COPPE/UFRJ

ALGORITMOS APRIMORADOS PARA VISUALIZAÇÃO VOLUMÉTRICA E PROCESSAMENTO DE MALHAS

André de Almeida Maximo

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: Ricardo Cordeiro de Farias
Amitabh Varshney

Rio de Janeiro
Julho de 2010

ALGORITMOS APRIMORADOS PARA VISUALIZAÇÃO VOLUMÉTRICA E PROCESSAMENTO DE MALHAS

André de Almeida Maximo

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. Ricardo Cordeiro de Farias, Ph.D.

____
Prof. Amitabh Varshney, Ph.D.

____
Diego Fernandes Nehab, Ph.D.

____
Prof. João Luiz Dihl Comba, Ph.D.

____
Prof. Ricardo Guerra Marroquim, D.Sc.

____
Prof. Cristiana Bentes, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
JULHO DE 2010

Maximo, André de Almeida

Algoritmos Aprimorados para Visualização Volumétrica e Processamento de Malhas/André de Almeida Maximo. – Rio de Janeiro: UFRJ/COPPE, 2010.

XV, 111 p.: il.; 29, 7cm.

Orientadores: Ricardo Cordeiro de Farias
Amitabh Varshney

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

Referências Bibliográficas: p. 81 – 90.

1. Computação Gráfica. 2. Visualização Volumétrica. 3. Processamento de Malhas. I. Farias, Ricardo Cordeiro de *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Aos meus queridos pais e
irmãos, pelo apoio e amor

Agradecimentos

Gostaria de agradecer a todos que contribuíram para a conclusão desta tese.

Aos professores do Laboratório de Computação Gráfica (LCG): Ricardo Farias, Ricardo Marroquim, Claudio Esperança, Paulo Roma e Antônio Oliveira. Pelas dúvidas sanadas e apoio sempre presente. Em especial aos Ricardos (Farias e Marroquim) pela amizade e conversas nas importantes decisões de doutorando.

Ao Professor Amitabh Varshney do *Graphics and Visual Informatics Laboratory* (GVIL), por ter me recebido bem como *intern* na *University of Maryland* (UMD) em 2009, no meu 1 ano de doutorado sanduíche. E pela sua paciência e sabedoria nas diversas reuniões onde conversamos de pesquisa e filosofia.

A todos os meus amigos do LCG que acompanharam junto comigo as idas e vindas do doutorado: Álvaro “Bubu”, “Dino” Saulo, Ricardo “Rico”, Yalmar, Wagner, Fláv-IO, Felipe “Cabeludo”; Leandro “Brucutu” e tantos outros; pelas conversas, discussões e conselhos necessários academicamente e pessoalmente.

A todos os meus amigos do GVIL que acompanharam a luta no decorrer do meu doutorado sanduíche: Robert Patro “Rob”, Sujal Bista e Cheuk Yiu Ip “Horace”. Pela ajuda e conversas nas incontáveis horas de trabalho no laboratório.

Agradeço aos meus amigos(as) de infância, de muitos anos e os conhecidos recentemente: André, Anderson, Nelson, Jô, Eneida, Danilo, Maise, Melissa, Bruna, Vanessa, Léo Claudino, Aninha, João, William, Léo Mineiro, Mandy e tantos outros. Pela força, amizade e incentivo no decorrer dos últimos anos.

Agradeço aos responsáveis pelos momentos de lazer: Thirsty Turtle, U Street, Lapa, Irish Pub, Cinemark, Wizards of the Coast, Joanne K. Rowling, Dan Brown e Bernard Cornwell. Momentos esses importantes para a continuidade do trabalho.

Aos meus familiares: Vó Nazita, Michel, Fabiano, Andréia, Cristiano, Tia Terezinha, Tia Celinha, Tio Tercílio, Tio Tuninho, e tantos outros; por toda a ajuda e importante presença na minha vida.

Agradeço também aos meus irmãos: Mário e Bárbara; e meus pais: Paulo e Magda; pelo apoio, presença e confiança essenciais para a conclusão desta tese.

Agradeço ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) pelo suporte financeiro durante o doutorado pleno e sanduíche; e ao Vicente Batista pela da classe CoppeTeX do LaTeX usada na escrita desta tese.

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

ALGORITMOS APRIMORADOS PARA VISUALIZAÇÃO VOLUMÉTRICA E PROCESSAMENTO DE MALHAS

André de Almeida Maximo

Julho/2010

Orientadores: Ricardo Cordeiro de Farias
Amitabh Varshney

Programa: Engenharia de Sistemas e Computação

Nesta tese são apresentados algoritmos aprimorados para visualização volumétrica e processamento de malhas. Na área de pesquisa de visualização volumétrica, o objetivo é a melhoria de desempenho computacional e consumo de memória, usando placas gráficas programáveis, enquanto na área de processamento de malhas o objetivo é introduzir um método para ampliar o uso de similaridade de formas de uma superfície. Os algoritmos de visualização se baseiam em traçado de raios e projeção de células, tratando tanto dados volumétricos regulares quanto irregulares e renderizando os dados tanto diretamente quanto indiretamente através de isosuperfícies. O método de processamento de malhas, por outro lado, amplia o uso de auto-similaridade de modelos para propagar processamento feito em uma parte do modelo para outras partes similares. O método apresentado é usado em duas aplicações distintas: transferência de detalhe e parametrização. E, finalmente, as técnicas de visualização volumétrica são comparadas entre si e a técnicas correlatas do estado da arte.

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

IMPROVED ALGORITHMS FOR VOLUME RENDERING AND MESH PROCESSING

André de Almeida Maximo

July/2010

Advisors: Ricardo Cordeiro de Farias
Amitabh Varshney

Department: Systems Engineering and Computer Science

In this thesis, improved algorithms are presented for volume rendering and mesh processing. In the research area of volume rendering, the goal is to improve computational performance and memory consumption, using programmable graphics cards, while in the area of mesh processing, the goal is to introduce a method to augment the usage of shape similarity of a surface. The volume rendering algorithms are based in ray casting and cell projection, handling both regular and irregular data, and employing both direct and indirect volume rendering techniques. The mesh processing method, on the other hand, augments the usage of self-similarity of models to propagate processing done in one part of the mesh to many others similar parts. The presented method is exploited in two distinct applications: detail transfer and parameterization. Finally, the volume rendering techniques are compared against each other and correlated state-of-art techniques.

Sumário

Lista de Figuras **x**

Lista de Tabelas **xii**

Índice Remissivo **xiii**

1 Introdução **1**
1.1 Visualização Volumétrica 2
1.2 Processamento de Malhas 5
1.3 Programação em GPU 7

2 Revisão Bibliográfica **11**
2.1 Visualização Volumétrica 12
2.1.1 Integral de Iluminação 12
2.1.2 Ordenação por Visibilidade 15
2.1.3 Traçado de Raios 15
2.1.4 Projeção de Células 17
2.2 Processamento de Malhas 19
2.2.1 Simetria Refletivas 19
2.2.2 Descritores de Similaridade 20

3 Visualização Volumétrica **23**
3.1 VF-Ray-GPU 24
3.2 RPTINT 31
3.3 IPTINT 36
3.4 HAPT 41

4 Processamento de Malhas **48**
4.1 SAMPLE 49
4.2 Aplicações 57

5 Resultados **60**
5.1 Visualização Volumétrica 61
5.2 Processamento de Malhas 75

6 Conclusões **79**

Referências Bibliográficas **81**

A Algoritmos Desenvolvidos **91**

B Algoritmo VF-Ray **92**

C Algoritmo PT **95**

D Algoritmo PTINT **98**

Glossário **104**

Lista de Figuras

1.1 Exemplo de imagens médicas . . . 2
1.2 Exemplo de visualização volumétrica direta . . . 3
1.3 Exemplo de edição da função de transferência . . . 4
1.4 Exemplo de visualização volumétrica indireta . . . 5
1.5 Exemplo de processamento de malhas . . . 6
1.6 Pipeline gráfico . . . 8
1.7 Esquema de processamento usando CUDA . . . 9

2.1 Modelo da integral de iluminação . . . 12
2.2 Modelo simplificado da integral de iluminação . . . 13
2.3 Exemplo de traçado de raios . . . 16
2.4 Tipos de faces . . . 16
2.5 Exemplo de projeção de células . . . 17
2.6 Exemplo de plano de reflexão . . . 19
2.7 Exemplo de assinatura de vértice . . . 21

3.1 Coerência entre raios do VF-Ray . . . 24
3.2 Kernels do algoritmo VF-Ray-GPU . . . 26
3.3 Estruturas de dados do VF-Ray-GPU . . . 27
3.4 Tabela de dispersão do VF-Ray-GPU . . . 29
3.5 Esquema de threads do terceiro kernel do VF-Ray-GPU . . . 30
3.6 Visão geral do algoritmo RPTINT . . . 31
3.7 Pipeline do algoritmo RPTINT . . . 33
3.8 Estrutura de vetores do RPTINT . . . 34
3.9 Esquema de texturas no IPTINT . . . 36
3.10 Entrada/Saída do primeiro passo do IPTINT . . . 37
3.11 Estrutura de vetores do IPTINT . . . 38
3.12 Renderização de iso-superfície do IPTINT . . . 40
3.13 Framework do HAPT . . . 42
3.14 Pipeline do HAPT . . . 42
3.15 Exemplo de classe 1 de projeção do PT . . . 45

4.1 Exemplo de espaço de similaridade do SAMPLE 48
4.2 Exemplo de descritor de similaridade do SAMPLE 50
4.3 Expansão de Zernike para mapa de alturas no SAMPLE 52
4.4 Diferença (I) entre métodos de similaridade 53
4.5 Diferenca (II) entre métodos de similaridade 54
4.6 Ilustração do espaço primal e dual no SAMPLE 55
4.7 Aplicação do SAMPLE: parametrização de superfície 57
4.8 Aplicação do SAMPLE: transferência de detalhe 58

5.1 Imagens geradas pelo VF-Ray-GPU 64
5.2 Imagens geradas pelo RPTINT 65
5.3 Volume "spx+" gerado pelo IPTINT 66
5.4 Imagens geradas (I) pelo IPTINT 67
5.5 Imagens geradas (II) pelo IPTINT 68
5.6 Volume "torso" renderizado com iso-superfícies pelo HAPT 70
5.7 Volume "torso" renderizado sem iso-superfícies pelo HAPT 71
5.8 Renderização de Dado 4D 72
5.9 Imagens geradas pelo HAPT 73
5.10 Exemplo de espaço dual do SAMPLE 75
5.11 Exemplo de simetria no SAMPLE 76
5.12 Exemplo de aplicação do vizinho dual imediato 76
5.13 Exemplo de similaridades mais próximas 77

Lista de Tabelas

3.1 Erro entre ordenação por centróide e MPVONC 44

5.1 Propriedades dos volumes de teste do VF-Ray-GPU 62
5.2 Aspectos de memória dos algoritmos 63
5.3 Comparação de tempo e memória 63
5.4 Comparação do VF-Ray original e VF-Ray-GPU 63
5.5 Medida de desempenho do RPTINT 64
5.6 Tempo do terceiro e quarto passo do RPTINT 66
5.7 Comparação de diferentes algoritmos com o IPTINT 67
5.8 Medida de desempenho do IPTINT 68
5.9 Medida de desempenho do HAPT 69
5.10 Comparação de diferentes algoritmos (I) com o HAPT 70
5.11 Comparação de diferentes algoritmos (II) com o HAPT 72
5.12 Tempo gasto para estabelecer o espaço dual no SAMPLE 78

Índice Remissivo

[2D] Duas Dimensões, bidimensional, 2
[3D] Três Dimensões, tridimensional, 2
[4D] Quatro Dimensões, quadridimensional, 41
[CFD] Computational Fluid Dynamics, 2
[CGAL] Computational Geometry Algorithms Library, 60
[CPU] Central Processing Unit, 7
[CT] Computed Tomography, 2
[CUDA] Compute Unified Device Architecure, 9
[DVR] Direct Volume Rendering, 41
[E/S] Entrada/Saída, 7
[EuroVis] Eurographics/IEEE Symposium on Visualization, 41
[FPS] Frames Per Second, 3
[FS] Fragment Shader, 41
[GATOR] GPU-Accelerated Tetrahedra Renderer, 18
[GB] Gigabytes, 62
[GHz] Giga Hertz, 62
[GLSL] OpenGL Shading Language, 9
[GLUT] OpenGL Utility Toolkit, 60
[GNU GPLv3] GNU General Public License v.3, 2
[GPGPU] General Purpose Computation on Graphics Hardware, 25
[GPU] Graphics Processing Unit, 1
[GRAPP] International Conference on Computer Graphics Theory and Applications, 31
[GS] Geometry Shader, 41
[HAPT] Hardware-Assisted Projected Tetrahedra, 23
[HARC] Hardware-Assisted Ray Casting, 16
[HAVIS] Hardware-Accelerated Volume and Isosurface Rendering Based on Cell-Projection, 61
[HAVS] Hardware-Assisted Visibility Sorting, 15
[HKS] Heat Kernel Signature, 21
[I/O] Input/Output, 95
[IPTINT] Improved Projected Tetrahedra with Partial Pre-Integration, 23
[ISO] Iso-surface Rendering, 41
[KB] Kilobytes, 63
[K] Thousand, 62
[LCF] local coordinate frame, 56
[LCGtk] Toolkit do Laboratório de Computação Gráfica, 60
[MB] Megabytes, 62
[MRI] Magnetic Resonance Imaging, 2
[MRT] Multiple Render Targets, 37
[MVONC] Meshed Polyhedra Visibility Ordering for Non-Convex meshes, 42
[M] Million, 62
[OpenGL] Open Graphics Library, 9
[PBO] Pixel Buffer Object, 94
[PCA] Principal Component Analysis, 22
[PCI] Peripheral Component Interconnect, 64
[PC] Personal Computer, 66
[PRST] Planar-Reflective Symmetry

Transform, 22
[PTINT] Projected Tetrahedra with Partial Pre-Integration, 18
[PT] Projected Tetrahedra, 18
[Pre-Int] Pré-Integração, 62
[RAM] Random Access Memory, 62
[RBF] Radial Basis Function, 22
[RGBA] Red Green Blue Alpha, 13
[RGB] Red Green Blue, 33
[RPTINT] Regular Projected Tetrahedra with Partial Pre-Integration, 23
[SAMPLE] Similarity Augmented Mesh Processing using Local Exemplars, 48
[SIBGRAPI] Brazilian Symposium on Computer Graphics and Image Processing, 24
[SIMD] Single Instruction Multiple Data, 9
[STL] Standard Template Library, 42
[Tet] Tetrahedra, 62
[VBO] Vertex Buffer Object, 43
[VCGLib] Visual Computing Lab Library, 60
[VF-Ray-GPU] Visible-Face Driven Ray Casting implemented on the GPU, 23
[VF-Ray] Visible-Face Driven Ray Casting, 23
[VICP] View-Independent Cell Projection, 15
[VRAM] Video RAM, 69
[VS] Vertex Shader, 41
[Verts] Vértices, 62
[blunt] Blunt Fin dataset, 61
[comb] Combustion Chamber dataset, 61
[delta] Delta Wing dataset, 61
[f16] F-16 Jet Simulation, 61
[fighter] Langley Fighter dataset, 61
[fuel] Fuel Injection dataset, 61
[ms] milisegundos, 63
[neghip] Electron Distribution Probability dataset, 61
[post] Liquid Oxygen Post dataset, 61
[spx] Super Phoenix dataset, 61
[torso] Human Torso dataset, 61
[turbjet] time-varying Turbulent Jet dataset, 61

Capítulo 1

Introdução

"Think of giving not as a duty
but as a privilege."
– John Davison Rockefeller

Esta tese apresenta um conjunto de algoritmos aprimorados para *Visualização Volumétrica* e *Processamento de Malhas*, duas grandes áreas de *Computação Gráfica* [1]. As técnicas de visualização apresentadas visam a melhoria de desempenho computacional e consumo de memória, enquanto que o método de processamento introduz um novo conceito no uso de auto-similaridade de modelos. Em visualização, os algoritmos desenvolvidos objetivam a *renderização* de dados volumétricos, especificamente campos escalares regulares ou irregulares, em placa gráfica programável (GPU) [1]. Em processamento de malhas, o método desenvolvido usa vizinhança *não-local* definida por descritores de malhas para propagar processamento feito em uma parte do modelo para outras partes, que compartilhem alguma propriedade desejada. As técnicas apresentadas das duas áreas são distintas em essência (listadas no Apêndice A), não havendo correlação entre elas.

Neste capítulo é apresentado uma breve introdução dos assuntos relacionados com os trabalhos desta tese. No capítulo seguinte (2), revisões bibliográficas compreendendo ambas as áreas são discutidas. As contribuições desta tese na área de visualização volumétrica são apresentadas no Capítulo 3, e na área de processamento de malhas no Capítulo 4. Os resultados obtidos em ambas as áreas são apresentados no Capítulo 5, e conclusões dos trabalhos no Capítulo 6. Adicionalmente, os Apêndices e o Glossário provêem informações complementares para entendimento do material aqui apresentado.

