 KB. Assim, esta lista ocupará no máximo 20% dos 32KB do cache L1. Como o cache L2 possui 4096 KB a questão de espaço não será um problema para esta lista que tem um tamanho variável.

Com a finalidade de determinar quanto do custo total de interseção é devido exclusivamente aos efeitos da hierarquia de memória, foi feito um algoritmo de *Teste*, onde o núcleo do algoritmo de ray-casting do *EFV-Ray*, sofreu uma modificação. Esta consistiu em fixar o tetraedro de entrada, assim, ao invés do raio corrente seguir seu caminho, este volta para o mesmo tetraedro, quantas vezes forem o número de faces cortadas por esse raio até sair do volume ou a opacidade máxima ser atingida. Com isto o cache miss gerado pela leitura dos vértices somente ocorre uma vez. A imagem final não é gerada corretamente, mas o objetivo é a eliminação dos custos associados às penalidades de cache miss. Como é preciso armazenar para cada pixel, o número de faces intersectadas numa lista, isto gera um pequeno overhead que não atrapalha a avaliação. núcleo da renderização

Foi medido o tempo e o cache/branch miss para cada uma das faces do conjunto de teste e para o volume como um todo. Teremos, então, três resultados com o uso do algoritmo normal – sem alterações – e o algoritmo com a modificação descrita acima. Resumindo temos:

1. os resultados do algoritmo otimizado, que é o *EVF-Ray*;
2. os resultados do algoritmo modificado, que chamaremos de *Teste*;
3. os resultados da diferença do *EVF-Ray* com o *Teste*, ou seja,*Diferença*.

O algoritmo *EVF-Ray* sofre os efeitos da hierarquia de memória e overheads. O algoritmo *Teste* já tem esses efeitos minimizados ao máximo, conforme descrito anteriormente. Assim, o que for computado para ele não pode ser atribuído aos efeitos da hierarquia de memória. E, finalmente, a diferença entre os algoritmos *EVF-Ray* e *Teste* é atribuída diretamente aos efeitos da hierarquia de memória.

O algoritmo *EVF-Ray* foi dividido em várias funções a fim de se obter uma melhor visão de cada parte, bem como, do algoritmo como um todo. Isto facilitará a avaliação do mesmo. O algoritmo ficou assim dividido em funções:

1. vfRay, núcleo da renderização (VF);
2. lê os identificadores do tetraedro corrente e da face visível, bem como, seus parâmetros de interpolação (TFIZS);
3. determina a Bounding Box ou caixa limitante da face (BBox);
4. projeta os vértices da face visível (PVF);
5. computa o produto cruzado da face visível (CVFC);
6. computa o produto cruzado do pixel e em conjunto com a função CFVC, determina se o pixel está dentro ou fora da face (CPxInF);

7. computa os valores de z e s (interpolado) para o pixel corrente que pertence à face (CZS);
8. faz a composição (RGB);
9. determina por qual face o raio sairá (CmpNxFc);
10. lê as faces já computadas (RdFc);
11. determina qual face conecta os tetraedros (FCBT);
12. computa a face interna ainda não computada, a insere na lista de faces computadas *computedFaces* e escreve no vetor *compFaceTet*, a posição desta na lista (CmpFc);
13. limpa o buffer de faces computadas (ClnBf);
14. No interior da função VF tem o ponto no qual se verifica se a face já foi computada ou não (CmT).

8.3 Ambiente de Testes

O algoritmo *EVF-Ray*, uma otimização do *VF-Ray*, também foi escrito em C++ ANSI sem usar qualquer biblioteca gráfica particular. Nossos experimentos foram conduzidos em um computador com processador Intel Core 2 Duo, de 2.0 GHz, com 4 GB de RAM, 32 KB de cache L1 e 4096 KB de cache L2, rodando sobre o Linux Ubuntu 10.04.

A cache tem as seguintes características:

- I1 cache: 32768 B, 64 B, 8-way associative;
- D1 cache: 32768 B, 64 B, 8-way associative;
- L2 cache: 4194304 B, 64 B, 16-way associative;

A análise é feita sobre o dado SPX, que é totalmente irregular, com buracos. Os eventos gravados e suas abreviações podem ser vistos na Tabela 8.1.

Eventos	Descrição
Ir	I cache reads (ie. instructions executed)
I1mr	I1 cache read misses
I2mr	L2 cache instruction read misses
Dr	D cache reads (ie. memory reads)
D1mr	D1 cache read misses
D2mr	L2 cache data read misses
Dw	D cache writes (ie. memory writes)
D1mw	D1 cache write misses
D2mw	L2 cache data write misses
Bc	Conditional branches executed
Bcm	Conditional branches mispredicted
Bi	Indirect branches executed
Bim	Conditional branches mispredicted

Tabela 8.1: Eventos de Cache gravados utilizando as ferramentas Cachegrind e Callgrind.

8.4 Resultados da Influência das Hierarquias de Memória

Nesta seção veremos os resultados obtidos sob duas perspectivas:

1. resultados exibidos tendo como base a quantidade de miss relativa ao núcleo da renderização, VF;

2. resultados exibidos tendo como base a quantidade de miss relativa ao total de leituras de memória de VF e da função Main. Estes valores de miss são absolutos.

Os resultados são exibidos sob estas duas formas a fim de vermos em relação à VF quanto representa cada parte que o compõe, bem como, em relação ao todo. As ferramentas Cachegrind e Callgrind apresentam os resultados desta mesma forma. Resumindo, se a quantidade de misses de uma função para $D1mr$ for K, tal que, K é o maior valor para $D1mr$ entre todas as funções, este passa a representar 100%. Todos os valores porcentuais das demais funções para $D1mr$ serão relativas a K. Este valor é relativo, pois o valor absoluto leva em conta o total de leituras feitas. No Gráfico 8.1 temos o valor $D1mr$ que representa a quantidade de miss de leitura e seu percentual em relação a VF. Isso quer dizer que o valor $D1mr$ de VF equivale a 100%. *VF incl* representa o custo de VF e de todas as chamadas internas. Já $VFSelf$ exibe o custo somente de VF, logo não leva em consideração o custo das chamadas internas. As demais funções apresentam o seu custo total em relação a VF. Como podemos observar, o maior custo está relacionado em computar por qual face o raio irá sair, ou seja, a próxima face. Isto ocorre, pois é preciso ler os vértices do tetraedro corrente, na medida em que o raio avança cortanto as faces internas.