[1] O leitor é convidado a referir ao Glossário na busca do significado de palavras-chave e siglas.

1.1 Visualização Volumétrica

A área de visualização volumétrica é responsável pela geração de imagens computacionais a partir de dados 3D ou volumétricos. As principais fontes de dados volumétricos são simulações numéricas de fenômenos naturais, e.g. *Computação Dinâmica de Fluídos* (CFD), e dispositivos de medição, e.g. *Ressonância Magnética* (MRI) e *Tomografia Computadorizada* (CT) na medicina e *Tomografia Sísmica* na geologia. Geralmente, dispositivos de medição produzem dados volumétricos regulares, enquanto simulações geram dados tanto regulares quanto irregulares.

Um exemplo de dado volumétrico regular em imagens médicas pode ser visto na Figura 1.1. Estas imagens são de um *CT scan* (ou *CAT scan*) de um crânio humano e formam o dado bruto que pode ser, e geralmente é, diretamente analisado por médicos. As imagens do *CT scan* compreendem uma sequência de imagens 2D, ou fatias, que unidas formam uma *imagem 3D* ou *volume*.

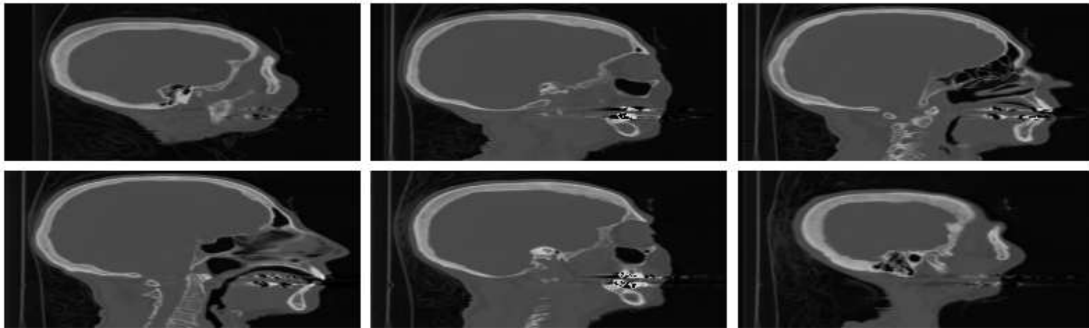


Figura 1.1: Imagens da tomografia computadorizada de um crânio humano com resolução de $256 \times 256 \times 113$, extraídas de um plano de corte em YZ.

O volume regular de exemplo, neste caso um crânio, é o objeto de interesse analisado de fatia em fatia pelas imagens mostradas na Figura 1.1. Estas imagens foram geradas a partir de um *dataset* real, por um aplicativo desenvolvido por mim disponível em:

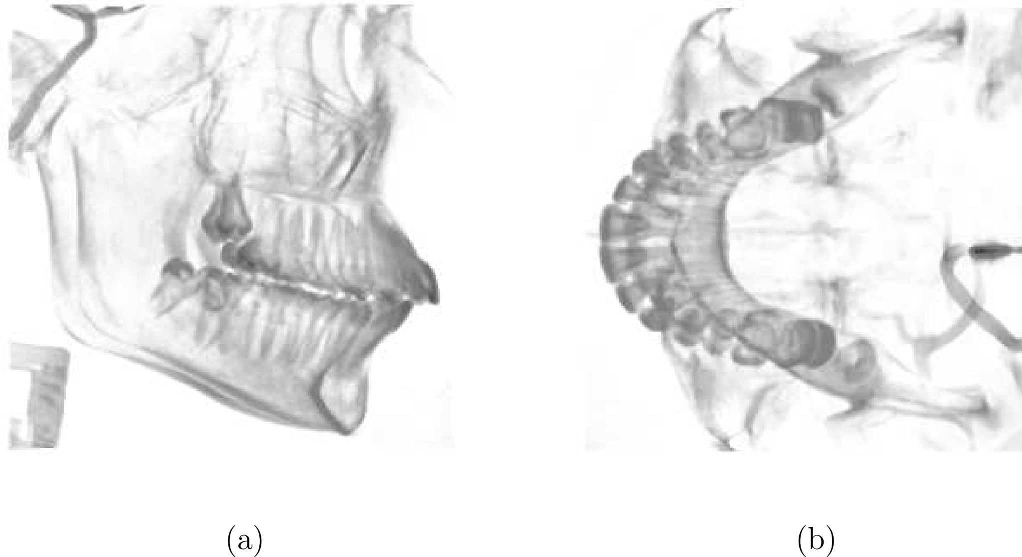
http://code.google.com/p/image3dviewer [1].

A principal desvantagem deste tipo de análise visual é a falta de uma componente tridimensional, importante para aproximar o dado volumétrico do objeto real analisado. A área de visualização volumétrica visa adicionar esta componente, gerando imagens do volume de um *ponto de vista* qualquer e compondo as diversas fatias para permitir a sensação visual de profundidade.

Um exemplo de visualização volumétrica, usando uma das técnicas apresentadas nesta tese (explicada na Seção 3.2), pode ser visto na Figura 1.2. O volume utilizado

[1] Códigos fontes de aplicativos relacionados a esta tese são abertos (*open source*) sob a licença *GNU General Public License* versão 3 (GNU GPLv3) no repositório do Google™ code.

para gerar estas imagens é o mesmo definido pelas fatias mostradas na Figura 1.1, porém compondo as fatias usando transparência. O dado regular neste exemplo é um campo escalar que representa valores de densidade do crânio humano, onde a escala cinza das imagens corresponde as amostras escalares do campo.



(a) (b)

Figura 1.2: Imagens geradas utilizando visualização volumétrica para dois pontos de vista diferentes: lateral (a); e superior (b).

Um dos desafios deste tipo de visualização é a interatividade, ou seja, permitir a manipulação eficiente do volume trocando o ponto de vista. Dependendo do tamanho do dado e da janela de visualização, o tempo de renderização pode ser muito longo (segundos ou até minutos) comprometendo a interação com o volume. Por exemplo, o aplicativo usado na Figura 1.2 permite uma interação em tempo real (usando máquinas contemporâneas) de 70 quadros por segundo (*fps*) para dados regulares com $64 \times 64 \times 64$ *voxels* e uma janela de tamanho 512×512 pixels. Porém para dados com 256^3 voxels ou mais, o desempenho cai para menos de 1 *fps*, comprometendo a resposta visual da interação. Este desafio de desempenho na manipulação do ponto de vista é importante na área de visualização volumétrica, onde os dados são geralmente grandes e requerem uma análise extensa.

Outra forma de manipulação de dados volumétricos se baseia no controle de transparência e realce do volume. Este controle é feito a partir da *função de transferência*, responsável por mapear valores escalares a cores e opacidade. Através desta função, por exemplo, partes do volume podem ser ocultadas permitindo visualizar regiões de interesse com mais detalhe. A Figura 1.3 exemplifica o uso da função de transferência para diminuir a opacidade de valores escalares pequenos, colorindo o volume do azul ao vermelho. O dado volumétrico usado neste exemplo é irregular e compreende uma simulação de fluídos. A janela da esquerda é a visualização volu-

métrica em si, usando um dos algoritmos apresentados neste trabalho de pesquisa (explicado na Seção 3.4), enquanto a da direita exibe uma interface para edição da função de transferência. Este padrão de janelas é usado por alguns dos aplicativos disponibilizados com esta tese.

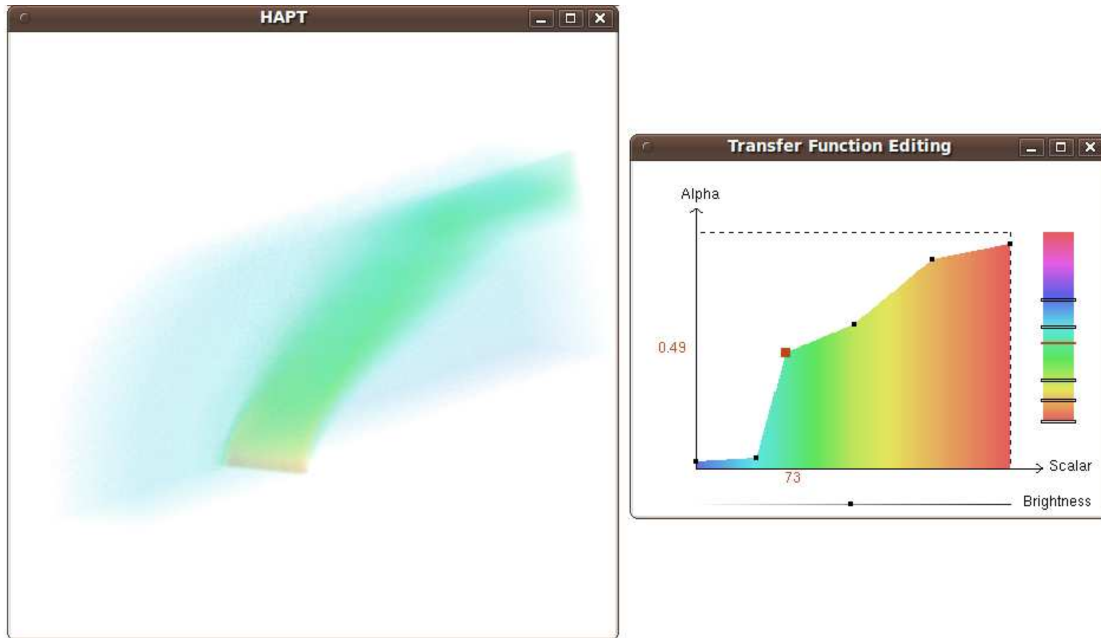


Figura 1.3: Exemplo de edição da função de transferência. A visualização do dado volumétrico (à esquerda) é modificada pela edição da função de transferência associada a ele (à direita).

Assim como o desempenho na manipulação do ponto de vista é importante na visualização volumétrica, a edição eficiente da função de transferência também é interessante. Este tipo de interação é alcançado se, ao passo que a função é modificada, o volume refletir as modificações ao mesmo tempo. A modificação da função de transferência é dada pela alteração de valores de cor e opacidade (o eixo *alpha* na Figura 1.3) associados aos valores escalares do volume. O alto desempenho nesta interação permite uma manipulação melhor do dado volumétrico, possibilitando que dados com grande variação de valores sejam analisados com maior eficácia.

Em contraste com a visualização volumétrica mostrada até agora, conhecida como *visualização volumétrica direta*, existe uma visualização alternativa de dados volumétricos chamada *visualização volumétrica indireta*. A visualização direta se baseia em renderizar o volume como um material semitransparente, enquanto que a visualização indireta busca encontrar e renderizar superfícies de mesmo valor dentro do volume, chamadas de *iso-superfícies*. A Figura 1.4 mostra um exemplo de renderização de iso-superfícies de um dado volumétrico representando probabilidade espacial de distribuição de elétrons. A técnica usada nesta renderização é parte desta tese (explicada em detalhes na Seção 3.3).

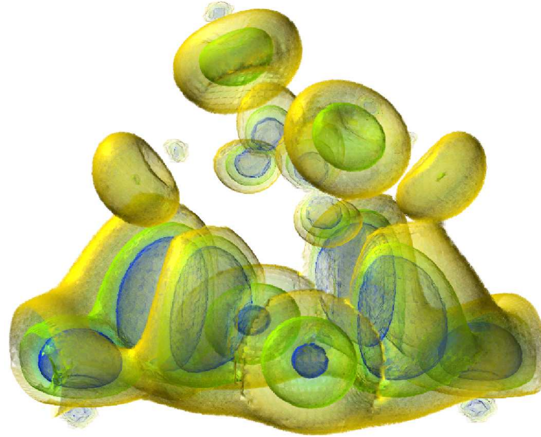


Figura 1.4: Exemplo de visualização volumétrica indireta. As superfícies de mesmo valor escalar, chamadas de iso-superfícies, são realçadas dentro do volume.

As contribuições de pesquisa desta tese na área de visualização volumétrica envolvem ambas as visualizações, direta e indireta, assim como tratam de ambos os dados, regulares e irregulares.

1.2 Processamento de Malhas

A área de processamento de malhas, chamada mais genericamente de *processamento de geometria*, abrange algoritmos relacionados com aquisição, reconstrução, análise, armazenamento, recuperação, manipulação, simulação e transmissão de objetos, ou modelos, tridimensionais. Os modelos 3D tratados nesta extensa área são, geralmente, malhas triangulares descrevendo a superfície de um objeto de interesse. Esta descrição é dada por um conjunto de vértices e faces que formam a superfície discreta do objeto. Exemplos de modelos 3D podem ser encontrados em jogos e filmes que usem computação gráfica, onde os objetos virtuais nas cenas são modelados por um artista ou *escaneados* de objetos reais usando um *scanner 3D*.

A Figura 1.5 mostra um exemplo de modelo 3D e de uma técnica matemática usada em processamento de malhas aplicada a parte do modelo de exemplo. O modelo mostrado é o *XYZ RGB Asian Dragon*, uma escultura de um dragão asiático escaneada pela corporação XYZ RGB. Este modelo é relativamente grande, contendo 108 mil vértices e 216 mil faces triangulares. A técnica mostrada "aplana" um pedaço da superfície ao redor do vértice selecionado (mostrado em vermelho). Esta técnica é chamada de parametrização da superfície e visa obter uma representação 2D da superfície definida em 3D. Uma das várias aplicações desta técnica é a texturização de objetos, onde uma imagem de textura 2D pode ser mapeada à superfície de um objeto 3D.

Figura 1.5: Exemplo de uma técnica de processamento de malhas. A superfície ao redor do vértice selecionado (em vermelho) é parametrizada em duas dimensões e mostrada no canto inferior direito.

A renderização do objeto mostrado na Figura 1.5 e a janela mostrando a parametrização da superfície fazem parte desta tese. Os dados para teste escolhidos nesta pesquisa são dados escaneados, descritos por uma lista de vértices e de faces triangulares que compõe o objeto. A partir destas informações básicas, outras informações podem ser computadas, como normal à superfície e curvatura. Informações relativas ao modelo 3D, tanto básicas quanto inferidas computacionalmente, são utilizadas nesta tese para análise de malhas quanto à sua similaridade, explicadas com detalhes no Capítulo 4.

Um dos principais desafios na área de processamento de malhas é como lidar com o volume de informações a cerca do objeto de forma eficiente e objetiva. Estruturas de dados complexas e compactas podem economizar memória porém impactando negativamente no desempenho dos algoritmos. De forma similar, estruturas grandes e mais completas podem facilitar o acesso à informação porém consumindo uma quantidade proibitiva de memória. Esta preocupação na estruturação dos dados de um modelo orienta-se na vizinhança local dos elementos. Por exemplo, uma estrutura pode armazenar para cada vértice quais são os vértices vizinhos, e para cada face quais são as faces vizinhas. Este tipo de estruturação local é importante para a maioria das técnicas de processamento de malhas.

Uma outra forma de pensar nas estruturas de dados para modelos 3D é globalmente. Uma estrutura global, ou *não-local*, pode ser usada para complementar uma estrutura local, possibilitando novos conceitos em algoritmos de processamento de malhas. Por exemplo, uma estrutura pode armazenar para cada vértice quais são os vértices mais similares dado algum critério de similaridade, e técnicas de processa-

mento de malhas podem tirar vantagem desta informação adicional. A grande maioria dos objetos, sejam eles escaneados do mundo real ou modelados artificialmente, possuem regiões com propriedades quase idênticas, como cor, forma ou textura. Estas propriedades podem ser usadas como critério de similaridade na construção de estruturas de dados não-locais.

As contribuições deste trabalho de pesquisa na área de processamento de malhas remetem ao uso de padrões repetidos de forma para propagar processamento feito em uma região para demais regiões similares da malha.

1.3 Programação em GPU

Os trabalhos de pesquisa desta tese relacionados à visualização volumétrica se baseiam em programação em GPU. Uma placa gráfica programável, ou simplesmente GPU, possibilita ao desenvolvedor acessar recursos e executar programas em uma unidade de processamento massivamente paralela, rompendo fronteiras de desempenho de programação convencional em CPU. A principal diferença de programação está no conceito do processador, enquanto a CPU executa instruções sequencialmente com alto grau de controle do processamento e possui diversos níveis de *cache* para reduzir a latência de acesso à memória, a GPU executa instruções em *fluxo* restrito sobre uma grande quantidade de dados em paralelo e possui alta latência de acesso à memória. Por esta diferença conceitual, a CPU tem menos espaço em *chip* para transistores que a GPU e, logo, menos capacidade de processamento.

Tanto no meio científico quanto no comercial, a programação em GPU não é só usufruída por desenvolvedores associados à computação gráfica. Problemas que requerem alto desempenho computacional podem usar a GPU como um coprocessador massivamente paralelo da CPU. No caso da GPU ser usada para gráficos, o desenvolvedor trabalha com *shaders* e respeita a linha de produção gráfica, chamada de *pipeline* gráfico. No caso da GPU ser usada como coprocessador genérico da CPU, o desenvolvedor utiliza os multiprocessadores sem restrição de pipeline e trabalha com *kernels*. O conceito de programação por kernels é mais abrangente que por shaders, o primeiro permite qualquer tipo de entrada/saída (E/S) enquanto o segundo é restrito à parte do pipeline que é executado. Independentemente da GPU executar um shader ou um kernel, todos os seus multiprocessadores serão utilizados na tarefa. Esta gerência dos recurso de computação da placa gráfica foi introduzida com a chamada arquitetura unificada de shaders.

Os shaders podem atuar em diferentes partes do pipeline gráfico. Figura 1.6 mostra um resumo dos estágios do pipeline de acordo com o modelo unificado de shaders (versão 4). Primeiro, uma aplicação em CPU envia vértices de primitivas geométricas, e.g. pontos ou triângulos, para a placa gráfica (I). Em seguida, diversas

operações por vértice são realizadas em paralelo pelos multiprocessadores da placa. Neste ponto, um **shader de vértice** pode ser utilizado substituindo a funcionalidade fixa em *hardware* que realiza transformações geométricas, cálculos de iluminação, etc. Após este shader, os vértices transformados (II) são enviados para o **shader de geometria** que realiza a *montagem de primitivas*. Neste passo, primitivas podem ser geradas, e.g. de acordo com as primitivas enviadas antes do passo (I), ou excluídas, e.g. por operações de corte de primitivas fora do campo de visão. As primitivas geradas são então *rasterizadas* (a), processo que preenche as primitivas gerando seus *pixels*, mais corretamente chamado de *pré-pixels* ou *fragmentos*, pois ainda não formam os pixels finais do *frame buffer*.

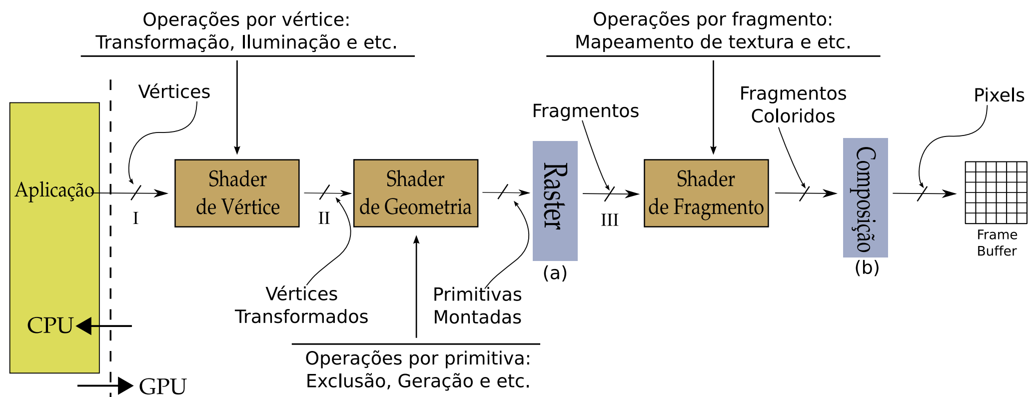


Figura 1.6: Pipeline gráfico resumido da GPU.

A cada novo estágio no pipeline, a saída é realimentada na entrada e todos os multiprocessadores da placa são utilizados. Antes da arquitetura unificada de shaders cada estágio do pipeline compreendia uma parte dos multiprocessadores da GPU e o pipeline era delineado em hardware. Atualmente o conceito de pipeline é abstrato, não estando mais presente nas arquiteturas modernas de placa gráfica.

Os fragmentos gerados pela rasterização (Figura 1.6a) são enviados para todos os multiprocessadores novamente, onde um **shader de fragmento** pode ser utilizado (III). Neste ponto, a funcionalidade fixa realiza, por exemplo, o *mapeamento de textura*, processo no qual *texels* (elementos de textura) são lidos da memória da placa gráfica e mapeados nos respectivos fragmentos de acordo com suas coordenadas de textura. Finalmente, os fragmentos coloridos, por textura ou simples atribuição de cor, são enviados para o processo de *composição* (b) responsável por agregá-los em uma matriz de pixels, chamada de frame buffer. Os fragmentos que caem no mesmo pixel podem ser compostos por uma *função de mistura* ou descartados por uma *função de profundidade*. O conteúdo do frame buffer é, normalmente, mostrado na janela gráfica da aplicação.

A memória da placa gráfica, também chamada de *memória de textura*, pode ser acessada por qualquer um dos três shaders. O acesso à memória de textura

pelos shaders é restrito à *somente-leitura* e sofre uma alta latência entre requisição e recuperação dos dados. Por este motivo, a intensidade aritmética, conceito definido pela razão de operações aritméticas por quantidade de acesso à memória, deve ser maximizada para obter alto desempenho computacional ao utilizar programação em GPU.

Exemplos de código na linguagem GLSL – *OpenGL* [1] *Shading Language* [2] – e funcionamento de cada shader e a relação entre eles são apresentados no tutorial: *Introduction to GPU Programming with GLSL* [3]. Além deste artigo estão disponíveis diversos códigos de shaders na linguagem GLSL em:

http://code.google.com/p/glsl-intro-shaders.

Em contraste com a programação em GPU usando shaders, a tecnologia CUDA – *Compute Unified Device Architecure* [4] – por exemplo, permite o desenvolvimento de aplicativos genéricos utilizando a placa gráfica como coprocessador da CPU. Em CUDA, a implementação é feita através de kernels que são executados por múltiplas *threads*. As threads são agrupadas em blocos, onde compartilham os recursos de um multiprocessador da GPU, como pode ser visto na Figura 1.7. Cada multiprocessador possui uma arquitetura SIMD (*Single Instruction Multiple Data*) que executa as mesmas instruções do kernel, porém em dados diferentes. De forma similar aos kernels, os shaders executam as mesmas instruções para múltiplos vértices, geometrias ou fragmentos.

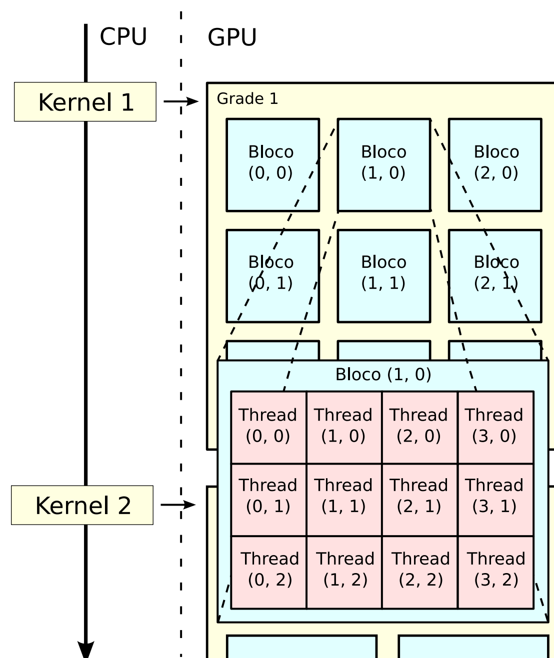


Figura 1.7: Esquema de processamento usando CUDA. Cada kernel está associado a uma grade que é dividida em blocos, que por sua vez são divididos em threads.

Os blocos de threads abstraem cada unidade de processamento em paralelo da placa, chamada multiprocessador, e são agrupados em grades. Uma grade, por sua vez, abstrai a placa gráfica em si, possuindo diversos multiprocessadores. Cada programa kernel é executado definindo o número de blocos por grade e o número de threads por bloco. Para um novo kernel ser executado depois do primeiro, uma nova grade é instanciada (veja o exemplo de Kernel 1 e 2 na Figura 1.7). Este modelo de programação simplifica o desenvolvimento de aplicações massivamente paralelas, possibilitando um nível de controle de execução entre programar usando shaders e programar para CPU.

Nesta tese, os algoritmos aprimorados de visualização volumétrica se baseiam em GPU e utilizam a linguagem de programação GLSL para shaders; e a linguagem *C for CUDA* para kernels.

Capítulo 2

Revisão Bibliográfica

"Even if you are on the right track,
you'll get run over if you just sit there."
– Will Rogers

Neste capítulo serão revisadas duas grandes áreas da computação gráfica: visualização volumétrica e processamento de malhas.

Na área de visualização volumétrica, o modelo da *integral de iluminação* [1] e *ordenação por visibilidade* são revisados e algoritmos importantes de *traçado de raios* e *projeção de células* são considerados. As técnicas mostradas aqui são usadas como base e/ou para comparação pelos algoritmos apresentados nesta tese.

Na área de processamento de malhas, algoritmos consagrados em análise de superfícies e aplicações de processamento são discutidos. Descritores e estruturas de dados apresentadas aqui reforçam a contribuição do método apresentado nesta tese.

Os trabalhos revisados estão divididos da seguinte forma:

- Seção 2.1 se refere à área de visualização volumétrica,
- com Subseções 2.1.1, 2.1.2, 2.1.3 e 2.1.4 discutindo diferentes métodos relacionados com a pesquisa desta tese;
- Seção 2.2 foca em trabalhos na área de processamento de malhas,
- com Subseções 2.2.1 e 2.2.2 discutindo diferentes técnicas relacionadas à contribuição apresentada aqui.

[1] Os termos padrões das áreas de pesquisa desta tese podem ser consultados no Glossário.

2.1 Visualização Volumétrica

Nesta seção são revisados técnicas da área de visualização volumétrica correlacionadas com os algoritmos apresentados no próximo capítulo. Esta seção está dividida em 4 partes: na Subseção 2.1.1, modelos para interação da luz com o volume são apresentados; na Subseção 2.1.2, técnicas de visualização volumétrica relacionadas com ordenação são discutidas; na Subseção 2.1.3, o método de traçado de raios é delineado e algoritmos baseados neste método são explicados; finalmente na Subseção 2.1.4, o método de projeção de células juntamente com algoritmos deste método vinculados aos trabalhos de pesquisa desta tese são elucidados.

2.1.1 Integral de Iluminação

Calcular a interação física da luz com o dado volumétrico exige a computação da *integral de iluminação* (ver Equação 2.1). A integral de iluminação é uma equação para computar a cor resultante da luz que passa através do volume. MAX [5] apresenta diferentes modelos para interação da luz com o volume na área de visualização volumétrica direta. Neste trabalho de pesquisa é utilizado o modelo de absorção mais emissão, tanto em projeção de células quanto em traçado de raios. Max mostra passo-a-passo a composição da integral até chegar à equação da integral de iluminação:

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

Através desta equação é calculada a mudança de intensidade no raio de luz I do final do volume $s = 0$ até o observador $s = D$ (veja Figura 2.1). O raio de luz atravessa uma distância D até o observador, onde o primeiro termo representa a quantidade de luz de entrada I_0, atenuada exponencialmente pela distância D. O segundo termo adiciona a quantidade de luz emitida por cada ponto ao longo do raio, levando em consideração a quantidade atenuada do ponto ao final do raio.

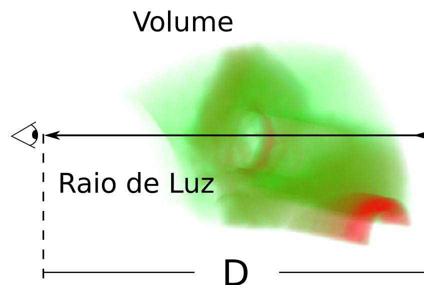


Figura 2.1: Modelo da integral de iluminação, com o raio de luz percorrendo uma distância D do final do volume até o observador.

O cálculo desta integral quadro-a-quadro para todos os pixels da imagem é um processo dispendioso computacionalmente. Alguns trabalhos [6–8] melhoram o desempenho de seus métodos de visualização simplificando essa integral. O modelo proposto por estes trabalhos difere da Equação 2.1 (veja a Figura 2.2). Neste modelo, o raio de visão é utilizado (partindo do observador até o final do modelo) ao invés do raio de luz. Note que a integral depende apenas da distância l (*length*) percorrida dentro de cada *célula do volume*, chamada de *espessura*, e os valores de entrada s_f (*scalar front*) e saída s_b (*scalar back*) do raio. O resultado da integral fornece então a parcela de contribuição da célula para a iluminação do pixel.

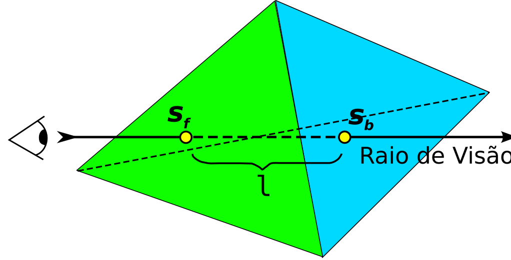


Figura 2.2: Modelo simplificado da integral de iluminação, utilizando apenas o escalar da frente s_f, de trás s_b e a espessura l de cada célula do volume.

Em um modelo simples, a cor pode ser obtida pela média das cores de entrada $C(s_f)$ e de saída $C(s_b)$, como expressado por SHIRLEY e TUCHMAN [6]:

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

Da mesma forma, a opacidade α pode ser calculada usando a média dos coeficientes de extinção de entrada $\tau(s_f)$ e de saída $\tau(s_b)$:

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

As cores e os coeficientes de extinção são diretamente associados aos valores escalares de entrada s_f e de saída s_b através da função de transferência [9], que define $\tau()$ e $C()$ nas Equações 2.2 e 2.3.

A cor e opacidade (RGBA) final do pixel, representada por I na Equação 2.1, são computadas pela combinação das células atravessadas por um mesmo raio. Para cada célula além da primeira, a cor e opacidade atualizadas C_{i+1} e α_{i+1} são as combinações lineares das cores anteriores C_i e C_{i-1}, e opacidades anteriores α_i e α_{i-1}, com a seguinte regra:

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

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

Outra forma de computar cor e opacidade é utilizando uma interpolação linear entre os valores de entrada e saída do raio. Considerando um espaço ortogonal e a integração realizada ao longo do eixo z, BUNYK *et al.* [10] expressam cor e opacidade com as seguintes funções:

$$C(z) = \frac{(z_b - z)C(s_f) + (z - z_f)C(s_b)}{\Delta_z}, \tag{2.6}$$

$$\alpha(z) = \frac{(z_b - z)\alpha(s_f) + (z - z_f)\alpha(s_b)}{\Delta_z}. \tag{2.7}$$

Nestas funções lineares, o valor de $C(z)$ e $\alpha(z)$ correspondem ao valor de cor e opacidade no ponto z ao longo do raio. Estas funções devem ser integradas de z_f, valor z de entrada (z *front*), até z_b, valor z de saída (z *back*), para obter C_{i+1} e α_{i+1}, i.e. cor e opacidade acumulada no passo atualizado:

$$C_{i+1}(z) = C_i + \int_{z_f}^{z} C(z)(1 - \alpha(z))dz, \tag{2.8}$$

$$\alpha_{i+1}(z) = \alpha_i + \int_{z_f}^{z} \alpha(z)dz. \tag{2.9}$$

Note que neste modelo simplificado, utilizado por BUNYK *et al.* [10], a opacidade é calculada apenas por interpolação linear, ou seja, a atenuação exponencial da Equação 2.1 é desconsiderada. Resolvendo as integrais das Equações 2.8 e 2.9 analiticamente, eles chegam as seguintes equações para o cálculo de cor e opacidade durante a integração do raio:

$$C_{i+1} = C_i + \frac{1}{2}(C_f + C_b)(\alpha_f - 1)\Delta_z - \frac{1}{24}(3C_f\alpha_f + 5C_b\alpha_f + C_f\alpha_b + 3C_b\alpha_b)\Delta_z^2, \tag{2.10}$$

$$\alpha_{i+1} = \alpha_i + \frac{1}{2}(\alpha_f + \alpha_b)\Delta_z. \tag{2.11}$$

Uma forma de contornar o cálculo quadro-a-quadro da integral de iluminação é computá-la previamente, armazenando os possíveis resultados discretizados em tabela. Esta técnica, chamada de *pré-integração*, foi introduzida no contexto de programação em GPU por RÖTTGER *et al.* [7].

Uma desvantagem do uso da técnica de pré-integração é que se a função de transferência for alterada, a aplicação precisa recalcular toda a tabela da integral de iluminação e armazená-la novamente. Este procedimento é muito dispendioso computacionalmente, dificultando a edição interativa da função de transferência e, por consequência, reduzindo as opções de interatividade na visualização do dado volumétrico.

MORELAND e ANGEL [11] apresentam uma solução diferente da pré-integração. Eles introduzem o conceito de *pré-integração parcial*, onde apenas parte da integral de iluminação é computada e armazenada em tabela. Eles modificam a integral de iluminação, tornando-a independente da função de transferência. Uma explicação mais detalhada da construção da tabela ψ de pré-integração parcial pode ser encontrada na tese de doutorado de MORELAND [12].

Este trabalho de pesquisa apresenta métodos de visualização volumétrica direta e indireta que utilizam algumas equações apresentadas aqui. A técnica de pré-integração parcial também é utilizada, analisando vantagens e desvantagens de diferentes tipos de visualização e integração do volume.