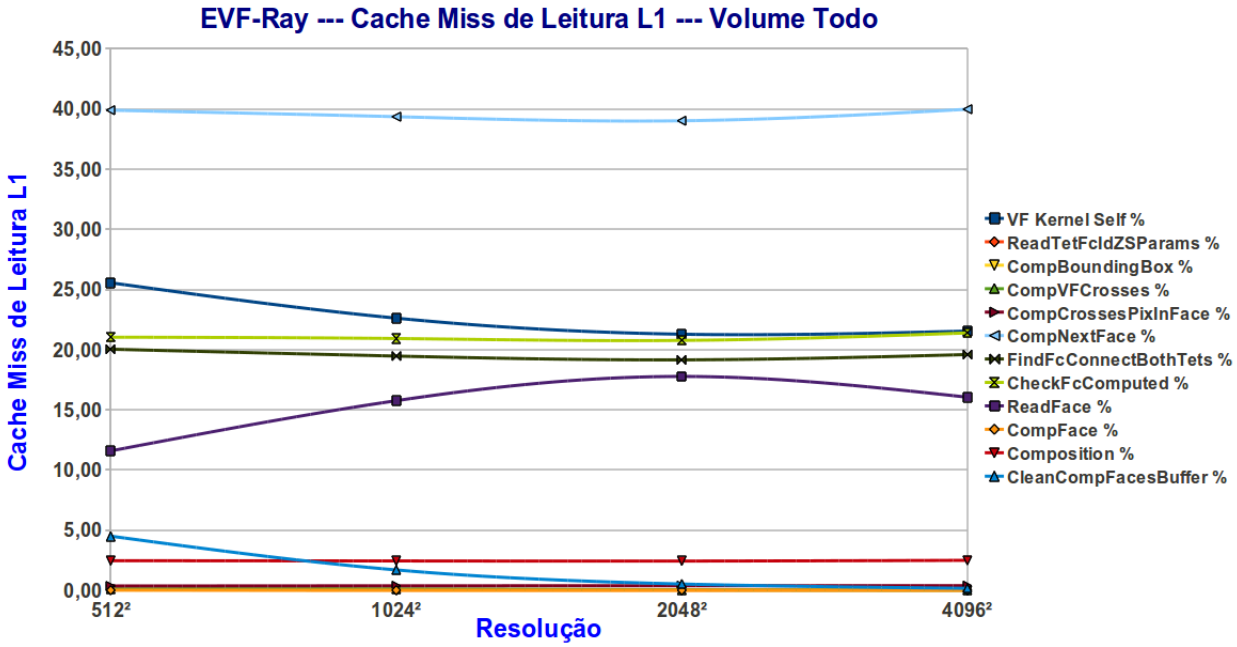

Figura 8.1: *EVF-Ray*: Cache Miss de Dados L1 (D1mr) para o dataset SPX2 tomando como base o valor de VF, inclusive (equivale a 100%).

Observe que na função *CompVFCrosses* – *CVFC* os vértices também são lidos, porém é somente do tetraedro de entrada. Para ler os vértices é preciso primeiro ler a face da lista de tetraedros *tetList* e depois ler os vértices da lista de vértices *vertList*. Este custo é elevado, pois estas listas são muito grandes, de fato, são as que ocupam maior espaço de memória. Porém, se observarmos a Tabela 8.2, constataremos que este valor não é mais do que 0.82% do valor $D1mr$ tomando como referência o valor Dr de VF. O mesmo acontece ao observarmos os valores de $D2mr$ na Tabela 8.3. Estes valores são devidos, primeiramente, ao fato de que os tetraedros vizinhos compartilham 3 vértices, justamente a face que os conecta. E por último, os raios lançados de pixels vizinhos cortam quase as mesmas faces, o que implica em um padrão, tornando o comportamento do algoritmo previsível. Desta forma tem-se uma alta taxa de acertos. O segundo e terceiro valores de peso estão associados à *CheckFcComputed* — CmT e à *FindFcConnectBothTets* — $FCBT$. A CmT é a clásula que verifica se uma face já foi ou não computada. Para tal, é preciso ler a lista *compFaceTet*. E para saber qual a face que conecta ambos os tetraedros é preciso ler a lista de conectividade *conTet*. Ambas as listas são muito grandes e nenhuma delas cabe no cache $L2$, tampouco, no $L1$. Mas temos o mesmo comportamento descrito acima. Como os raios lançados de pixels vizinhos tendem a reutilizar os mesmos dados, estes estão no cache $L1$ ou no $L2$. Pouquíssimas vezes o dado não se encontra na cache $L2$, tendo de acessar a memória principal. Este comportamento se repete para todas as funções do núcleo de VF. A Tabela 8.2 com o custo de $D2mr$ apresenta resultados similares.

Res.	VF Dr(100%)	VF incl	%	VF Self	%	TFIZ	%	CVFC	%	CmpNxFc	%	FCBT	%	CmpFc
512^2	2423022128	44524367	1,84	11377033	0,47	16882	0	47070	0	17763412	0,73	8918558	0,37	854
1024^2	8777004231	177066613	2,02	40038447	0,46	20025	0	54985	0	69688615	0,79	34459748	0,39	1870
2048^2	33749582047	711205599	2,11	151325892	0,45	21920	0	59689	0	277502378	0,82	136166198	0,40	1846
4096^2	133303748664	2710187720	2,03	583679379	0,44	23043	0	62929	0	1083436695	0,81	530874080	0,40	1852
Res.	**Main Dr(100%)**	**VF incl**	**%**	**VF Self**	**%**	**TFIZ**	**%**	**CVFC**	**%**	**CmpNxFc**	**%**	**FCBT**	**%**	**CmpFc**
512^2	4603457949	44524367	0,97	11377033	0,25	16882	0	47070	0	17763412	0,39	8918558	0,19	854
1024^2	10978611677	177066613	1,61	40038447	0,36	20025	0	54985	0	69688615	0,63	34459748	0,31	1870
2048^2	35980840018	711205599	1,98	151325892	0,42	21920	0	59689	0	277502378	0,77	136166198	0,38	1846
4096^2	135558069062	2710187720	2,00	583679379	0,43	23043	0	62929	0	1083436695	0,80	530874080	0,39	1852
Res.	**VF Dr(100%)**	**RdFc**	**%**	**CPxInF**	**%**	**BBox**	**%**	**CmT**	**%**	**RGB**	**%**	**ClnBf**	**%**	
512^2	2423022128	5154821	0,21	154640	0,01	11171	0	9362839	0,39	1091097	0,05	1998995	0,08	
1024^2	8777004231	27858191	0,32	643305	0,01	13089	0	37041515	0,42	4301427	0,05	2978587	0,03	
2048^2	33749582047	126303734	0,37	2620365	0,01	14340	0	147659452	0,44	17203577	0,05	3646077	0,01	
4096^2	133303748664	434709696	0,33	10220843	0,01	15065	0	579552245	0,43	67179203	0,05	4105849	0,00	
Res.	**Main Dr(100%)**	**RdFc**	**%**	**CPxInF**	**%**	**BBox**	**%**	**CmT**	**%**	**RGB**	**%**	**ClnBf**	**%**	
512^2	4603457949	5154821	0,11	154640	0,00	11171	0	9362839	0,20	1091097	0,02	1998995	0,04	
1024^2	10978611677	27858191	0,25	643305	0,01	13089	0	37041515	0,34	4301427	0,04	2978587	0,03	
2048^2	35980840018	126303734	0,35	2620365	0,01	14340	0	147659452	0,41	17203577	0,05	3646077	0,01	
4096^2	135558069062	434709696	0,32	10220843	0,01	15065	0	579552245	0,43	67179203	0,05	4105849	0,00	

Tabela 8.2: *EVF-Ray*: Valor Real de Cache Miss de Leitura L1 (D1mr) para o dataset SPX2 tomando como base o valor Dr de VF e $Main$ (equivale a 100%).