2.1.2 Ordenação por Visibilidade

Algoritmos de visualização volumétrica direta envolvem composição e, por este motivo, dependem de uma ordenação correta das células para um dado ponto de vista para atravessar o volume. Se por um lado algoritmos de traçado de raios empregam uma estrutura auxiliar de adjacência de forma que quando um raio deixa uma célula ele tenha informação suficiente para encontrar a próxima. Por outro lado, algoritmos de projeção de células usualmente computam uma ordenação aproximada em *espaço de objeto*. Apesar de haver algoritmos exatos para ordenação de células [13–15], eles são complexos e computacionalmente dispendiosos.

Uma abordagem visando combinar o melhor de projeção de células e traçado de raios é o *View-Independent Cell Projection* (VICP) de WEILER *et al.* [16]. Fazendo apenas traçado de raios dentro de cada célula projetada, o VICP alcança alta qualidade de imagens consumindo menos memória que algoritmos de traçado de raios. CALLAHAN *et al.* [17] apresentam uma abordagem de ordenação aproximada nomeada *Hardware-Assisted Visibility Sorting* (HAVS), que usa uma estratégia híbrida de visualização volumétrica como o método VICP. Primeiro, faces do volume são ordenadas em espaço de objeto pelos seus centróides em CPU e renderizadas usando rasterização normal de triângulos. Depois, integração do raio é avaliada em *espaço de imagem*, enquanto um método de ordenação refinado é realizado usando a técnica introduzida no HAVS de *k-buffer*, onde k determina a precisão de ordenação, balanceando entre desempenho e qualidade. Uma desvantagem é que por unir ordenação com renderização o algoritmo HAVS fica limitado por um estrutura fixa, proibindo o algoritmo de usar uma técnica de ordenação exata.

2.1.3 Traçado de Raios

A técnica de *traçado de raios* data da década de 60 e seus conceitos foram inicialmente considerados em visualização volumétrica por BLINN [18] na década de 80.

Na concepção de Blinn, um raio de luz é lançado para cada pixel da tela, computando a absorção de luz por onde ele atravessa, como pode ser visto no exemplo da Figura 2.3.

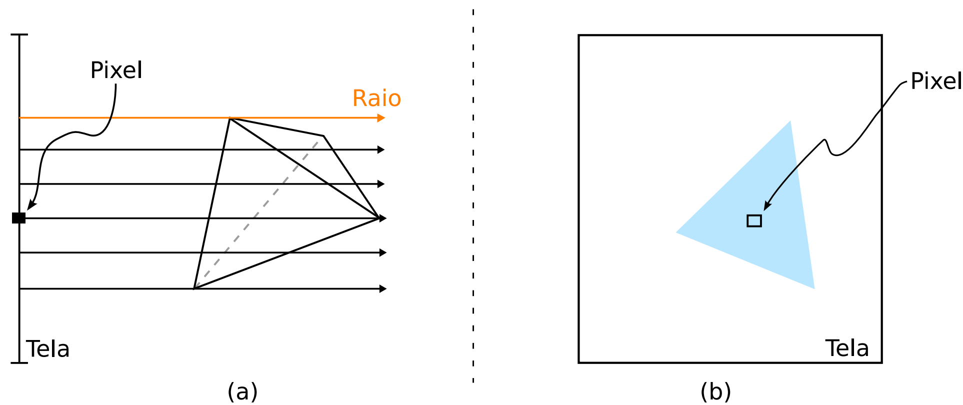


Figura 2.3: Exemplo da técnica de traçado de raios. A visão lateral dos raios atravessando uma célula tetraedral do volume por pixel da tela é mostrada em (a) e o resultado em (b).

Posteriormente, o trabalho de GARRITY [19] apresenta uma abordagem mais eficiente para o algoritmo de traçado de raios. Seu método traça raios em dados irregulares com transparência, usando a conectividade entre as células do volume para computar o *caminho do raio*, i.e. descobrir as células que foram intersectadas pelo raio a partir de um pixel até o final do volume.

Esta abordagem foi aperfeiçoada por BUNYK *et al.* [10] onde todas as células são quebradas em faces, em um passo de pré-processamento. Desta forma, as *faces externas*, também chamadas de *faces da borda*, são projetadas para determinar as *faces visíveis* (veja a Figura 2.4). As faces visíveis definem o ponto de entrada do raio para cada pixel da imagem. Com todas as faces criadas e guardadas em memória, BUNYK *et al.* [10] reduzem os cálculos de intersecção e integração do raio ganhando em desempenho, porém aumentando o consumo de memória. Os trabalhos de Garrity e Bunyk *et al.* formam a base do algoritmo de traçado de raios apresentado por este trabalho na Seção 3.1.

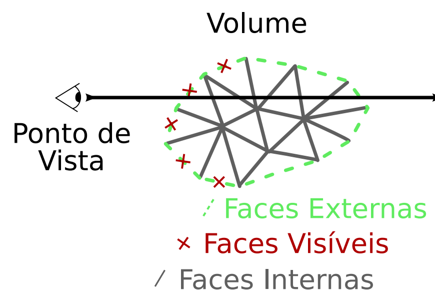


Figura 2.4: Tipos de faces de um dado volumétrico: faces externas e visíveis ficam na fronteira do volume; enquanto faces internas ficam dentro do volume.

WEILER *et al.* [20] apresentam um método para realizar traçado de raios em GPU, usando shader de fragmento, chamado *Hardware-Assisted Ray Casting* (HARC). No algoritmo HARC é encontrada a entrada inicial do raio renderizando as faces externas, de forma similar a ideia de Bunyk *et al.*. O algoritmo atravessa o volume, armazenando as computações das células em texturas.

ESPINHA e CELES [21] incrementam o algoritmo HARC usando pré-integração parcial, ao invés de pré-integração normal, e empregando uma estrutura de dados mais eficiente que a implementação do HARC original. O algoritmo proposto por eles alcança alta qualidade e possibilita a alteração interativa da função de transferência.

No Capítulo 5, os resultados do algoritmo HARC e da versão melhorada de Espinha e Celes são analisados e comparados com os algoritmos apresentados neste trabalho de pesquisa.

2.1.4 Projeção de Células

A técnica de *projeção de células*, também chamada de *projeção direta*, visa a geração de imagens do volume a partir da projeção de suas células na tela. A projeção é determinada transformando as células tridimensionais em primitivas geométricas bidimensionais no *plano da imagem*, ou *plano de visão*. Depois que as projeções das células são determinadas (veja exemplo na Figura 2.5), o processo de rasterização preenche as primitivas geométricas geradas com fragmentos, que são combinados na composição final do pixel.

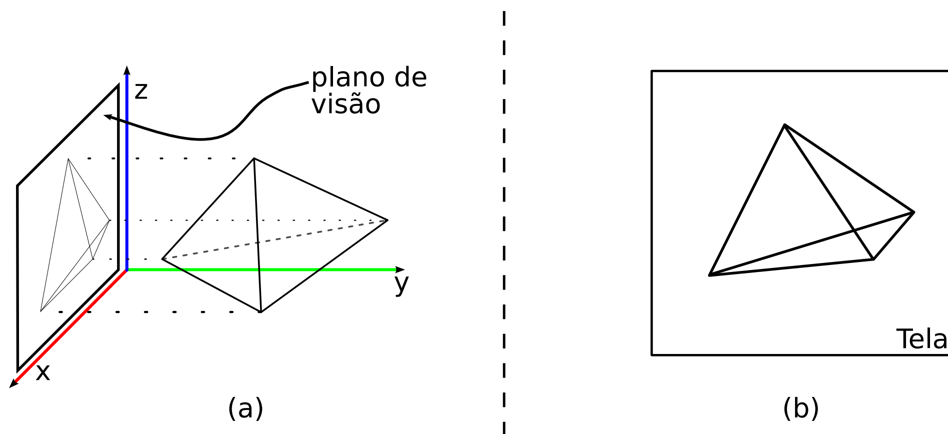


Figura 2.5: Exemplo da técnica de projeção de células. A projeção de uma célula do volume é mostrada em (a) e o resultado em (b).

A principal vantagem da projeção de células sobre o traçado de raios é que, na técnica de projeção de células, todas as intersecções dos raios com a célula são computadas implicitamente na projeção. Enquanto que, na técnica de traçado de raios, as intersecções devem ser computadas para cada raio, mesmo que raios próxi-

mos intersectem a mesma célula. O trabalho de UPSON e KEELER [9] discute as vantagens e desvantagens dos métodos de traçado de raios e projeção de células.

SHIRLEY e TUCHMAN [6] introduziram o primeiro algoritmo de projeção de células em dados volumétricos irregulares. O algoritmo trata exclusivamente de células tetraedrais e, por esta razão, foi nomeado de *Projected Tetrahedra* (PT), ou *Projeção de Tetraedros*. O algoritmo PT consiste em projetar e classificar tetraedros no plano da imagem e compô-los em ordem de visibilidade. Este algoritmo forma a base dos algoritmos de projeção de células apresentados nesta tese no Capítulo 3 e é explicado no Apêndice C.

KRAUS *et al.* [22] melhoram a qualidade das imagens geradas pelo PT aplicando uma escala logarítmica para a tabela de pré-integração. Além disso, Kraus *et al.* apontam o uso de texturas com maior precisão (16 bits por componente de cor) como responsável por parte da melhora da qualidade.

WEILER *et al.* [16] desenvolveram outro método de projeção de células implementado completamente em GPU, usando shader de vértice e fragmento. O algoritmo de Weiler *et al.* , chamado *View Independent Cell Projection* (VICP), aplica o mesmo shader de vértice e fragmento independente do *ponto de vista*. O algoritmo VICP combina a ideia de traçado de raios, de forma similar ao HARC, com a técnica de projeção de células. Ambos os algoritmos de Weiler *et al.* (HARC e VICP) utilizam uma textura com a tabela de pré-integração, sugerida por RÖTTGER *et al.* [7]. A cor final do pixel é determinada pelo processo de composição dos fragmentos usando as Equações 2.4 e 2.5.

WYLIE *et al.* [8] propõem implementar o algoritmo PT na placa gráfica, usando shader de vértice. O principal empecilho no uso do shader de vértice para este algoritmo é o número fixo de vértices de entrada/saída, comprometendo a ideia original do PT que define uma forma de projeção variável, de 1 a 4 triângulos, dependendo da projeção do tetraedro. Visando resolver este problema, o algoritmo criado por Wylie *et al.* , chamado *GPU-Accelerated Tetrahedra Renderer* (GATOR), classifica as projeções dos tetraedros de forma diferente do algoritmo PT de SHIRLEY e TUCHMAN [6]. No algoritmo GATOR é utilizada uma topologia fixa, chamada de *grafo base*, isomorfa à projeção bidimensional do tetraedro no plano da imagem. Como resultado, a limitação de entrada/saída fixa do shader de vértice é evitada, possibilitando ao algoritmo GATOR gerar imagens como o algoritmo PT.

O algoritmo introduzido por MARROQUIM *et al.* [23], chamado *Projected Tetrahedra with Partial Pre-Integration* (PTINT), realiza o algoritmo PT em dois passos, evitando o problema enfrentado pelo algoritmo GATOR. O algoritmo PTINT, explicado no Apêndice D, forma a base dos algoritmos de projeção de células apresentados nesta tese.

2.2 Processamento de Malhas

Nesta seção são revisados técnicas da área de processamento de malhas correlacionadas com o método apresentado no Capítulo 4. Esta seção está dividida em 2 partes: na Subseção 2.2.1, o método básico de comparação por similaridade usando reflexão é introduzido; e na Subseção 2.2.2, descritores de similaridade por assinatura vinculados a esta tese são discutidos.

2.2.1 Simetria Refletivas

Simetria em modelos 3D são normalmente detectadas e descritas como reflexões planares entre partes na superfície de uma malha [24–26]. Um exemplo de plano de reflexão pode ser visto na Figura 2.6, onde o modelo *Stanford Armadillo*, de um boneco de um tatu escaneado pela *Universidade de Stanford*, apresenta simples simetria bilateral. A renderização deste modelo é parte desta tese e ilustra um dos objetos usados nos testes do método introduzido aqui.

Figura 2.6: Exemplo de plano de reflexão no centro do modelo, com cada metade apresentando forma aproximadamente simétrica na superfície.

Em adição à simetria refletiva ou espelhada, semelhanças entre detalhes na superfície também podem ser encontradas [27–29]. O conceito de auto-similaridade de uma malha é definido por essas semelhanças, e naturalmente generaliza o conceito de simetria refletiva. Nesta tese é dito que duas ou mais partes de uma malha são similares quando elas compartilham características locais de superfície, sejam refletivas ou não. O método introduzido neste trabalho de pesquisa utiliza descritores de similaridade para propagar processamento através de regiões similares de

uma malha. Apesar de detecção e análise estrutural de simetrias refletivas terem sido usadas para aprimorar algumas técnicas de processamento de malhas [30], existem poucos trabalhos [31, 32] lidando com padrões repetidos mas sem a ideia de propagar processamento pela malha apresentada nesta tese.

Medidas de auto-similaridade em malhas são empregadas como uma ferramenta em diversas aplicações, como *re-triangulação* [33–35], renderização [36], teste [37] e recuperação [38–40] de forma da malha, e entendimento de cena [41]. As técnicas de detecção de auto-similaridade empregadas nestas aplicações geralmente utilizam reflexão como método de comparação base para determinar regiões similares. O uso de reflexão na maioria destas técnicas vem do fato que simetria na natureza tende a ter padrões repetidos entre metades, como por exemplo o rosto humano, o corpo de um animal, ou uma folha de planta. Entretanto, objetos ambos naturais e artificiais podem também conter classes mais gerais de auto-similaridades, por exemplo um teclado de computador tem a maioria de suas teclas compartilhando uma forma similar.

Uma das contribuições desta tese é um método de detecção de auto-similaridade não limitado àqueles baseados em reflexão, que o torna adequado para malhas escaneadas ou modeladas, inspiradas tanto por objetos naturais como artificiais.

2.2.2 Descritores de Similaridade

Um dos aspectos que torna a identificação de auto-similaridades especialmente desafiadora é a ausência de uma ferramenta de medida eficaz para comparação de formas locais da superfície. GATZKE *et al.* [42] apresentam um método para comparar diferentes regiões locais, introduzindo o *mapa de curvaturas* de um ponto. Eles avaliam as curvaturas médias e Gaussianas como uma função de distância do ponto, usando *anéis de vizinhança* ou *leque geodésico* como ZELINKA e GARLAND [31], para cada ponto da malha. Em seguida, a auto-similaridade da superfície é medida como a diferença entre essas funções de curvatura. O mapa de curvaturas usado por GATZKE *et al.* [42] funciona como um descritor do vértice, onde vértices similares na malha possuem descritores aproximadamente iguais.

Um outro exemplo de descritor de vértice pode ser visto na Figura 2.7. O descritor, ou assinatura, neste exemplo é um mapa de alturas (mostrado no canto inferior esquerdo) assumindo como base uma região ao redor do vértice selecionado (em vermelho) e o plano tangente à superfície neste vértice definido pela posição e normal do vértice. Este descritor de vértice é parte da contribuição desta tese, que o utiliza ao invés de uma função de curvatura como assinatura de um vértice da malha para descrever auto-similaridade. Tanto a construção do mapa de alturas quanto o seu uso são detalhados no Capítulo 4.

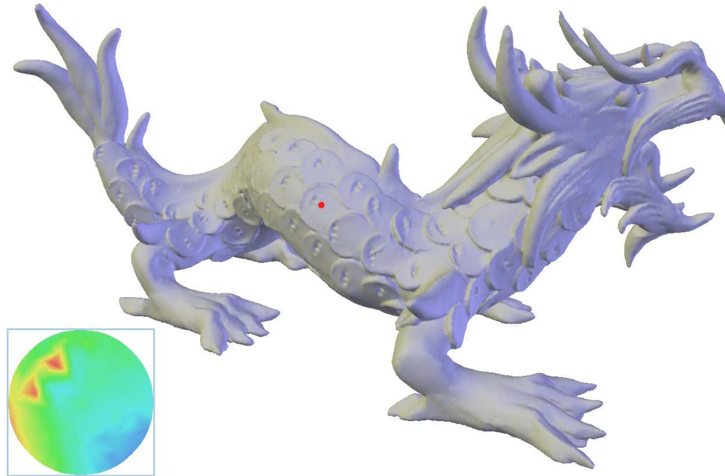


Figura 2.7: Exemplo de assinatura de um vértice (em vermelho) no modelo XYZ RGB Asian Dragon. A assinatura é um mapa de alturas da região ao redor do vértice, note as duas cavidades da escama do dragão mais fundas que as demais partes do mapa no canto inferior esquerdo.

Técnicas visando identificar auto-similaridades dependem principalmente das propriedades de forma de um dado modelo. Vários trabalhos recentes em processamento de geometria consideram funções coordenadas, ou uma função mais geral, como um sinal definido na superfície da malha. Um exemplo é o método de compressão espectral de malhas proposto por KARNI e GOTSMAN [43]. Neste método, a informação geométrica é codificada como uma combinação linear compacta dos autovetores ortogonais do grafo Laplaciano discreto. O método de compressão espectral é baseado em geometria discreta diferencial não detalhada nesta tese. O leitor pode referir ao curso completo chamado *Discrete Differential Geometry: An Applied Introduction* [44] em busca de mais informações.

VALLET e LÉVY [45] descrevem como usar auto-vetores de uma formulação assistida por geometria do operador *Laplace-Beltrami* como uma função base para representação de geometria da malha; eles chamam esta base ortogonal de *Base Variedade Harmônica*.

OVSJANIKOV *et al.* [28] apresentam um método para computar simetrias intrínsecas globais usando autofunções. O método deles determina correspondência invariante de pose sobre uma forma, i.e. simetrias que permanecem intactas sob deformações isométricas. Entretanto, eles restringem o método para simetrias refletivas com base nos eixos principais.

SUN *et al.* [29] aprimoram o método de OVSJANIKOV *et al.* [28] definindo uma assinatura de vértice local, chamada *Heat Kernel Signature* (HKS). A assinatura HKS é um descritor multi-escala da forma circundando o vértice baseado na evolu-

ção temporal de um processo de difusão do calor na malha. Apesar deste método prover um descritor de vértice robusto, ele não admite prontamente um descritor de forma baseado em regiões necessário para o método de propagação de processamento introduzido nesta tese.

Depois da identificação de auto-similaridade em uma malha, a questão que permanece é como descrevê-las sucintamente. Métodos definindo assinaturas por vértice, como o HKS ou o mapa de curvaturas, são uma maneira de descrever similaridades. Outra maneira de definir similaridades é usar uma estrutura de dados mais global. SIMARI *et al.* [46] apresentam um método para computar o que eles chamam de *folding tree*, uma estrutura de dados compacta usando simetria planar. O método deles, entretanto, é restrito à localizar simetria acerca dos eixos principais. Detecção de simetria acerca de planos mais gerais, como aqueles definidos por *análise de componentes principais* (PCA), foram exploradas por KAZHDAN *et al.* [24] e CHENG *et al.* [47]. Talvez o método mais geral para análise de simetria do objeto inteiro é o *Planar-Reflective Symmetry Transform* (PRST) [25] que captura as simetrias refletivas com respeito a todos os possíveis planos. XU *et al.* [26] aprimoram a ideia do PRST para possibilitar a detecção de simetrias rotacionais intrínsecas.

Descritores de similaridade, variando de local (por vértice) a global (malha inteira), adicionam informação sobre uma superfície. GOLOVINSKIY *et al.* [30] apresentam um sistema para explorar tais informações, descrevendo ferramentas de processamento de malhas para detectar e preservar simetrias usando o PRST e o trabalho de MITRA *et al.* [48] em detecção de simetrias parciais. O processamento de malhas assistido por simetria de GOLOVINSKIY *et al.* [30] é guiado pelas simetrias de uma superfície, enquanto que o trabalho apresentado nesta tese usa as similaridades conhecidas para replicar computação.

O uso de descritores de similaridades para auxiliar em processamento de malhas foi também explorado por YOSHIZAWA *et al.* [49]. Eles usam *funções de base radiais* (RBFs) para aproximar vizinhança local de vértices e descrever similaridade pela diferença entre as formas locais codificadas nestas RBFs. Estas diferenças são usadas como pesos para remover ruído de malhas baseado em técnicas de remoção de ruído de imagens. Outro trabalho visando o filtro de ruídos de malhas é o apresentado por SCHALL *et al.* [50]. Em contraste com o trabalho de YOSHIZAWA *et al.* [49], eles lidam com dados de scanner bruto (*range image data*), computando uma diferença de altura ponto-a-ponto dos vizinhos ao invés de usar RBFs. O método de detecção de similaridade introduzido nesta tese também computa diferenças de altura ao redor de um vértice, entretanto o método apresentado pode ser usado para qualquer tipo de malha, não só dados brutos de scanner. Além disso, a abordagem deste trabalho de pesquisa pode ser aplicada a diversas técnicas de processamento de malhas, como parametrização de superfícies e transferência de detalhe.

Capítulo 3

Visualização Volumétrica

"... in 10 years, all rendering
will be volume rendering."
– Jim Kajiya at SIGGRAPH '91

Neste capítulo apresentamos três algoritmos de projeção de células baseado no PT e um algoritmo de traçado de raios, compreendendo as contribuições desta tese na área de visualização volumétrica. Os quatro algoritmos são: *VF-Ray-GPU – Visible-Face Driven Ray Casting implemented on the GPU* – um método de traçado de raios eficiente no uso de memória e completamente implementado em GPU; *RPTINT* e *IPTINT – Regular and Improved Projected Tetrahedra with Partial Pre-Integration* – duas técnicas baseadas no algoritmo PT original [6], onde a primeira é especializada para dados regulares e a segunda combina visualização volumétrica direta e indireta; *HAPT – Hardware-Assisted Projected Tetrahedra* – uma adaptação para *hardware* gráfico mais próxima do algoritmo PT, capaz de extrair iso-superfícies e tratar dados que variam no tempo interativamente. Os algoritmos de visualização volumétrica apresentados nesta tese abrangem projeção de células e traçado de raios, tratando dados regulares e irregulares, e empregando ambas as técnicas de visualização volumétrica direta e indireta.

O algoritmo VF-Ray-GPU, explicado na Seção 3.1, é baseado no algoritmo *VF-Ray* (*Visible-Face Driven Ray Casting*) [51] explicado no Apêndice B, porém implementando-o em placa gráfica e modificando suas estruturas de dados. Os algoritmos RPTINT e IPTINT, apresentados nas Seções 3.2 e 3.3, são baseados no algoritmo PTINT [23] que por sua vez é baseado no algoritmo PT [6], explicados respectivamente nos Apêndices D e C. Finalmente na Seção 3.4, o algoritmo HAPT é detalhado.

3.1 VF-Ray-GPU

VF-Ray O algoritmo *Visible-Face Driven Ray Casting* (VF-Ray) é parte da pesquisa de doutorado em andamento de Ribeiro, e foi publicado na conferência internacional *SIBGRAPI* 2007 [51]. No algoritmo VF-Ray, malhas irregulares compostas de células tetraedrais ou hexaedrais são tratadas. Cada célula tetraedral é composta de quatro faces e cada célula hexaedral é composta de seis faces. O algoritmo é baseado na seguinte premissa: o armazenamento das informações acerca das faces de cada célula é a chave para o consumo de memória e tempo de execução. Estas informações são guardadas em uma estrutura de dados de face que incluem sua geometria e seus parâmetros, i.e. constantes da equação do plano definido pela face, que são os dados que mais consomem memória no traçado de raios.

A ideia básica por trás do VF-Ray é explorar a coerência entre raios em uma vizinhança, diminuindo o consumo total de memória. No algoritmo VF-Ray apenas os dados de faces intersectadas por um conjunto de raios próximos são mantidos. O conjunto considerado é determinado pelos pixels contidos na projeção de uma dada face visível, como ilustrado na Figura 3.1. A este conjunto de pixels é dado o nome de *conjunto visível*.

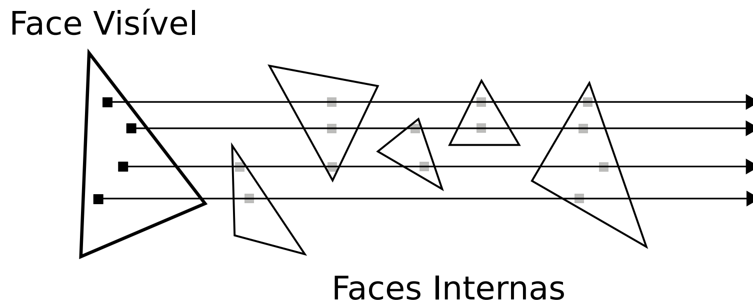


Figura 3.1: Coerência entre os raios lançados por uma mesma face visível.

O algoritmo começa determinando as faces visíveis dado o conjunto de faces externas do volume e o ponto de vista corrente. As faces externas são pré-computadas de maneira similar ao algoritmo de BUNYK *et al.* [10], assim como a determinação das faces visíveis. Em suma, faces externas pertencem somente a uma célula do volume e são fixas para cada modelo, enquanto faces visíveis são as faces externas que possuem seu vetor normal apontando na direção oposta do *vetor de visão*, classificação esta que depende do ponto de vista.

No algoritmo VF-Ray são processadas as faces visíveis uma de cada vez, projetando-as no plano da imagem e determinando o seu conjunto visível. Para cada pixel deste conjunto, um raio é lançado e suas intersecções contra *faces internas* e externas são computadas. As faces internas são todas as faces do volume que não são externas, ou seja, faces compartilhadas por duas células. As faces atingidas

por um raio são determinadas testando a intersecção do raio contra cada face de cada célula até que o raio saia do volume. As faces intersectadas são criadas durante o processo, i.e. seus parâmetros de interpolação são computados.

A ideia principal do VF-Ray é guardar cada face intersectada em um *buffer*, chamado de *computedFaces*. Toda vez que um raio lançado do conjunto visível corrente atravessa uma face previamente guardada no buffer, o algoritmo VF-Ray lê a face do buffer ao invés de recomputar os parâmetros da face. Estes parâmetros de interpolação são usados para computar a distância percorrida (*length* l) pelo raio dentro da célula e os escalares de entrada e saída (*scalar front* s_f e *back* s_b) através da equação do plano da face e a equação de linha do raio. Finalmente, a contribuição da célula para cor e opacidade do pixel é computada usando l, s_f e s_b com o modelo de iluminação de Bunyk *et al.* explicado na Seção 2.1.1, onde a diferença do valor z de entrada e saída (z_f e z_b) dada por Δ_z é equivalente a distância l em uma projeção não necessariamente ortogonal.

O algoritmo VF-Ray explora a coerência entre raios, usando a vizinhança de pixels da face visível para guiar a criação e destruição dos dados de faces internas em memória. As faces intersectadas por raios vizinhos tendem a serem as mesmas, como exemplificado na Figura 3.1, e seus dados são guardados no buffer *computedFaces* enquanto os raios do conjunto visível são lançados. Depois que todos os pixels de um conjunto visível são processados, o algoritmo VF-Ray limpa o buffer *computedFaces* e prossegue para a próxima face visível.

Casos Degenerados Situações de casos degenerados acontecem quando um raio acerta uma aresta ou um vértice do modelo. O método e as estruturas de dados propostas por Bunyk *et al.* não tratam corretamente estes casos degenerados, gerando cores incorretas para os pixels. No algoritmo VF-Ray, por outro lado, estes casos são tratados da mesma maneira como proposto por PINA *et al.* [52]. A ideia é pré-computar uma nova estrutura de dados chamada *Use_set*, contendo todas as células incidentes em cada vértice do volume. Assim, o *Use_set* pode ser utilizado para continuar a computação de um raio, mesmo que este tenha atingido um vértice ou uma aresta, garantindo que a imagem final seja gerada corretamente.

VF-Ray-GPU O algoritmo *Visible-Face Driven Ray Casting implemented on the GPU* (VF-Ray-GPU) é baseado no conceito de implementação *GPGPU* (*General Purpose Computation on Graphics Hardware*) para paralelizar o algoritmo VF-Ray original (veja Apêndice B). Neste conceito, a placa gráfica é vista como um co-processador capaz de executar em paralelo tarefas de alta intensidade aritmética, independente do pipeline gráfico da GPU. O algoritmo VF-Ray-GPU foi publicado no simpósio internacional *Volume Graphics* 2008 [53].

A arquitetura utilizada para implementar o algoritmo apresentado nesta tese de traçado de raios – VF-Ray-GPU – foi CUDA [4] devido a sua simplicidade e possibilidade de explorar todos os recursos da placa gráfica. Como visto na Seção 1.3, a implementação em CUDA é realizada através de um kernel associado a uma grade de blocos, onde cada bloco possui múltiplas threads que executam em paralelo. As threads têm uma capacidade de execução limitada, o que ocasiona em dois problemas principais: se a função associada ao kernel for muito extensa, a execução pode falhar; e a quantidade de recursos alocados, e.g. registradores, podem passar do limite, impossibilitando a compilação do código. Esta limitação de recursos depende do número de threads por bloco, pois quanto maior este número menor a quantidade de recursos por multiprocessador disponível.

O algoritmo VF-Ray-GPU é dividido em três passos e, como resultado, em três kernels diferentes (veja a Figura 3.2). No primeiro kernel, as faces externas são lidas da memória de textura e as faces visíveis são determinadas. O segundo kernel é responsável por computar as projeções de cada face visível, dividindo-as em pixels no plano da imagem. Finalmente, no terceiro kernel é executado o algoritmo de traçado de raios para cada pixel da face visível previamente projetada.

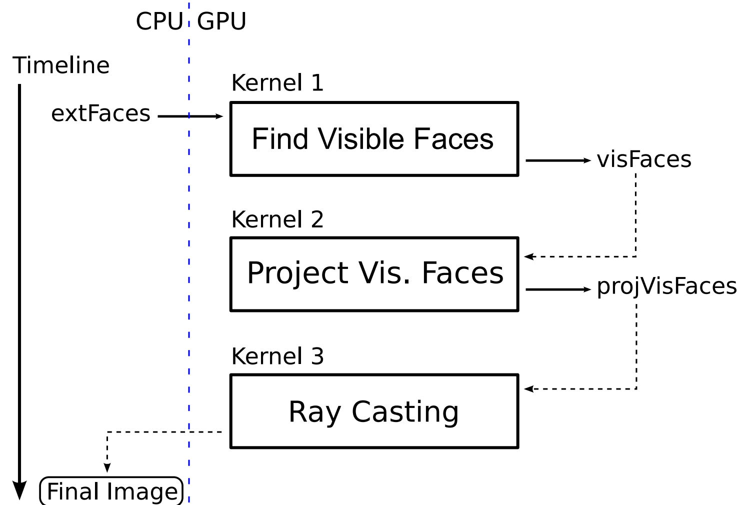


Figura 3.2: Os três kernels do algoritmo VF-Ray-GPU. As faces externas *extFaces* são lidas pelo kernel 1 (*Find Visible Faces*), responsável por determinar as faces visíveis *visFaces*. Em seguida, as faces visíveis são projetadas no plano da imagem pelo kernel 2 (*Project Vis. Faces*) e o traçado de raios é realizado usando as faces projetadas *projVisFaces* no kernel 3 (*Ray Casting*).

Estruturas de Dados Antes de processar os três kernels, as seguintes estruturas de dados são usadas pelo algoritmo VF-Ray-GPU. A conectividade dos tetraedros (*conTet*) é computada de maneira similar aos trabalhos de GARRITY [19] e BUNYK *et al.* [10], isto é a partir das informações básicas do dado volumétrico: a lista

de vértices (*vertList*) e a lista de tetraedros (*tetList*). A lista *conTet* armazena para cada face de cada tetraedro, o índice do tetraedro t_i que compartilha aquela face. No caso de uma face externa, o índice t_i armazenado é o próprio índice do tetraedro. Na Figura 3.3 as estruturas de dados usadas pelo algoritmo VF-Ray-GPU são mostradas, ilustrando o primeiro tetraedro ($tetraedro_0$). A lista *vertList* contém as coordenadas x, y e z, e o valor escalar s de cada vértice. A lista *tetList* contém os índices dos vértices v_i que compõem cada tetraedro.

Para evitar construir outras estruturas de dados, o algoritmo em GPU utiliza a ordem dos vértices dentro da *tetList* para determinar cada face do tetraedro. Assim, os vértices da face f_i são definidos por v_i, $v_{(i+1)mod4}$ e $v_{(i+2)mod4}$. Por exemplo, a face f_2 de um dado tetraedro t_i é composta pelos vértices v_2, v_3 e v_0 de t_i, como pode ser visto na direita da Figura 3.3. Em adição a essas estruturas de dados, o algoritmo VF-Ray-GPU utiliza a lista de faces externas (*extFaces*) pré-computada da mesma maneira que o algoritmo VF-Ray original.

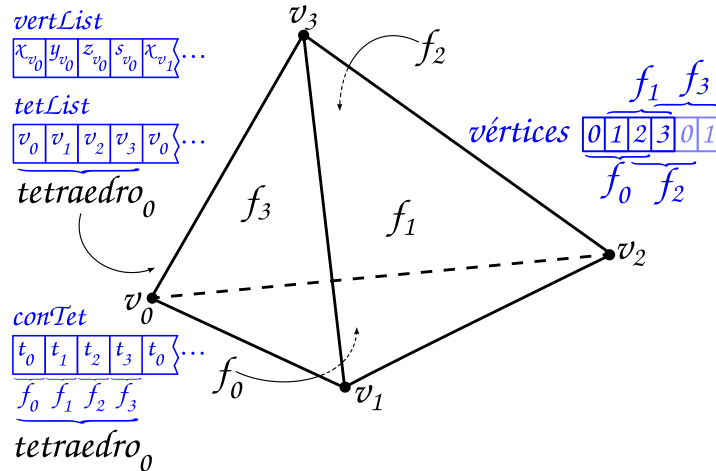


Figura 3.3: As estruturas de dados básicas do algoritmo VF-Ray-GPU.