Res.	VF Dr(100%)	VF incl	%	VF Self	%	TFIZ	%	CVFC	%	CmpNxFc	%
512^2	2423022128	805221	0,03	275068	0,01	17235	0,00	47070	0,00	292520	0,01
1024^2	8777004231	833625	0,01	284665	0,00	20291	0,00	54985	0,00	302086	0,00
2048^2	33749582047	860285	0,00	293426	0,00	22324	0,00	59689	0,00	311906	0,00
4096^2	133303748664	906032	0,00	308061	0,00	23696	0,00	62929	0,00	329067	0,00
Res.	**Main Dr(100%)**	**VF incl**	**%**	**VF Self**	**%**	**TFIZ**	**%**	**CVFC**	**%**	**CmpNxFc**	**%**
512^2	4603457949	805221	0,02	275068	0,01	17235	0,00	47070	0,00	292520	0,01
1024^2	10978611677	833625	0,01	284665	0,00	20291	0,00	54985	0,00	302086	0,00
2048^2	35980840018	860285	0,00	293426	0,00	22324	0,00	59689	0,00	311906	0,00
4096^2	135558069062	906032	0,00	308061	0,00	23696	0,00	62929	0,00	329067	0,00
Res.	**VF Dr(100%)**	**FCBT**	**%**	**CmpFc**	**%**	**BBox**	**%**	**CmT**	**%**	**RGB**	**%**
512^2	2423022128	217310	0,01	22	0,00	2986	0,00	272081	0,01	137	0,00
1024^2	8777004231	223446	0,00	33	0,00	3605	0,00	281059	0,00	140	0,00
2048^2	33749582047	229583	0,00	29	0,00	4083	0,00	289342	0,00	142	0,00
4096^2	133303748664	241535	0,00	22	0,00	4336	0,00	303724	0,00	180	0,00
Res.	**Main Dr(100%)**	**FCBT**	**%**	**CmpFc**	**%**	**BBox**	**%**	**CmT**	**%**	**RGB**	**%**
512^2	4603457949	217310	0,00	22	0,00	2986	0,00	272081	0,01	137	0,00
1024^2	10978611677	223446	0,00	33	0,00	3605	0,00	281059	0,00	140	0,00
2048^2	35980840018	229583	0,00	29	0,00	4083	0,00	289342	0,00	142	0,00
4096^2	135558069062	241535	0,00	22	0,00	4336	0,00	303724	0,00	180	0,00

Tabela 8.3: *EVF-Ray*: Valor Real de Cache Miss de Leitura de Dados L2 (D2mr) para o dataset SPX2 tomando como base o valor Dr de VF e $Main$ (equivale a 100%).

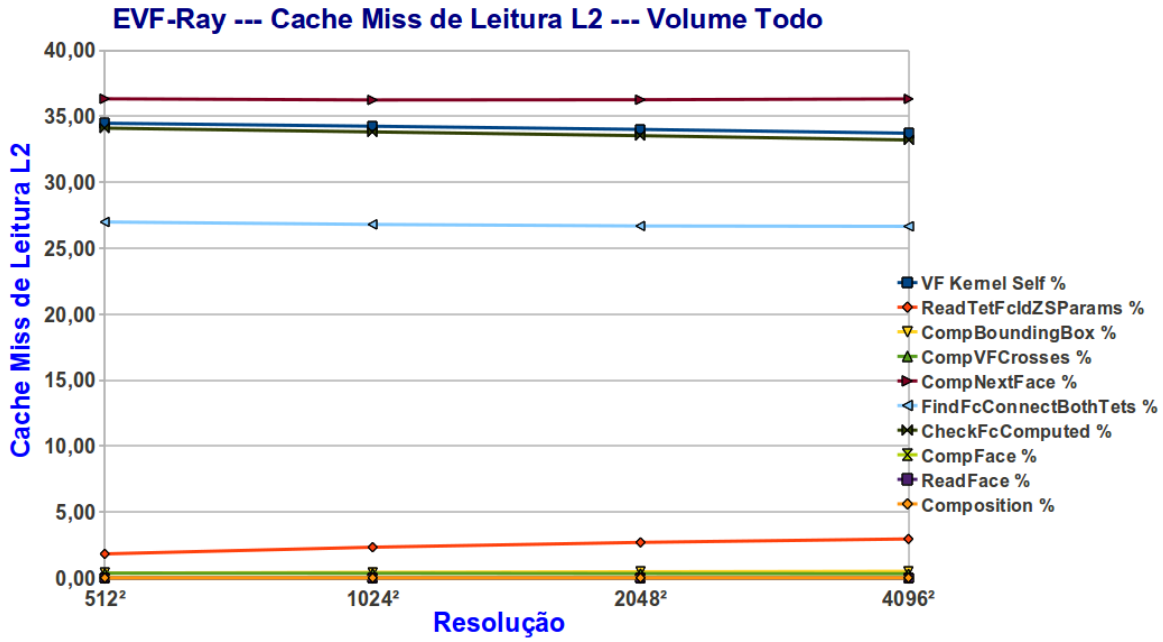

Figura 8.2: *EVF-Ray*: Cache Miss de Leitura de Dados L2 (D2mr) para o dataset SPX2 tomando como base o valor de VF, inclusive (equivale a 100%).

Nos Gráficos 8.3, 8.4, 8.5 e 8.7 podemos ver o comportamento do algoritmo *EVF-Ray*, que tem todos os cutos a ele associados, do algoritmo *Teste*, que não sofre influência das hierarquias de memória, e do resultado da diferença entre eles, *Diferença*, cujo custos estão associados exclusivamente aos efeitos da hierarquia de memória.

Conforme podemos observar, a diferença $EVF\text{-}Ray - Teste$ está muito próxima dos resultados do algoritmo *EVF-Ray* e muito longe dos de *Teste*. Nos Gráficos 8.3 e 8.4 temos os resultados levando em conta todo o volume, e nos Gráficos 8.5 e 8.7 os resultados relativos a 32 faces. O mesmo comportamento se observa em ambos:

- considerando todo volume, $EVF\text{-}Ray - Teste$ fica em torno de 99.97% para $D1mr$ e 94.83% para $D2mr$;
- para o conjunto de 32 faces, a média de $EVF\text{-}Ray - Teste$ é 99.44% para $D1mr$ e para $D2mr$ é 98.54%;

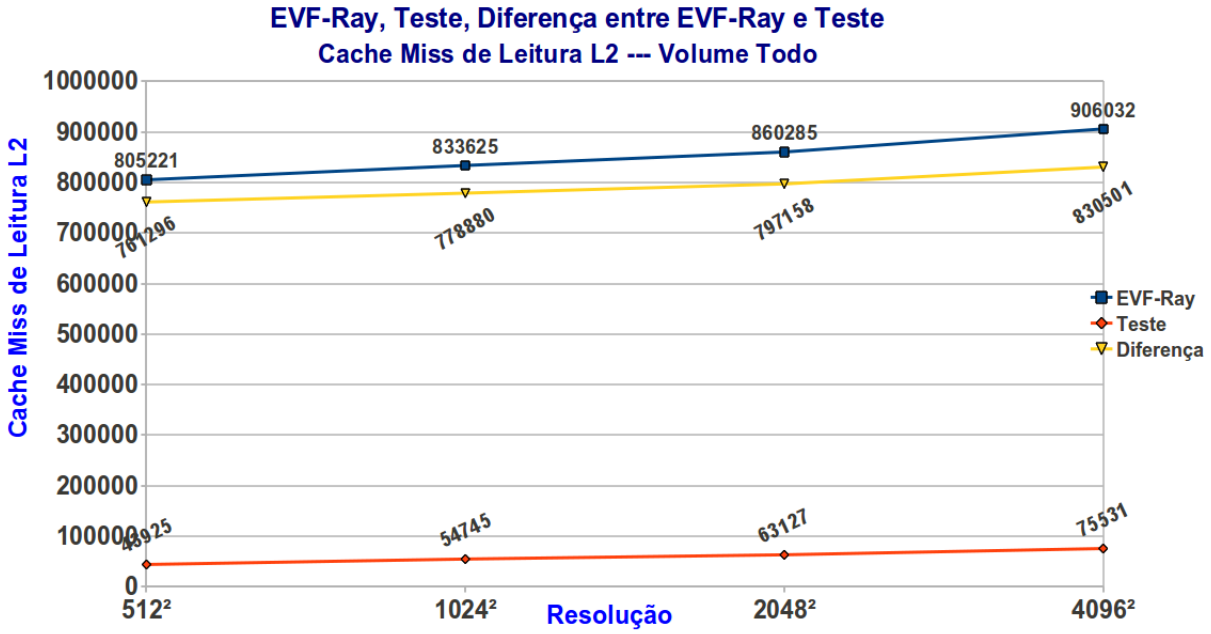

Figura 8.3: Cache Miss de Leitura de Dados L2 (D2mr) do núcleo da função de renderização (VF) exibindo o resultado *EVF-Ray*, *Teste* e a diferença entre eles, para todas as faces do volume.