No algoritmo VF-Ray original são guardados três índices de vértices dentro da estrutura de dados da face, apontando para os três vértices da face na *vertList*. No algoritmo em GPU são guardados apenas dois índices por face criada. Estes índices são t_i do tetraedro e f_i da face, que são suficientes para identificar os vértices de cada face, como explicado anteriormente. Esta redução no número de índices usado na identificação de uma face economiza memória, porém aumenta a quantidade de acessos uma vez que a *tetList* deve ser acessada antes de ler cada vértice da face.

Primeiro Kernel No primeiro kernel são lidas as faces externas da memória de textura e seus parâmetros são computados. Esta computação é chamada de *criação da face* e visa resolver dois sistemas lineares 3×3: um para interpolar a coordenada z; e outro para interpolar o valor escalar s. Neste kernel, somente a interpolação

da coordenada z é feita para cada face externa e, uma vez que esta face é marcada como visível, a interpolação do valor escalar s é realizada. Ambos os parâmetros de interpolação são guardados na lista de faces visíveis (*visFaces*).

A visibilidade de uma face externa é determinada comparando a coordenada z do quarto vértice do tetraedro, o vértice que não pertence a face, com a coordenada z de sua projeção na face externa. A projeção do quarto vértice é feita usando os parâmetros de interpolação da coordenada z, computados para aquela face externa. A face é visível se o z projetado é menor que a coordenada z do quarto vértice. Note que esta comparação é equivalente ao teste de *back face culling*, onde a normal da face é comparada contra o vetor de visão usando produto interno.

Somente os parâmetros das faces visíveis são escritos em memória global em CUDA, ou seja, apenas os parâmetros de interpolação de z e s das faces visíveis são guardados na lista *visFaces*, como ilustrado na Figura 3.2.

O primeiro kernel emprega a criação da face e o teste de visibilidade para cada thread, usando o número máximo de threads por bloco disponível. O número total de threads em todos os blocos é fixo, pois depende apenas do número de faces externas do volume que não varia com o ponto de vista. O tempo de computação e espaço de memória gasto pelo primeiro kernel corresponde a menos que 5% do total. No entanto, o papel principal deste kernel é reduzir a quantidade de threads que serão executadas nos próximos kernels. De agora em diante, os dois próximos kernels irão executar sobre o número de elementos na lista *visFaces* ao invés da lista *extFaces*. Desta forma, o número de threads nos Kernels 2 e 3 será igual ao número de faces visíveis ao invés do número de faces externas, reduzindo em aproximadamente 50% a quantidade de threads.

Segundo Kernel O segundo kernel executa sobre a lista *visFaces*, lendo as coordenadas dos vértices da *vertList* em memória de textura. Os índices dos vértices para acessar a *vertList* são lidos da *tetList* em memória global. Note que, em CUDA, coordenadas de textura não podem ser acessadas diretamente, forçando a implementação do algoritmo VF-Ray-GPU a guardar a *tetList* em memória global ao invés de memória de textura, evitando assim instruções de *desvios* desnecessárias.

Com as coordenadas dos vértices, o algoritmo VF-Ray-GPU projeta a face visível no plano da imagem. A *caixa limitante* da face projetada, i.e. o menor retângulo que contém a face, é computada e guardada em memória global na lista *projVisFaces*, como ilustrado na Figura 3.2, para ser usada pelo próximo kernel.

O segundo kernel usa uma thread para cada face visível computada. Como o primeiro kernel, o segundo utiliza poucos recursos de um multiprocessador. Portanto, o número de threads por bloco pode ser maximizado e o tempo de computação e espaço de memória são desprezíveis.

Terceiro Kernel No terceiro kernel são lidas duas listas: *visFaces*, computada no primeiro kernel; e *projVisFaces*, computada no segundo kernel. Os pixels de cada face visível, que definem o *conjunto visível*, são determinados usando a caixa limitante da projeção e um teste de ponto no interior de um triângulo. Este teste emprega simples operações de produtos vetoriais. Para cada pixel determinado, o traçado de raios é acionado usando a face visível como face de entrada. Todos os pixels do conjunto visível utilizam os parâmetros guardados na lista *visFaces*. A partir deste ponto, o caminho do raio é descoberto computando as intersecções do raio com cada face interna.

Similarmente ao algoritmo VF-Ray original, as faces internas são guardadas em um buffer chamado *computedFaces*. Entretanto, o algoritmo em GPU é projetado para executar em paralelo aproveitando a coerência entre os raios dentro da computação de cada thread. Portanto, o buffer *computedFaces* é alocado em memória local do CUDA para cada thread com um tamanho fixo, e indexado por uma *tabela de dispersão*. O índice da dispersão utilizado é a coordenada z do centróide da face (*centroidZ*), a ideia pode ser vista na Figura 3.4.

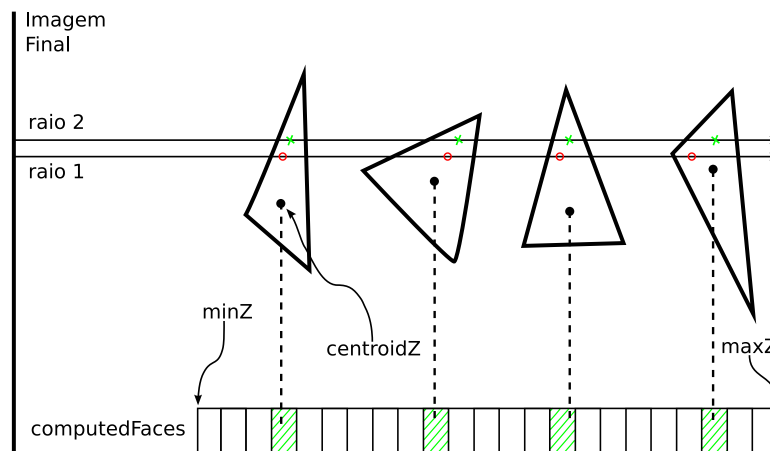


Figura 3.4: O buffer *computedFaces* é usado para guardar faces previamente criadas (círculos vermelhos) para, no futuro, serem lidas (cruzes verdes). O *centroidZ* é usado para dispersar as faces no buffer.

Para realizar a dispersão das faces, a primeira posição do buffer é associada à coordenada z mínima do dado volumétrico (*minZ*), enquanto que a última posição é associada à coordenada z máxima (*maxZ*). Na Figura 3.4, dois raios próximos processados pela mesma thread são apresentados, um após o outro. O raio 1 cria quatro faces internas e o raio 2 apenas lê os dados da face do buffer *computedFaces*. Este buffer é usado para guardar os parâmetros da face e dois índices: t_i do tetraedro e f_i da face. Em contrapartida, o algoritmo VF-Ray original gasta um espaço de memória maior para guardar todos os índices de face para cada tetraedro do volume, na indexação do buffer *computedFaces*.

O terceiro kernel divide a grade CUDA em blocos (veja a Figura 3.5), onde cada bloco é associado a uma face visível. Os blocos são, por sua vez, divididos em threads, onde cada uma computa um pequeno conjunto de pixels. Diferentemente dos dois primeiros kernels, a quantidade de computação executada no caminho do raio usa muitos recursos de um multiprocessador, tornando a computação de todos os pixels do conjunto visível por uma única thread no bloco impraticável.

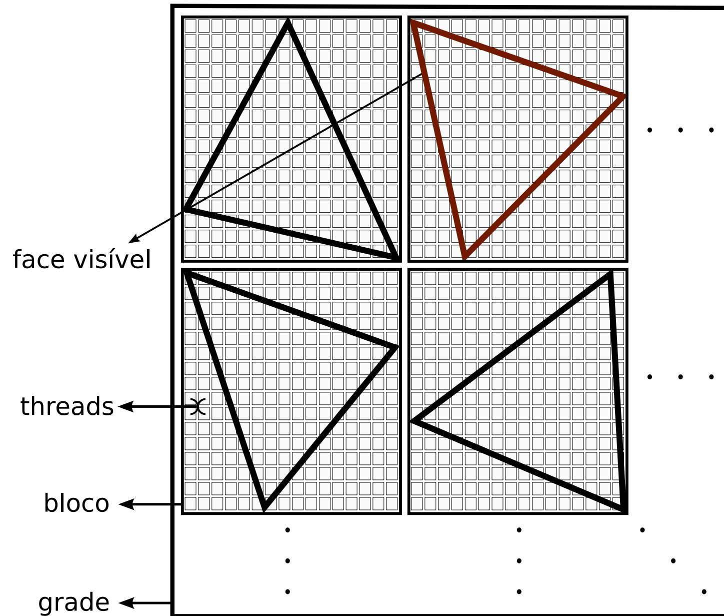


Figura 3.5: O esquema de grade/blocos/threads utilizado pelo terceiro kernel da implementação do algoritmo VF-Ray-GPU. Cada bloco dentro da grade computa uma face visível, e cada thread dentro do bloco computa um pequeno conjunto de pixels. O número de pixels neste conjunto depende da quantidade de pixels da face visível, balanceando número de threads por bloco com o tamanho do conjunto de pixels tratados por thread.

Casos Degenerados Os casos degenerados são tratados de forma diferente do algoritmo VF-Ray original. Enquanto no VF-Ray a lista *Use_set* é pré-computada e consultada toda vez que um raio acerta um vértice ou aresta; no algoritmo em GPU, o raio é perturbado para evitar o caso degenerado, continuando a próxima iteração com o raio na mesma direção sem considerar a perturbação. Os resultados obtidos por esta nova abordagem são similares ao VF-Ray original, porém sem manter a lista *Use_set* reduzindo o consumo de memória.

Código Fonte O código do VF-Ray em CPU e GPU usando C/C++ e *C for CUDA* é disponibilizado com esta tese em:

http://code.google.com/p/vfray.

3.2 RPTINT

O algoritmo *Regular Projected Tetrahedra with Partial Pre-Integration* (RPTINT) foi publicado na conferência internacional *GRAPP* 2007 [54], e é a primeira extensão do algoritmo PTINT apresentada por esta tese. O algoritmo *Projected Tetrahedra with Partial Pre-Integration* (PTINT) foi tema da dissertação de mestrado realizada por mim em 2006 [55], e foi publicado no SIBGRAPI do mesmo ano [23].

A ideia básica desta primeira extensão é usar o algoritmo PT [6] em GPU aproveitando as vantagens da regularidade do volume. Uma grade regular consiste de *voxels* (elementos de volume) regularmente espaçados onde a topologia é implícita. As células de dados regulares são hexaedros, onde cada vértice do hexaedro corresponde a um voxel com seu valor escalar atribuído.

O algoritmo RPTINT é dividido em quatro passos, onde os primeiros três passos são realizados em CPU e o quarto em GPU (veja a Figura 3.6). Primeiro, a projeção de um único hexaedro é computada dividindo-o em cinco tetraedros. Segundo, a ordem do caminho a ser percorrido dentro do volume é determinada. O passo final a ser realizado em CPU consiste em alocar a informação do volume em uma estrutura de dados chamada de *vetor de vértices*. Finalmente, *leques de triângulos* correspondendo aos tetraedros do hexaedro projetado são renderizados em GPU.

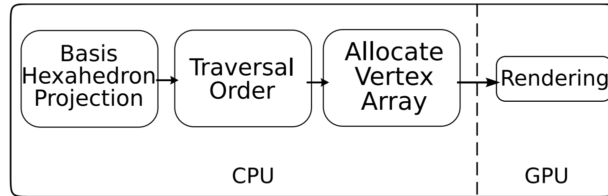


Figura 3.6: Visão geral do algoritmo RPTINT. No primeiro passo o hexaedro base é projetado (*basis hexahedron projection*) dividindo-o em 5 tetraedros e usando o algoritmo PT original. Em seguida, a ordem de renderização das células do volume é determinada (*traversal order*). No terceiro e último passo em CPU o vetor de vértices é alocado (*Allocate Vertex Array*) e usado pelo quarto passo em GPU responsável por renderizar o volume (*Rendering*).

Projeção do Hexaedro Base Dado que o algoritmo PT espera uma malha tetraedral como entrada, o dado volumétrico regular deve ser pré-processado subdividindo cada hexaedro em cinco tetraedros. A este conjunto de tetraedros é dado o nome de *volunit* (*volume unit*) ou *unidade de volume*. A contribuição chave do algoritmo é a ideia que ao renderizar um volume regular, todas as *volunits* são projetadas no plano da imagem exatamente da mesma maneira (considerando uma projeção ortogonal). Logo, o algoritmo RPTINT evita computação redundante calculando os valores resultantes da projeção de um *hexaedro base* apenas uma vez. O algoritmo PT é usado na projeção da volunit relativa ao hexaedro base.

Cada *tetraedro base*, i.e. tetraedro resultante da subdivisão do *hexaedro base*, é projetado no plano da imagem e sua classe de projeção é determinada pelos 4 testes de produto vetorial do algoritmo PTINT (explicado no Apêndice D). Depois de projetado, o vértice espesso e os valores escalares de entrada s_f e saída s_b podem ser computados por intersecção de segmentos.

Para cada um dos 5 *tetraedros base*, os seguintes parâmetros de projeção são computados e guardados:

- *classe base da projeção*;
- coordenadas dos *vértices base* projetados;
- coordenadas do *vértice espesso base*;
- *parâmetros base de intersecção* para computar os valores escalares de entrada e saída;
- *ordem base de renderização.*

Renderização Como no algoritmo PTINT original, a renderização do volume pelo RPTINT é realizada enviando cada tetraedro como um leque de triângulos para a placa gráfica. As primitivas são desenhadas na ordem *de-trás-para-frente* e os fragmentos são compostos usando uma função de mistura. O *hexaedro base* é iterativamente deslocado para a posição de cada *volunit*, e os *vértices base* guardados são usados para compor os triângulos de cada tetraedro. Cada leque de triângulos é renderizado com o número de triângulos relativo à sua *classe base de projeção* seguindo a *ordem base de renderização.* O *vértice espesso base* é utilizado como primeiro vértice do leque, i.e. o vértice do centro do leque de triângulos.

Mesmo que a geometria possa ser resolvida somente deslocando o *hexaedro base*, os valores de cor e opacidade são diferentes para cada vértice das células do volume, e precisam ser computados em cada quadro para cada *volunit.* A cor final somente é computada no shader de fragmento do último passo em GPU. Os valores atribuídos à cor dos vértices são os valores escalares de entrada s_f e saída s_b, e a espessura l.

Para cada vértice fino, os valores escalares de entrada e saída são iguais aos escalares originais do dado volumétrico. Além disso a espessura do vértice fino é zero, uma vez que o raio não atravessa nenhuma distância no tetraedro por estes vértices. Por outro lado, os valores escalares do vértice espesso são calculados usando os *parâmetros base de intersecção*, enquanto que a *espessura base* já foi computada para cada *tetraedro base* e não muda para as volunits do dado. As definições de vértice espesso e fino podem ser conferidas na explicação do algoritmo PT no Apêndice C.

Shaders de Vértice e Fragmento As coordenadas do vértice são computadas pelo RPTINT em cada quadro usando shader de vértice e se baseando nos 3 índices da amostra na grade (i, j e k da grade 3D do dado regular posiciona cada vértice do volume). Esta computação necessita de alguns parâmetros calculados previamente em CPU para o *hexaedro base* e passados para o shader por valores globais, chamados *variáveis uniformes* na linguagem GLSL [2].

Cada vértice da grade é renderizado diversas vezes, uma para cada tetraedro incidente. Logo, junto das coordenadas do vértice na grade, informações adicionais são passadas usando a normal do vértice: o índice local do tetraedro (tet_{id}) e o índice local do vértice ($vert_{id}$). O tet_{id} é guardado na coordenada x da normal e identifica qual dos 5 tetraedros (dentro do hexaedro) está sendo renderizado. O $vert_{id}$ é guardado na coordenada y da normal e identifica o vértice dentro do tetraedro. Note que com este esquema de passagem de informação, a maior parte da carga de computação é transferida da CPU para a GPU.

O shader de fragmento recebe as cores RGB dos vértices (s_f, s_b, l) linearmente interpoladas. Os valores escalares interpolados são usados para consultar os valores de cromaticidade e opacidade na função de transferência. Esta função é guardada em uma textura 1D da mesma forma como no algoritmo PTINT original. Adicionalmente, no intuito de melhorar o desempenho do algoritmo, os tetraedros com todos os vértices definidos com opacidade zero (pela edição da função de transferência) são descartados, ou seja, não são enviados para o pipeline do algoritmo.

Na Figura 3.7 o pipeline do algoritmo RPTINT (passo 4 de renderização) é apresentado. Cada hexaedro do volume é enviado para a GPU como cinco leques de triângulos, onde cada leque corresponde a um tetraedro projetado. O shader de vértice usa as informações do *hexaedro base*, guardadas globalmente como variáveis uniformes, para deslocar corretamente os vértices. O processo de rasterização é responsável por interpolar os valores s_f, s_b e l em cada triângulo.