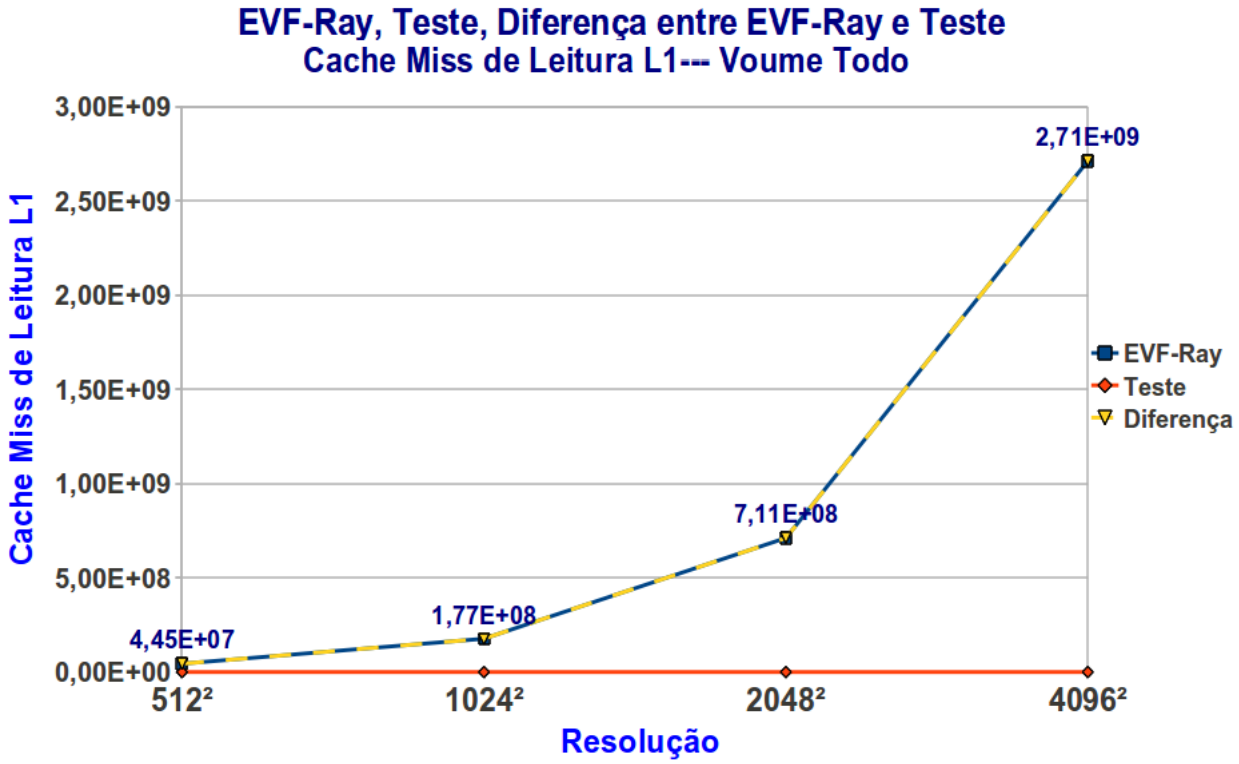

Figura 8.4: Cache Miss de Leitura L1 (D1mr) do núcleo da função de renderização (VF) exibindo o resultado de *EVF-Ray*, *Teste* e a diferença entre eles, para todas as faces do volume.

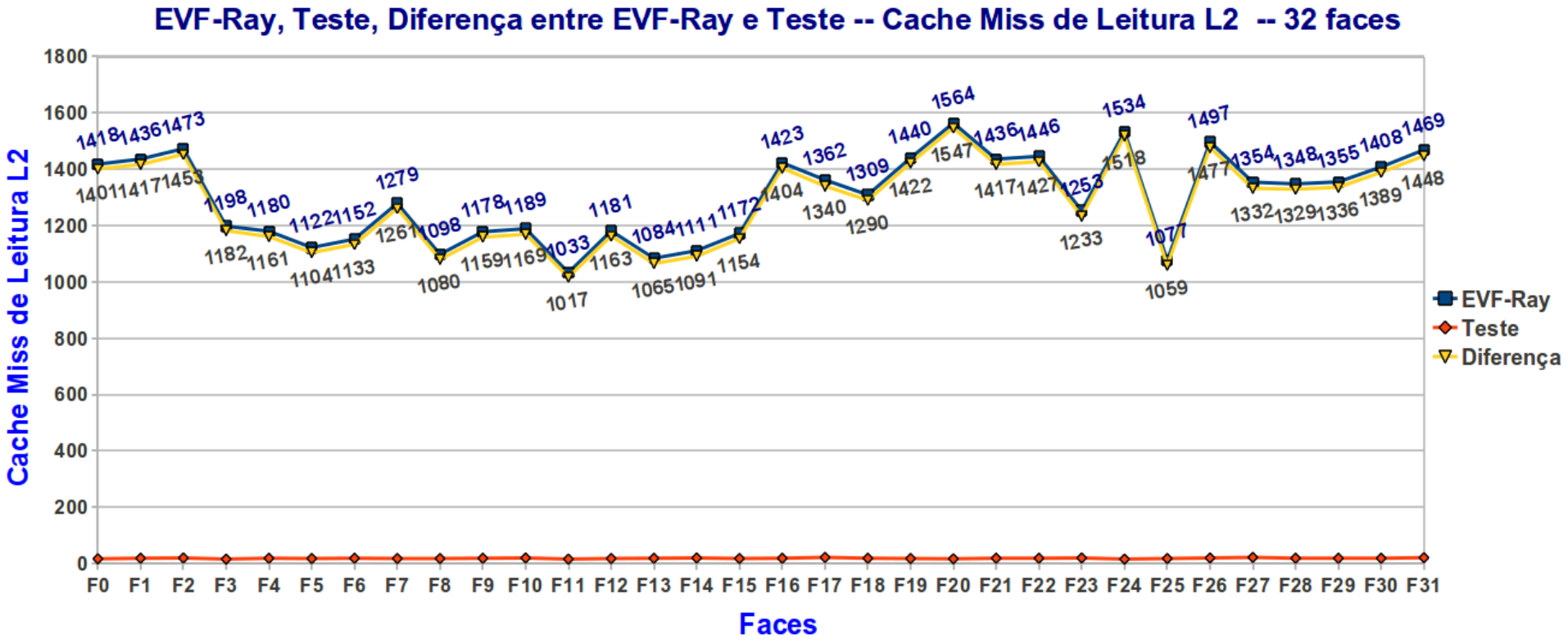

Figura 8.5: Cache Miss de Leitura de Dados L2 (D2mr) do núcleo da função de renderização (VF) exibindo o resultado de *EVF-Ray*, *Teste* e a diferença entre eles, para 32 faces.

A diferença *EVF-Ray* − *Teste* que representa os efeitos exclusivos da hierarquia de memória, representa a maior parte do custo total. Uma vez que para os valores locais e globais tem-se o mesmo comportamento, podemos concluir que o algoritmo de ray-casting *EVF-Ray* para dados irregulares faz um uso eficiente da cache. Além disso, devido aos baixíssimos valores absolutos de miss podemos ver que este está bem otimizado.