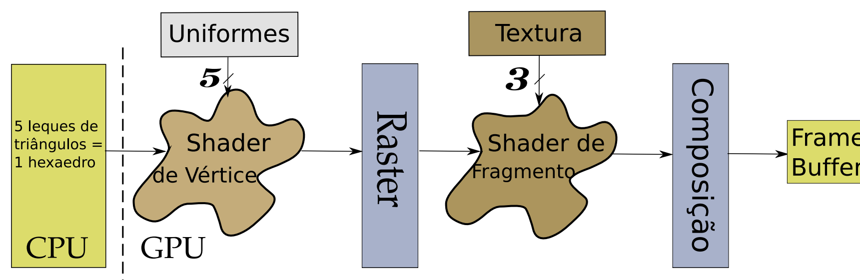


Figura 3.7: Pipeline do algoritmo RPTINT. Os 5 *vértices base* projetados, incluindo o vértice espesso, são lidos de variáveis globais (uniformes) no shader de vértice. Enquanto que no shader de fragmento, são lidas 3 texturas: uma tabela com resultados da função exponencial $f(x) = e^x$ para acelerar a sua avaliação; uma tabela com a função de transferência; e a tabela ψ de pré-integração parcial para melhorar a qualidade de integração original do PT.

No shader de fragmento, os valores interpolados s_f, s_b e l são usados para computar a cor final do fragmento. Os escalares de entrada e saída são transformados em cores pela função de transferência. A função exponencial, em forma de tabela de consulta, é usada para calcular a opacidade do fragmento. As cores de entrada e saída são empregadas na consulta à tabela ψ de pré-integração parcial de MORELAND e ANGEL [11], de forma idêntica ao algoritmo PTINT original. A cor e opacidade final do fragmento é a saída do shader, sendo composta no último estágio do pipeline (veja Figura 3.7) antes de formar a imagem final no frame buffer.

Estruturas de Dados No algoritmo RPTINT não são armazenadas as texturas de vértice e de tetraedros, responsáveis por guardar o volume em GPU no algoritmo PTINT original. No RPTINT, a projeção e classificação dos tetraedros é realizada em CPU (no primeiro passo) ao invés de em GPU como no PTINT original. Com isto, o consumo de memória em GPU é extremamente reduzido, sendo constante e independente do tamanho do volume visualizado.

O volume é descrito através de vetores enviados para GPU no terceiro passo do algoritmo. Três vetores são utilizados: de vértice; de cor; e de normal. Cada vetor com tamanho fixo e contendo os cinco vértices de cada um dos cinco tetraedros (quatro vértices mais o vértice espesso). A Figura 3.8 mostra um exemplo de projeção com a parte associada ao tetraedro projetado em destaque.

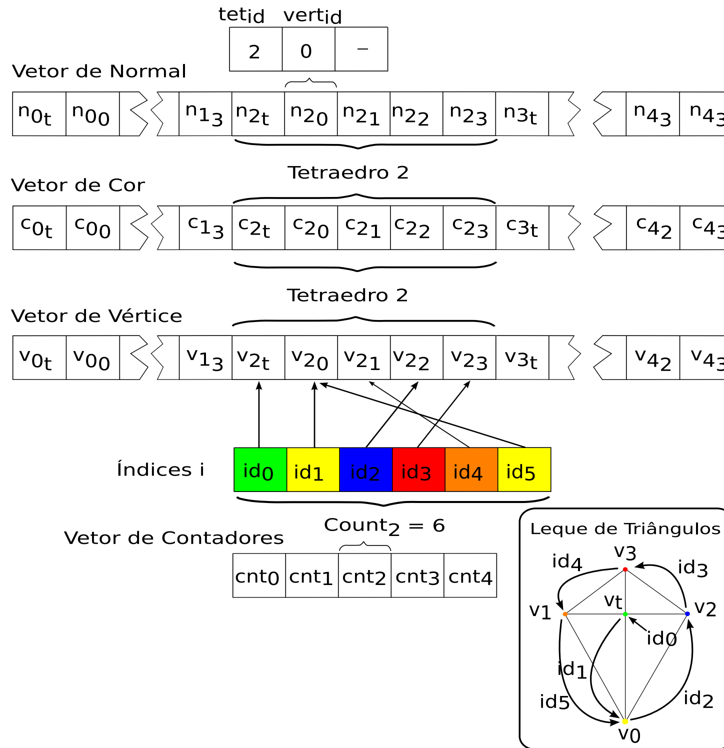


Figura 3.8: Os vetores de vértice, cor e normal formam as estruturas de dados que enviam a informação do volume da CPU para GPU no algoritmo RPTINT.

Adicionalmente, um vetor de contadores é utilizado para determinar a ordem e a quantidade dos vértices no leque de triângulos, como ilustrado na Figura 3.8. O caso do tetraedro exemplificado na figura é da classe 2 de projeção, gerando 4 triângulos no leque para renderização (no canto inferior direito da figura). O tamanho e estrutura dos vetores usados pelo RPTINT não se modificam com o ponto de vista, sendo construídos e armazenados em pré-computação. O terceiro passo do algoritmo fica apenas responsável por atualizar determinados valores, que dependam da forma de projeção, e enviar os vetores pré-armazenados.

Ordenação de Células Uma desvantagem dos algoritmos de projeção de células é a necessidade de ordenação. Ordenar milhões de células é custoso computacionalmente, requerendo estruturas de dados auxiliares e passos de pré-processamento [13–15, 56]. Entretanto, o algoritmo RPTINT evita o custo de ordenar as células se beneficiando do fato que o dado volumétrico é regular e, logo, implicitamente ordenado como um vetor 3D. A única etapa necessária é determinar a ordem do caminho a ser percorrido dentro do dado (segundo passo em CPU), que pode ser feito em tempo constante e desprezível. Na verdade, só existem 8 possíveis ordens para navegar dentro do volume de qualquer ponto de vista.

Seja o *vetor de visão* definido por $\vec{v} = \{v_x, v_y, v_z\}$. Um valor positivo em v_x indica que as *volunits* devem ser percorridas em ordem ascendente no índice x, e um valor negativo em v_x indica ordem descendente. O mesmo raciocínio pode ser usado para os outros eixos. Note que, a ordem relativa em que os eixos são percorridos não importa em dados regulares, isto é, iterar na ordem x, y e z produz o mesmo resultado que iterar em z, y e x, ou qualquer outra permutação.

Interação com o Volume A interação com o dado volumétrico é aprimorada no algoritmo RPTINT acrescentando uma ferramenta de corte, além da ferramenta de edição interativa da função de transferência que aparece no PTINT original [55]. A ferramenta de corte age diretamente no dado volumétrico. Ao selecionar uma área retangular, o volume é cortado e características pequenas são ampliadas. A única limitação é que o corte deve ser feito paralelo a um dos lados da caixa limitante do volume, assegurando que a grade regular resultante seja completa (um cubo ou paralelepípedo).

Código Fonte O código do RPTINT, usando C/C++ e GLSL, é disponibilizado com esta tese em:

http://code.google.com/p/rptint.

3.3 IPTINT

O algoritmo *Improved Projected Tetrahedra with Partial Pre-Integration* (IPTINT) foi publicado na revista *Computer Graphics Forum* em 2008 [57], e é a segunda extensão do algoritmo PTINT apresentada nesta tese. No artigo publicado, o algoritmo PTINT original é a variante do PT em GPU com pré-integração parcial, e o algoritmo IPTINT corresponde a variante com ambas as técnicas de renderização de iso-superfícies e pré-integração parcial.

O algoritmo IPTINT utiliza a mesma ideia do algoritmo PTINT, sendo executado quase que completamente em placa gráfica e dividido em dois passos principais executados em shaders de fragmento. No primeiro passo, todas as informações relevantes de cada projeção de tetraedro são computadas, ou seja, a classe de projeção, as propriedades do vértice espesso, e a coordenada z do centróide do tetraedro. No segundo passo, os escalares dos vértices e os vetores gradiente (da variação escalar) são interpolados na rasterização e usados para computar a cromaticidade e opacidade do fragmento. O escalar interpolado é utilizado para detectar iso-superfícies, enquanto o gradiente interpolado é usado na iluminação da iso-superfície.

Informação do Gradiente Diferentemente do algoritmo PTINT original, o vértice espesso tem a informação de gradiente de entrada (g_f) e saída (g_b). Estes valores são computados no IPTINT da mesma forma que o escalar de entrada (s_f) e saída (s_b) são computados no PTINT original, por interpolação dos gradientes nos vértices. Para computar estes valores, uma terceira textura chamada *Textura de Gradientes* é empregada. Cada texel *RGB* dessa textura guarda as coordenadas x, y e z do vetor gradiente pré-computado por vértice (veja a Figura 3.9).

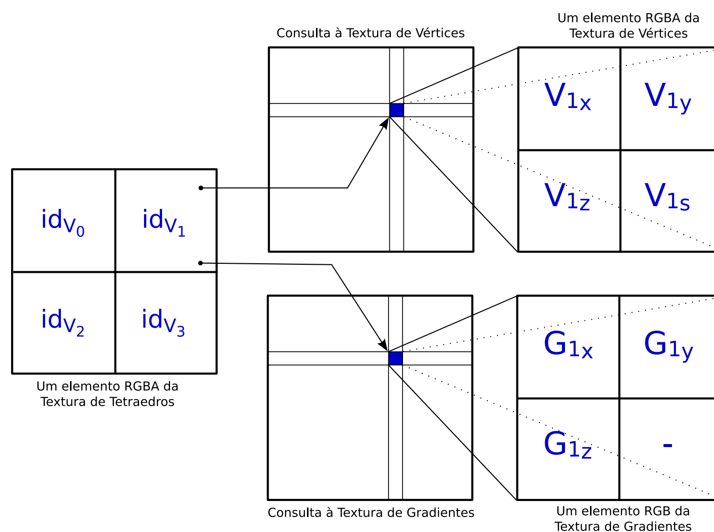


Figura 3.9: Consulta aos dados do vértice no primeiro shader de fragmento. Cada texel da Textura de Tetraedros contém os índices dos seus quatro vértices nas Texturas de Vértice e Gradiente.

Com exceção da leitura e computação do gradiente, o primeiro passo do algoritmo IPTINT é idêntico ao PTINT original. A entrada e saída do primeiro passo é alterada para suportar a nova informação de gradiente, como pode ser visto na Figura 3.10. Note que, para o IPTINT é necessário ler de 4 texturas e renderizar em 4 frame buffers diferentes. Esta renderização é feita usando a técnica chamada de *renderização em múltiplos alvos* – MRT (*Multiple Render Targets*). No caso do PTINT, apenas 3 texturas e 2 frame buffers são empregados (veja Apêndice D). Com isto, dois vetores gradiente por vértice espesso de cada tetraedro são lidos de volta para a CPU.

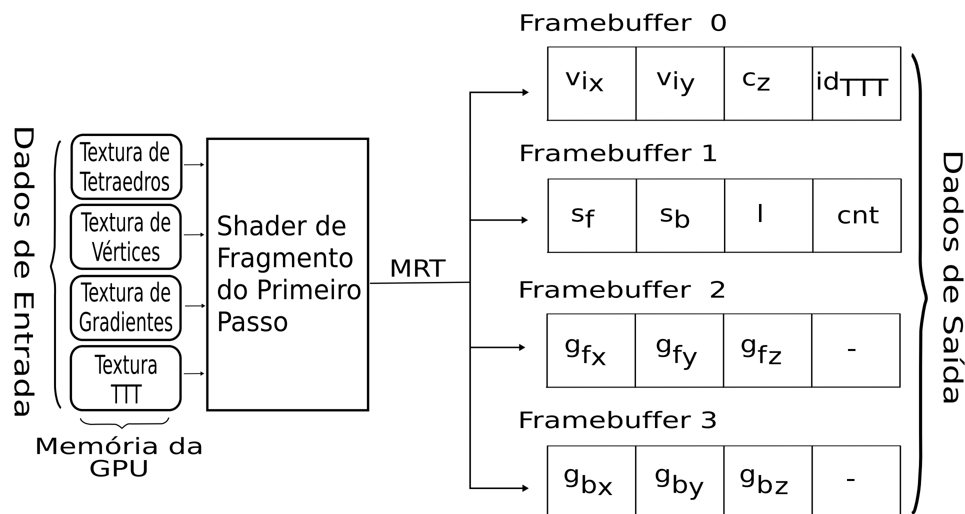


Figura 3.10: Esquema de entrada/saída do primeiro passo do algoritmo IPTINT. A Textura de Gradientes é usada para ler o vetor gradiente de cada vértice no shader de fragmento. Os framebuffers 2 e 3 são usados para passar a informação do gradiente computado de entrada e saída do vértice espesso do primeiro para o segundo passo do algoritmo.

Preparação dos Vetores para Renderização O passo intermediário entre o primeiro e segundo passo em GPU é realizado pela CPU, e consiste de: ordenar as células; e organizar os vetores para renderização. Da mesma forma que o PTINT original, o IPTINT utiliza duas abordagens na ordenação das células: uma simples e inexata ordenação por fatias (*bucket sort*) para quando o volume está sendo rotacionado; e uma ordenação padrão *merge sort* mais custosa para o primeiro quadro do volume parado. A ordenação por fatias divide o volume em intervalos de distância do ponto de vista e os intervalos, ou fatias, são ordenados deixando os tetraedros dentro de cada intervalo sem ordenação. Enquanto que o *merge sort* ordena todos os tetraedros do volume. Ambas as ordenações usam o centróide z do tetraedro (c_z) computado no primeiro passo do algoritmo. O uso do centróide do tetraedro é uma forma aproximada de posicioná-lo no espaço para ordenação.

Na organização dos vetores para renderização, o algoritmo IPTINT emprega quatro estruturas ao invés de duas como no PTINT original. Estas quatro estruturas guardam as coordenadas, valores de cor (atribuindo s_f, s_b e l) e vetores gradiente de todos os vértices do volume (veja a Figura 3.11).

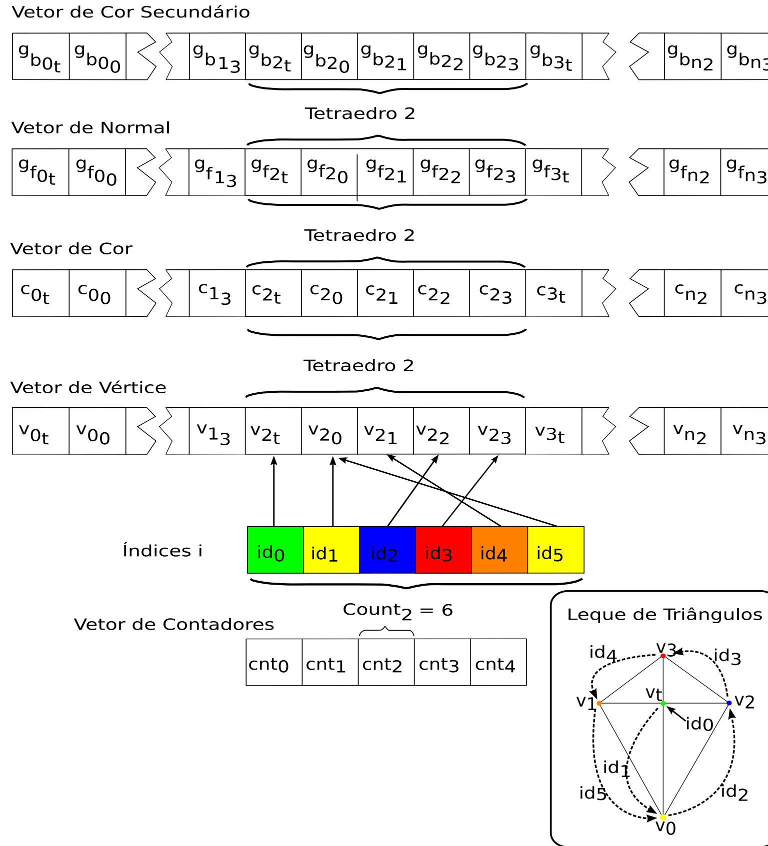


Figura 3.11: Os vetores de vértice, cor e gradientes (como normal e cor secundária) formam as estruturas de dados que enviam a informação do volume da CPU para GPU no algoritmo IPTINT.

As quatro estruturas em forma de vetores contém os tetraedros agrupados em cinco elementos: o vértice espesso e os quatro vértices originais do tetraedro. Uma vez que a mudança do ponto de vista apenas atualiza a posição, cor e gradiente do vértice espesso, a maior parte destes vetores são constantes. Isto possibilita ao OpenGL [1] manter a maior parte da informação do volume em memória da GPU, evitando sobrecarga de transferência de dados CPU–GPU.