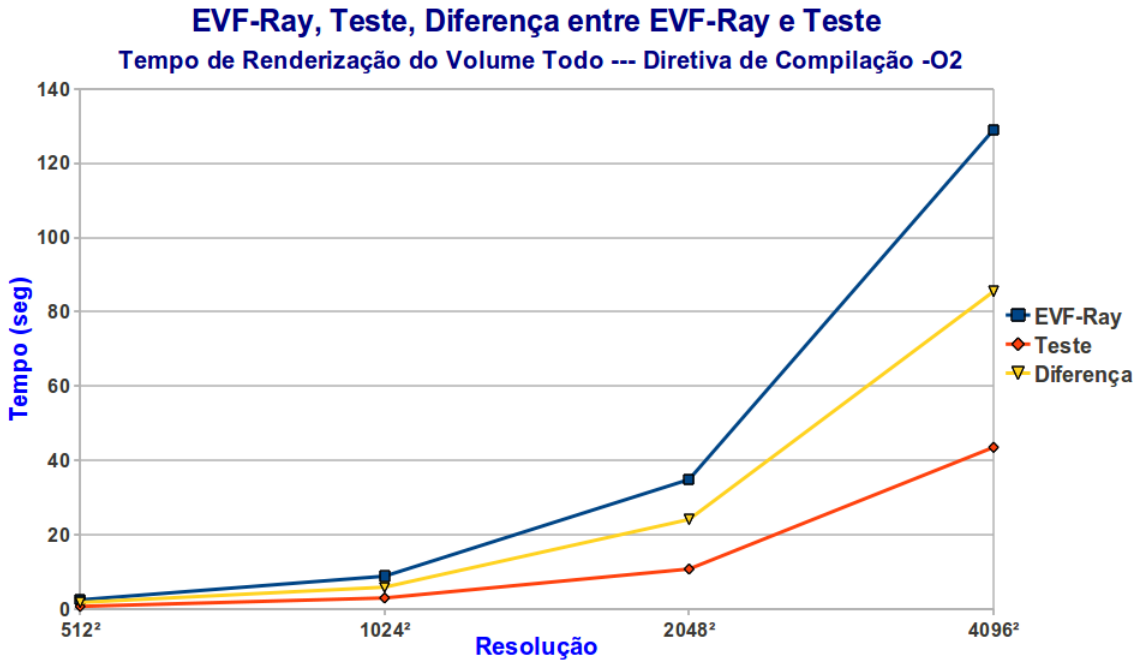

Figura 8.6: Tempo gasto pela função de renderização (VF) exibindo o resultado de *EVF-Ray*, *Teste* e a diferença entre eles, para todas as faces do volume.

O Gráfico 8.6 mostra o tempo gasto para renderizar o modelo SPX2, onde na compilação se utilizou a diretiva $-O2$, a mesma utilizada na medição de cache. O valor porcentual do tempo da diferença *EVF-Ray* − *Teste* equivale em média à $68,43\%$ do tempo do *EVF-Ray*. Isto mostra que a maior parte do tempo é devida exclusivamente aos efeitos da hierarquia de memória, corroborando os resultados mostrados neste capítulo.

A metodologia seguida foi feita para se ter uma visão do comportamento do algoritmo *EVF-Ray* de forma global e local. Ficou evidenciado como a localidade temporal de ray-casting influencia o comportamento e o desempenho deste. Do ponto de vista global a localidade temporal contribui para uma alta taxa de hit comprovando o bom desempenho. Do ponto de vista local, descobrimos que o maior custo está associado aos efeitos exclusivos da hierarquia de memória (*EVF-Ray* − *Teste*), evidenciando a importância do uso eficiente da cache.

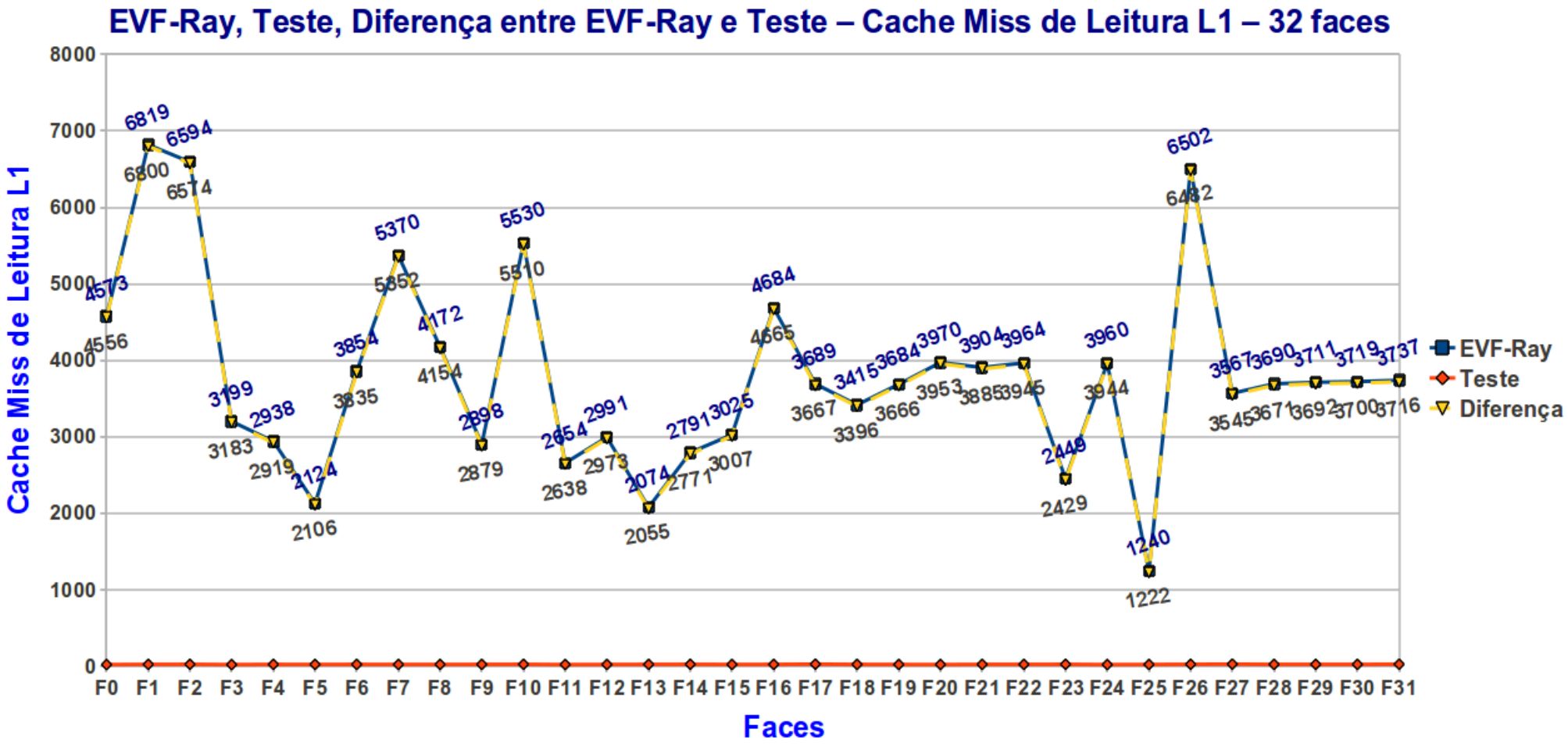

Figura 8.7: Cache Miss de Leitura L1 do núcleo da função de renderização (VF) exibindo o resultado de *EVF-Ray*, *Teste* e a diferença entre eles, para 32 faces.

Capítulo 9

Considerações Finais

Como em todo trabalho de vizualização volumétrica o que procuramos é reduzir o tempo de renderização com um gasto mínimo de memória, ou seja, um algoritmo eficiente e de baixo custo. O algoritmo *VF-Ray* e sua otimização, o *EVF-Ray*, preenchem esses requisitos.

Para tal, foi necessário estudar em detalhe o uso de cache para o *VF-Ray* e ter uma comparação de desempenho com outros modelos. Este estudo foi importante, pois o objetivo era aumentar a localidade temporal a fim de se obter uma taxa elevada de cache hit e baixa de cache miss. Este objetivo foi conseguido conforme podemos ver. A localidade temporal é aproveitada em *VF-Ray* pelo fato de que raios de uma mesma face visível provavelmente atravessarão quase todas as mesmas faces internas.

Em *VF-Ray*, entretanto, existem faces internas que são calculadas mais de uma vez, pois raios de pixels, de faces visíveis diferentes, interceptaram uma mesma face. Isto gera redundância. Como as faces visíveis são ordenadas por profundidade e depois são projetadas e renderizadas, nem sempre se renderizam em ordem de vizinhança. Portanto, para aumentar a localidade temporal do algoritmo, seria interessante que a próxima face visível escolhida fosse determinada a partir da lista de faces vizinhas da face corrente. Foram feitos vários experimentos neste sentido, mas nenhum apresentou um desempenho satisfatório.

O algoritmo *VF-Ray* e sua otimização *EVF-Ray* apresentaram excelentes resultados em relação ao uso da memória cache. Os maiores custos estão associados à leitura dos vértices, a verificar se a face já foi ou não computada, a determinar por qual face o raio sai da célula corrente e a determinar qual face conecta dois tetraedros. Vimos também que a metodologia empregada validou a grande influência da hierarquia de memória sobre o algoritmo de ray-casting para dados irregulares, evidenciando sua alta localidade temporal, uma vez que o custo da diferença entre os algoritmos *EVF-Ray* e *Teste* tem a maior proporção no custo total.

9.1 Direções Futuras

Como trabalho futuro, fica a implementação de uma versão paralela (em CPU) para ser executada em um *cluster* de PCs. Pois em GPU e Cell, já foram feitas e com sucesso. Um *cluster* de PCs é uma arquitetura atraente para computação paralela devido ao seu baixo custo e à sua alta disponibilidade. Esta arquitetura, entretanto, não possui memória compartilhada e apresenta um alto custo de comunicação. Portanto, a forma como a tarefa de renderização é distribuída pelos processadores do *cluster* tem grande influência no desempenho da renderização paralela. O problema de distribuição de carga na renderização paralela não é trivial de ser resolvido. Pode ser tratado com algoritmos de previsão e distribuição equilibrada ou com mecanismos de balanceamento dinâmico de carga. No caso do algoritmo *VF-Ray*, é possível utilizar a própria divisão em faces visíveis ou em grupos de faces visíveis como fator de divisão do trabalho, quando a paralelização é realizada no espaço da imagem. Para divisão no espaço da imagem, pode-se também utilizar a divisão tradicional da imagem em tiles. Com a implementação dos dois tipos de divisão de trabalho pode-se comparar a aceleração obtida em uma arquitetura paralela. Além disso, pode-se também inserir mecanismos de balanceamento de carga para evitar que processadores fiquem ociosos enquanto outros estão sobrecarregados.

Esperamos ter contribuído no desenvolvimento de um algoritmo de ray-casting, que é leve devido ao seu baixo consumo de memória, mas mantendo um bom desempenho. E com o estudo de cache para dados irregulares, podemos ver que a localidade temporal é fundamental para se ter um uso mais eficiente das hierarquias de memória.

Referências Bibliográficas

[1] MCCORMICK, B. H. "Visualization in scientific computing". N. 1, pp. 15–21, New York, NY, USA, 1987. ACM. doi: http://doi.acm.org/10.1145/43965.43966.

[2] HADWIGER, M., KNISS, J. M., ENGEL, K., et al. "High-Quality Volume Graphics on Consumer PC Hardware". In: *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, New York, NY, USA, 2002. ACM. ISBN: 1-58113-521-1.

[3] FOLEY, VAN DAM, FEINER, et al. *Computer Graphics Principles and Practice*. Second ed. , Addison-Wesley, 1996. ISBN: 0-201-84840-6.

[4] LICHTENBELT, B., CRANE, R., NAQVI, S. *Introduction to Volume Rendering*. First ed. , Hewlett-Packard, 1998. ISBN: 0-13-861683-3.

[5] CHALLINGER, J. "Scalable parallel volume raycasting for nonrectilinear computational grids". In: *In ACM SIGGRAPH Symposium on Parallel Rendering*, 1993.

[6] FARIAS, R., MITCHELL, J. S. B., 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*, pp. 91–99, New York, NY, USA, 2000. ACM Press. ISBN: 1-58113-308-1. doi: http://doi.acm.org/10.1145/353888.353905.

[7] HOFSETZ, C., MA, K.-L. "Multi-threaded rendering unstructured-grid volume data on the sgi origin 2000". In: *VIS '00: Proceedings of the conference on Visualization '00*, 2000.

[8] MA., K.-L. "Parallel volume ray-casting for unstructured-grid data on distributed-memory architectures", *IEEE Parallel Rendering Symposium*, pp. 23–30, 1995.

[9] MA, K.-L., CROCKETT, T. “A scalable parallel cell-projection volume rendering algorithm for three-dimensional unstructured data”, *IEEE Parallel Rendering Symposium*, pp. 95–104, 1997.

[10] ENGEL, K., HADWIGER, M., SALAMA, C. R. “Tutorial 7: Real Time Volume Graphics. Eurographics 2006.” , 2006. http://www.sbc.org.br/sbac/2006/index.htm.

[11] HONG, L., KAUFMAN, A. “Accelerated ray-casting for curvilinear volumes”. In: *VIS ’98: Proceedings of the conference on Visualization ’98*, pp. 247–253, 1998.

[12] MEIBNER, M., HUANG, J., BARTZ, D., et al. “A practical evaluation of popular volume rendering algorithms”. In: *VVS ’00: Proceedings of the 2000 IEEE symposium on Volume visualization*, pp. 81–90, 2000.

[13] KOYAMADA, K. “Fast Ray-Casting for Irregular Volumes”. In: *ISHPC ’00: Proc. of the Third International Symposium on High Performance Computing*, pp. 557–572, 2000.

[14] NEUBAUER, A., MROZ, L., HAUSER, H., et al. “Cell-based first-hit ray casting”. In: *VISSYM ’02: Proceedings of the symposium on Data Visualisation 2002*, pp. 77–84, 2002.

[15] GARRITY, M. P. “Raytracing irregular volume data”. In: *VVS ’90: Proceedings of the 1990 workshop on Volume visualization*, pp. 35–40, New York, NY, USA, 1990. ACM Press. ISBN: 0-89791-417-1. doi: http://doi.acm.org/10.1145/99307.99316.

[16] BUNYK, P., KAUFMAN, A. E., SILVA, C. T. “Simple, Fast, and Robust Ray Casting of Irregular Grids”. In: *Dagstuhl ’97, Scientific Visualization*, pp. 30–36, Washington, DC, USA, 1997. IEEE Computer Society. ISBN: 0-7695-0505-8.

[17] PINA, A., BENTES, C., FARIAS, R. “Memory Efficient and Robust Sofware Implementation of the Ray-casting algorithm”, *The 15-th International Conference in Central Europe on Computer Graphics Visualization and Computer Vision*, 2007.

[18] SAULO RIBEIRO, ANDRÉ MAXIMO, CRISTIANA BENTES, et al. “Memory-Aware and Efficient Ray-Casting Algorithm”, *Computer Graphics and Image Processing, Brazilian Symposium on*, v. 0, pp. 147–154, 2007. ISSN: 1530-1834. doi: http://doi.ieeecomputersociety.org/10.1109/SIBGRAPI.2007.28.

[19] PALMER, M. E., TOTTY, B., TAYLOR, S. “Ray Casting on Shared-Memory Architectures: Memory Hierarchy Considerations in Volume Rendering”. In: *IEEE Concurrency*, v. 6, pp. 6(1):20–35, Oklahoma, 1998. IEEE Press.

[20] KAHLE, J. A., DAY, M. N., HOFSTEE, H. P., et al. “Introduction to the Cell multiprocessor”, *IBM Journal of Research and Development*, v. 49, n. 4/5, 2005.

[21] MAXIMO, A., RIBEIRO, S., BENTES, C., et al. “Memory Efficient GPU-Based Ray Casting for Unstructured Volume Rendering”. pp. 155–162, Los Angeles, California, USA, 2008. Eurographics Association. ISBN: 978-3-905674-12-5. doi: http://doi.ieeecomputersociety.org/10.2312/VG/VG-PBG08/155-162.

[22] COX, G., MÁXIMO, A., BENTES, C., et al. “Irregular Grid Raycasting Implementation on the Cell Broadband Engine”, *Computer Architecture and High Performance Computing, Symposium on*, v. 0, pp. 93–100, 2009. ISSN: 1550-6533. doi: http://doi.ieeecomputersociety.org/10.1109/SBAC-PAD.2009.15.

[23] RASMUSSEN, N., NGUYEN, D. Q., GEIGER, W., et al. “Smoke Simulation For Large Scale Phenomena”. v. 22, pp. 703–707, 2003.

[24] GUEDES, A. R. C. “Campos vetoriais e escalares”. . http://socrates.if.usp.br/ everton/download/html/vol02/node4.html/.

[25] PINA, A. *Implementação Robusta e Eficiente em Uso de Memória do Algoritmo de Raycast*. Tese de Mestrado, Universidade Federal do Rio de Janeiro, 2005.

[26] ADAMS, P. “Paraview Data Formats”. . https://visualization.hpc.mil/wiki/.

[27] LE, S., MYSID. “Unstructured grid”. . http://en.wikipedia.org/wiki/.

[28] UPSON, C., KEELER, M. “V-buffer: visible volume rendering”. In: *SIGGRAPH ’88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pp. 59–64, New York, NY, USA, 1988. ACM Press. ISBN: 0-89791-275-6. doi: http://doi.acm.org/10.1145/54852.378482.

[29] SABELLA, P. “A rendering algorithm for visualizing 3D scalar fields”. In: *SIGGRAPH ’88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pp. 51–58, New York, NY, USA,

1988. ACM Press. ISBN: 0-89791-275-6. doi: http://doi.acm.org/10.1145/54852.378476.

[30] LEVOY, M. "Display of surfaces from volume data". In: *IEEE Computer Graphics and Applications*, pp. 29–37. IEEE Press, 1988.

[31] WESTOVER, L. "Footprint evaluation for volume rendering". In: *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pp. 367–376, New York, NY, USA, 1990. ACM Press. ISBN: 0-201-50933-4. doi: http://doi.acm.org/10.1145/97879.97919.

[32] LACROUTE, P., LEVOY, M. "Fast volume rendering using a shear-warp factorization of the viewing transformation". In: *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pp. 451–458, New York, NY, USA, 1994. ACM. ISBN: 0-89791-667-0. doi: http://doi.acm.org/10.1145/192161.192283.

[33] KAUFMAN, A. E. "Volume visualization", *ACM Comput. Surv.*, p. 479, 1991.

[34] KIM, C. E. "Three-dimensional digital planes", *Pattern Recogn. Lett.*, v. 1, pp. 639–645, 1984. ISSN: 0167-8655.

[35] MAX, N., HANRAHAN, P., CRAWFIS, R. "Area and volume coherence for efficient visualization of 3D scalar functions". In: *VVS '90: Proceedings of the 1990 workshop on Volume visualization*, pp. 27–33, New York, NY, USA, 1990. ACM Press. ISBN: 0-89791-417-1. doi: http://doi.acm.org/10.1145/99307.99315.

[36] MAX, N. L. "Vectorized procedural models for natural terrain: Waves and islands in the sunset". In: *SIGGRAPH '81: Proceedings of the 8th annual conference on Computer graphics and interactive techniques*, pp. 317–324, New York, NY, USA, 1981. ACM Press. ISBN: 0-89791-045-1. doi: http://doi.acm.org/10.1145/800224.806820.

[37] 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*, pp. 21–29, New York, NY, USA, 1982. ACM Press. ISBN: 0-89791-076-1. doi: http://doi.acm.org/10.1145/800064.801255.

[38] MAX, N. "Optical Models for Direct Volume Rendering", *IEEE Transactions on Visualization and Computer Graphics*, v. 1, n. 2, pp. 99–108, 1995. ISSN: 1077-2626. doi: http://dx.doi.org/10.1109/2945.468400.

[39] SHIRLEY, P., TUCHMAN, A. A. "Polygonal Approximation to Direct Scalar Volume Rendering". In: *Proceedings San Diego Workshop on Volume Visualization, Computer Graphics*, v. 24(5), pp. 63–70, 1990. Disponível em: ¡citeseer.ist.psu.edu/shirley90polygonal.html¿.

[40] ROETTGER, S., KRAUS, M., ERTL, T. "Hardware-accelerated volume and isosurface rendering based on cell-projection". In: *VIS '00: Proceedings of the conference on Visualization '00*, pp. 109–116, Los Alamitos, CA, USA, 2000. IEEE Computer Society Press. ISBN: 1-58113-309-X.

[41] WYLIE, B., MORELAND, K., FISK, L. A., et al. "Tetrahedral projection using vertex shaders". In: *VVS '02: Proceedings of the 2002 IEEE Symposium on Volume visualization and graphics*, pp. 7–12, Piscataway, NJ, USA, 2002. IEEE Press. ISBN: 0-7803-7641-2.

[42] WESTOVER, L. "Interactive volume rendering". In: *VVS '89: Proceedings of the 1989 Chapel Hill workshop on Volume visualization*, pp. 9–16, New York, NY, USA, 1989. ACM. doi: http://doi.acm.org/10.1145/329129.329138.

[43] MUELLER, K., SHAREEF, N., HUANG, J., et al. "High-Quality Splatting on Rectilinear Grids with Efficient Culling of Occluded Voxels", *IEEE Transactions on Visualization and Computer Graphics*, v. 5, n. 2, pp. 116–134, 1999. ISSN: 1077-2626. doi: http://dx.doi.org/10.1109/2945.773804.

[44] MUELLER, K., CRAWFIS, R. "High-Quality Splatting on Rectilinear Grids with Efficient Culling of Occluded Voxels", *IEEE Transactions on Visualization and Computer Graphics*, pp. 239–245, 1998.

[45] LACROUTE, P. "Real-time volume rendering on shared memory multiprocessors using the shear-warp factorization". In: *PRS '95: Proceedings of the IEEE symposium on Parallel rendering*, pp. 15–22, New York, NY, USA, 1995. ACM. ISBN: 0-89791-774-1. doi: http://doi.acm.org/10.1145/218327.218331.

[46] HENNESSY, J. L., PATTERSON, D. A. *Computer Architecture: A Quantitative Approach*. USA, Morgan Kaufmann Publications, 2007. ISBN: 0123704900.

[47] DENNING, P. J. "The working set model for program behavior", *Commun. ACM*, v. 11, n. 5, pp. 323–333, 1968. ISSN: 0001-0782. doi: http://doi.acm.org/10.1145/363095.363141.

[48] STALLINGS, W. *Computer Organization and Architecture: Designing for Performance*. Upper Saddle River, NJ, USA, Prentice Hall PTR, 2005. ISBN: 0131856448.

[49] DREPPER, U. "What Every Programmer Should Know About Memory", *White Paper RedHat, Inc*, 2007.

[50] WEILER, M., KRAUS, M., MERZ, M., et al. "Hardware-Based Ray Casting for Tetrahedral Meshes". In: *VIS '03: Proceedings of the 14th IEEE conference on Visualization '03*, pp. 333–340, 2003. ISBN: 0-7695-2030-8/03.

[51] BERNARDON, F., PAGOT, C., COMBA, J., et al. "GPU-based Tiled Ray Casting using Depth Peeling", *Journal of Graphics Tools*, 2007.

[52] ESPINHA, R., 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*, p. 273. IEEE Computer Society, 2005. ISBN: 0-7695-2389-7. doi: http://dx.doi.org/10.1109/SIBGRAPI.2005.29.

[53] MARROQUIM, R., MAXIMO, A., FARIAS, R., et al. "GPU-Based Cell Projection for Interactive Volume Rendering". In: *SIBGRAPI '06: Proceedings of the XIX Brazilian Symposium on Computer Graphics and Image Processing*, pp. 147–154, Los Alamitos, CA, USA, 2006. IEEE Computer Society. doi: http://doi.ieeecomputersociety.org/10.1109/SIBGRAPI.2006.22.

[54] CHALLINGER, J. *Parallel Volume Rendering on a Shared-Memory Multiprocessor*. Relatório técnico, Santa Cruz, CA, USA, 1991.

[55] NIEH, J., LEVOY, M. "Volume rendering on scalable shared-memory MIMD architectures". In: *VVS '92: Proceedings of the 1992 workshop on Volume visualization*, pp. 17–24, New York, NY, USA, 1992. ACM. ISBN: 0-89791-527-5. doi: http://doi.acm.org/10.1145/147130.147141.

[56] SINGH, J. P., GUPTA, A., LEVOY, M. "Parallel Visualization Algorithms: Performance and Architectural Implications", *Computer*, v. 27, n. 7, pp. 45–55, 1994. ISSN: 0018-9162. doi: http://dx.doi.org/10.1109/2.299410.

[57] CORRIE, B., MACKERRAS, P. "Parallel volume rendering and data coherence". In: *PRS '93: Proceedings of the 1993 symposium on Parallel rendering*, pp. 23–26, New York, NY, USA, 1993. ACM. ISBN: 0-89791-618-2. doi: http://doi.acm.org/10.1145/166181.166184.

[58] MACKERRAS, P., CORRIE, B. "Exploiting Data Coherence to Improve Parallel Volume Rendering", *IEEE Concurrency*, v. 2, n. 2, pp. 8–16, 1994. ISSN: 1063-6552. doi: http://doi.ieeecomputersociety.org/10.1109/88.311568.

[59] LAW, A., YAGEL, R. "The Active-Ray Approach to Rendering on Distributed Memory Multiprocessors". In: *SPDP '96: Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing (SPDP '96)*, p. 414, Washington, DC, USA, 1996. IEEE Computer Society. ISBN: 0-8186-7683-3.

[60] LAW, A., YAGEL, R. "Multi-Frame Thrashless Ray Casting with Advancing Ray-Front". In: *In Proc. of Graphics Interfaces*, pp. 70–77, 1996.

[61] LAW, A., YAGEL, R. "An Optimal Ray Traversal Scheme for Visualizing Colossal Medical Volumes". In: *Proceedings of Visualization in Biomedical Computing, VBC '96*, pp. 33–43, 1996.

[62] PALMER, M. E. "Tese de Doutorado Exploiting Parallel Memory Hierarchies for RayCasting Volumes". In: *IEEE Concurrency*, v. 6, pp. 6(1):20–35, Oklahoma, 1997. IEEE Press.

[63] GRIMM, S., BRUCKNER, S., KANITSAR, A., et al. "A refined data addressing and processing scheme to accelerate volume raycasting", *COMPUTERS & GRAPHICS*, 2004.

[64] GRIMM, S., BRUCKNER, S. "Memory Efficient Acceleration Structures and Techniques for CPU-based Volume Raycasting of Large Data". In: *In Proceedings of the IEEE/SIGGRAPH Symposium on Volume Visualization and Graphics 2004 (2004*, pp. 1–8, 2004.

[65] BRUCKNER, S. *Efficient Volume Visualization of Large Medical Datasets*. Tese de Mestrado, Institute of Computer Graphics and Algorithms, Vienna University of Technology, May 2004.

[66] DREBIN, R. A., CARPENTER, L., HANRAHAN, P. "Volume rendering". In: *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pp. 65–74, New York, NY, USA, 1988. ACM Press. ISBN: 0-89791-275-6. doi: http://doi.acm.org/10.1145/54852.378484.

[67] NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. Relatório técnico, NVIDIA Corporation, January 2007.

[68] ARMOUR-BROWN, C., FITZHARDINGE, J., HUGHES, T., et al. "VALGRIND An instrumentation framework for building dynamic analysis tools." , 2000.

[69] DONGARRA, J., MOORE, S., RALPH, J., et al. "PAPI The Performance API". , 2000.

[70] BOND, M. D., NETHERCOTE, N., KENT, S. W., et al. "Tracking bad apples: reporting the origin of null and undefined value errors", *SIGPLAN Not.*, v. 42, n. 10, pp. 405–422, 2007. ISSN: 0362-1340. doi: http://doi.acm.org/10.1145/1297105.1297057.

[71] NETHERCOTE, N., SEWARD, J. "Valgrind: a framework for heavyweight dynamic binary instrumentation", *SIGPLAN Not.*, v. 42, n. 6, pp. 89–100, 2007. ISSN: 0362-1340. doi: http://doi.acm.org/10.1145/1273442.1250746.

[72] NETHERCOTE, N., SEWARD, J. "How to shadow every byte of memory used by a program". In: *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, pp. 65–74, New York, NY, USA, 2007. ACM. ISBN: 978-1-59593-630-1. doi: http://doi.acm.org/10.1145/1254810.1254820.

[73] DREWRY, W., ORMANDY, T. "Flayer: exposing application internals". In: *WOOT '07: Proceedings of the first USENIX workshop on Offensive Technologies*, pp. 1–9, Berkeley, CA, USA, 2007. USENIX Association.

[74] WEIDENDORFER, J., KOWARSCHIK, M., TRINITIS, C. "A Tool Suite for Simulation Based Analysis of Memory Access Behavior". In: *In Proceedings of International Conference on Computational Science*, pp. 440–447. Springer, 2004.

[75] BROWN, A. W., KELLY, P. H. J., LUK, W. "Profile-directed speculative optimization of reconfigurable floating point data paths". In: *Proceedings of the 1st HiPEAC Workshop on Reconfigurable Computing*, 2007.

[76] MUEHLENFELD, A., WOTAWA, F. "Fault detection in multi-threaded c++ server applications". In: *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 142–143, New York, NY, USA, 2007. ACM. ISBN: 978-1-59593-602-8. doi: http://doi.acm.org/10.1145/1229428.1229457.

[77] NEWSOME, J. “Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software”. In: *NDSS 2005: In Proceedings of the Network and Distributed System Security Symposium*, 2005.

[78] TERPSTRA, D., JAGODE, H., YOU, H., et al. “Collecting Performance Data with PAPI-C”. In: *Proceedings of the 3rd Parallel Tools Workshop*, 2010.

[79] MOORE, S., CRONK, D., WOLF, F., et al. “Performance Profiling and Analysis of DoD Applications using PAPI and TAU”. In: *Proceedings of DoD HPCMP UGC 2005*. IEEE, 2005.

[80] ANDERSSON, U., MUCCI, P. “Analysis and Optimization of Yee-Bench using Hardware Performance Counters”. In: *ParCo: Proceedings of Parallel Computing*, 2005.

[81] MUCCI, P., AHLIN, D., DANIELSSON, J., et al. “PerfMiner: Cluster-Wide Collection, Storage and Presentation of Application Level Hardware Performance Data”. In: *Euro-Par: Proceedings of 2005 European Conference on Parallel Computers*, 2005.