O vetor de vértice contém as coordenadas $\{x, y, z\}$ de cada vértice. O vetor de cor contém os valores $\{s_f, s_b, l\}$ ao invés da cor propriamente dita, que será computada no shader de fragmento do segundo passo. Finalmente, os vetores $\{x, y, z\}$ dos gradientes de entrada e saída são armazenados nos vetores de normal e cor secundária, respectivamente. Note que, para um dado vértice fino v_i o $s_f = s_b = s_i$, onde s_i é o valor escalar do vértice v_i, a espessura $l = 0$ e o $g_f = g_b = g_i$, onde o g_i é o vetor gradiente do vértice v_i.

Os quatro vetores são renderizados utilizando a função do OpenGL *glMultiDrawElements*, como nos algoritmos PTINT e RPTINT. Em adição aos quatro vetores com a informação do volume, existem os vetores de índices e de contadores usados para guiar a função *glMultiDrawElements* de renderização. O vetor de índices é dividido em n grupos, onde n é o número de tetraedros. Para cada tetraedro, a ordem correta de renderização do leque de triângulos é guardada como 6 inteiros (grupo destacado chamado *Índices i* na Figura 3.11) que indexam os vértices do tetraedro. O vetor de contadores contém o número de vértices em cada leque. É importante lembrar que o número máximo de vértices em um leque é seis (casos da classe 2) resultando em quatro triângulos. Para casos com menos de 6 vértices, o vetor de índices é acessado somente até a posição cnt_i.

Renderização das Iso-superfícies O segundo passo do algoritmo IPTINT é responsável por computar a cor de cada fragmento. Como no algoritmo original, o IPTINT utiliza os valores interpolados pelo rasterizador da placa gráfica na computação da cor e opacidade para geração da imagem final. Entretanto, os valores interpolados em cada fragmento não são somente $\{s_f, s_b, l\}$, mas também $\{g_{f_x}, g_{f_y}, g_{f_z}\}$ e $\{g_{b_x}, g_{b_y}, g_{b_z}\}$, ou seja, os valores dos atributos de cada vértice (cor, normal e cor secundária).

Além da renderização volumétrica, o algoritmo IPTINT também possibilita a renderização de iso-superfícies. Uma iso-superfície pode ser definida por uma função de segmentação binária $f(s)$ aplicada ao dado volumétrico, onde $f(s)$ retorna 1 se o valor s é considerado parte da superfície e 0 se não for. Quando $f(s)$ é uma função degrau $f(s) = 1, \forall s = s_{iso}$, onde s_{iso} é chamado de *iso-valor*, a região resultante é uma *iso-superfície*. Para o caso de um intervalo $[s_1, s_2]$ em que $f(s) = 1, \forall s \in [s_1, s_2]$, onde $[s_1, s_2]$ é chamado de *iso-intervalo*, a região resultante é uma estrutura chamada de *curvas de nível*.

Um iso-valor é associado a um valor escalar e, para cada fragmento, IPTINT determina se a iso-superfície corta ou não o tetraedro corrente. Dado que cada fragmento contém o valor escalar interpolado de entrada e saída, s_f e s_b respectivamente, a avaliação consiste em testar se o iso-valor está dentro deste intervalo. Caso não esteja, o fragmento é computado utilizando somente a técnica de pré-integração parcial, ou seja, a técnica de visualização volumétrica direta do PTINT original. Por outro lado, caso o iso-valor esteja entre s_f e s_b, a iso-superfície corta o tetraedro corrente e a computação da cor daquele fragmento é realizada considerando o modelo de iluminação de Phong e usando o gradiente interpolado como normal da superfície. A Figura 3.12 mostra um exemplo de iso-superfície encontrada dentro de um tetraedro e os fragmentos resultantes dentro do leque de triângulos daquele tetraedro (exemplo classe 1 de projeção gerando 3 triângulos).

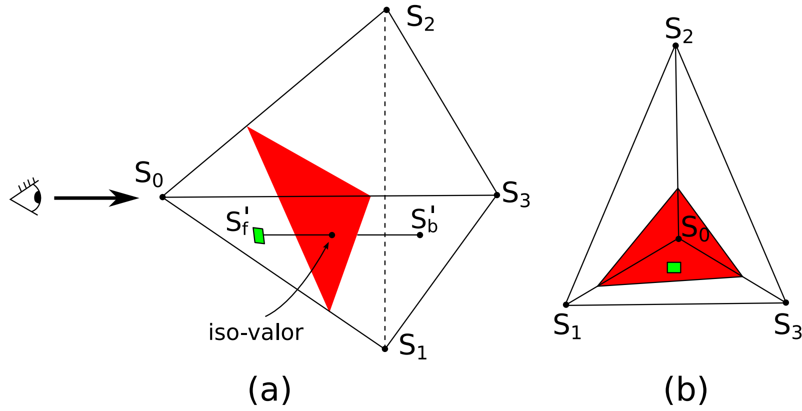


Figura 3.12: Exemplo de uma iso-superfície cortando o tetraedro (a) e o tetraedro projetado com um fragmento (em verde) renderizado dentro da iso-superfície (b). Ambas as técnicas de visualização volumétrica direta e indireta são usadas na avaliação final da cor do fragmento em destaque.

Esta abordagem é similar ao trabalho de Klein *et al.* [58]. Entretanto, no algoritmo IPTINT a iso-superfície não é computada explicitamente, e sim por fragmento aproveitando a vantagem do pipeline de renderização do algoritmo PTINT base. Com esta vantagem, não há necessidade de tratar os diferentes casos em que a iso-superfície corta o tetraedro, como no algoritmo de Pascucci [59].

As iso-superfícies são renderizadas usando o modelo de iluminação de Phong. Os gradientes interpolados atuam como o vetor normal da superfície na computação da luz difusa e reflexão especular. Os gradientes de entrada e saída são combinados linearmente por um peso relativo à distância da iso-superfície para o ponto de entrada e saída do raio no tetraedro. Isto significa que, se a iso-superfície estiver mais próxima do ponto de saída do raio, por exemplo, o gradiente de saída terá um peso maior na avaliação do vetor normal da superfície, do que o gradiente de entrada.

Esta técnica permite uma visualização híbrida do volume (direta e indireta) com múltiplas iso-superfícies. As iso-superfícies podem ser renderizadas completamente opacas (tornando a visualização estritamente indireta) ou semi-transparentes. E, dado que toda a computação de iso-superfície é realizada durante o segundo shader de fragmento, o algoritmo IPTINT não necessita empregar um passo adicional de extração em CPU ou GPU.

Código Fonte O código do PTINT e do IPTINT (com a tabela verdade ternária TTT apresentada no Apêndice D), usando C/C++ e GLSL, é disponibilizado com esta tese em:

http://code.google.com/p/ptint.

3.4 HAPT

O algoritmo *Hardware-Assisted Projected Tetrahedra* (HAPT) foi recentemente aceito no simpósio internacional *EuroVis* 2010, para ser publicado em uma edição especial da revista *Computer Graphics Forum* [60].

O algoritmo HAPT é baseado no algoritmo PT, explicado no Apêndice C, e foi desenvolvido para explorar completamente os recursos da placa gráfica. Diferentemente do PTINT, e das duas extensões apresentadas anteriormente (RPTINT e IPTINT), o HAPT usufrui do shader de geometria, além do shader de vértice e fragmento, adaptando a ideia de projeção de tetraedros de forma mais eficiente e próxima do algoritmo PT original. O problema de ordenação de células é tratado pelo HAPT usando um algoritmo de ordenação rápida, *quicksort* em GPU de CEDERMAN e TSIGAS [61], adaptado para o nosso cenário e implementado em CUDA [4]. Uma vez que as células estão ordenadas, HAPT realiza o algoritmo PT em um único passo, aproveitando os três tipos de shaders e os processadores de rasterização de triângulos da GPU. Desta maneira, HAPT tem quatro principais características: primeiro, o algoritmo efetua visualização volumétrica de estruturas irregulares em um único passo de renderização após ordenação; segundo, ele praticamente não consome memória da GPU uma vez que os tetraedros são enviados por fluxo para a placa gráfica; terceiro, em adição a visualização volumétrica direta, iso-superfícies podem ser extraídas e renderizadas interativamente, e dados que variam no tempo (*Dados 4D*) são facilmente tratados; por último, a sequência de tarefas (ou *framework*) realizada pelo HAPT possibilita a troca trivial de seus módulos, como ordenação, fora do pipeline de renderização, ou método de integração, dentro do pipeline, provendo grande flexibilidade de implementação.

Framework do Algoritmo O framework do HAPT é apresentado na Figura 3.13. Primeiro a ordem de visibilidade das células é computada usando um método de ordenação em CPU ou GPU. Os tetraedros ordenados são enviados para o pipeline gráfico através do shader de vértice (*Vertex Shader* – VS), e decompostos em triângulos no shader de geometria (*Geometry Shader* – GS). Os triângulos descem no pipeline com valores escalares e espessura como atributos de cor para a técnica de visualização volumétrica direta (*Direct Volume Rendering* – DVR), e vetores normais para a técnica de renderização de iso-superfícies (*Iso-surface Rendering* – ISO). Este processo traz um grande benefício altamente desejável para uma abordagem baseada em GPU: adaptar apropriadamente uma primitiva volumétrica (tetraedros) à primitivas bem suportadas por placas gráficas (triângulos). Finalmente, HAPT usa um shader de fragmento (*Fragment Shader* – FS) para computar a integral de iluminação dentro do volume e gerar a imagem final.

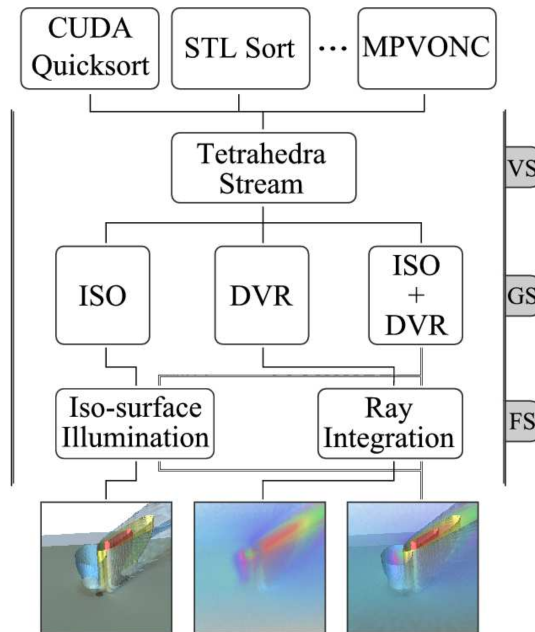


Figura 3.13: Framework do HAPT dividido em shaders de vértice (VS), geometria (GS) e fragmento (FS). A visualização pode ser direta (DVR – *direct volume rendering*) ou indireta por iso-superfícies (ISO). Qualquer método de ordenação pode ser usado antes da renderização, como por exemplo: *quicksort* [61] em CUDA; *STL-based introsort* [62] na CPU; ou o algoritmo MPVONC [56] na CPU.

Pipeline do Algoritmo O pipeline de renderização é apresentado na Figura 3.14. Um importante ponto sobre o fluxo de dados do HAPT é o processamento independente e paralelo dos tetraedros. Além disso, nenhuma estrutura de dados auxiliar é necessária durante a renderização, uma vez que cada primitiva da placa gráfica usada pelo HAPT (pontos, triângulos e fragmentos) contém toda a informação necessária para seu processamento. Isto é especialmente importante porque reduz o armazenamento de dados em GPU, eliminando quaisquer restrições do HAPT no que diz respeito à limitação de memória da GPU ao renderizar um volume. A memória da placa gráfica é limitada quando comparada com a memória da CPU, por exemplo, um volume com alguns milhões de tetraedros pode ser renderizado por uma abordagem de traçado de raios ou projeção de células na CPU, mas falha ao ser armazenado na GPU pelo PTINT [23], IPTINT ou pelo HARC [20].

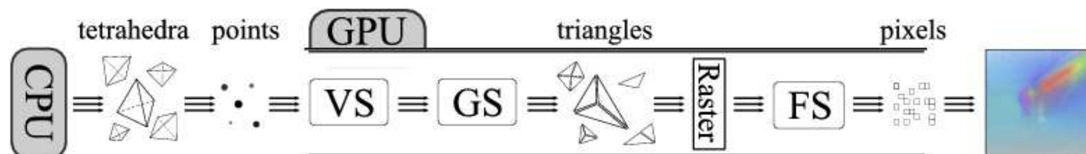


Figura 3.14: Pipeline do HAPT: tetraedros ordenados são enviados como pontos para a GPU; decompostos em triângulos no shader de geometria (GS); e finalmente, durante a rasterização, a integração do raio é computada por fragmento e composta na imagem final.

A característica de renderização por fluxo do HAPT, possibilita-o tratar trivialmente dados que variam no tempo, como uma sequência de volumes estáticos por quadro de renderização. O fluxo de dados do HAPT minimiza a latência de memória, o que usualmente impõe um atraso na transferência de dados e, consequentemente, reduz o desempenho do algoritmo. No restante desta seção cada característica e parte do algoritmo HAPT são apresentadas.

Ordenação de Células O algoritmo HAPT foi testado com quatro métodos de ordenação diferentes para comparação: o *introsort* do *STL* [62] na CPU; o MPVONC [56], *Meshed Polyhedra Visibility Ordering for Non-Convex meshes*, na CPU; o *bitonic sort* na GPU usando CUDA [63]; e o *quicksort* na GPU usando CUDA [61]. Exceto para o MPVONC, as outras três estratégias realizam ordenação aproximada usando os centróides dos tetraedros.

Para os algoritmos usando CUDA, os índices ordenados dos tetraedros são escritos diretamente em um buffer de dados (ou especificamente em um *vertex buffer object* – VBO) da GPU, evitando transferir os índices de volta para a CPU. No caso dos dados não caberem em um VBO, qualquer um dos métodos de ordenação pode ler de volta para a CPU e enviar os tetraedros por fluxo para o pipeline de renderização. Além disso, o método de ordenação *bitonic* em CUDA trata apenas vetores com potência de dois de tamanho, obrigando aumentar o buffer de índices para o tamanho correspondente permitido, e tornando o algoritmo mais lento.

Para o algoritmo de ordenação exata na CPU, i.e. MPVONC, é importante notar que duas estruturas de dados auxiliares são necessárias: a lista de adjacências dos tetraedros e as normais pré-computadas das faces. Esta informação adicional aumenta o consumo de memória na CPU em uma proporção de quatro vezes o espaço gasto para armazenar o volume.

Existem alguns casos degenerados que o método MPVONC não computa a ordenação de células corretamente, contudo, ele funciona para a maioria das malhas [22]. Por outro lado, a ordenação por centróide usualmente introduz um erro, onde em alguns casos a ordem de tetraedros adjacentes pode ser invertida. Felizmente, este erro é muito baixo. Na realidade, não notamos nenhuma diferença visual das ordenações por centróide para o MPVONC, quanto mais quaisquer artefatos.

Para suportar esta última afirmação, nós rodamos uma série de testes para estimar o erro de ordenação por centróide tomando como base o MPVONC. A Tabela 3.1 apresenta o erro máximo e médio para seis volumes diferentes, descritos no Capítulo 5. Os erros são computados por canal de cor separadamente, logo a segunda coluna (Erro Máximo) é a diferença máxima entre todos os canais de todos os pixels correspondentes. A última coluna (Pixels Diferentes) fornece a porcentagem de pixels diferentes, isto é, pixels entre imagens que não são uma correspondência

exata em todos os três canais de cor (RGB). Para todas as estatísticas, somente pixels com erro acima de zero foram levados em consideração, logo pixels corretos e de fundo não foram incluídos. Um erro de aproximadamente 1.2% é equivalente a uma diferença de 3 unidades em um domínio RGB de [0, 255] para um canal de cor específico de um dado pixel. Usualmente o erro médio é aproximadamente de uma única unidade, sendo imperceptível para efeitos de visualização.

Volume	Erro Máximo	Erro Médio	Pixels Diferentes
blunt	1.961%	0.4069%	6.04%
post	2.353%	0.4245%	33.13%
spx2	1.569%	0.3985%	8.13%
delta	5.098%	0.5895%	14.25%
torso	1.176%	0.3933%	1.51%
fighter	1.569%	0.3943%	2.02%

Tabela 3.1: Erro introduzido pelos algoritmos de ordenação por centróide quando comparados contra o método MPVONC.

As estatísticas da Tabela 3.1 foram conduzidas com visualização volumétrica direta usando ambos os métodos (ordenação por centróide e MPVONC) de ao menos 100 pontos de vista diferentes amostrados sobre uma esfera. É também importante notar que os números podem variar com funções de transferência diferentes, ainda assim, não percebemos nenhuma discrepância dos valores apresentados. Desta maneira, o método de centróide fornece uma boa alternativa para acelerar a renderização e diminuir espaço de memória gasto, ao passo que introduz um baixo erro. E se para dados estáticos o erro já é visualmente imperceptível, mesmo quando existem muitos pixels diferentes, este método é ainda mais adequado para visualização interativa de dados que variam no tempo.

Projeção Para simplificar o fluxo de dados para a GPU, um tetraedro é enviado como um vértice com três atributos, isto é, os outros três vértices do tetraedro são passados como coordenadas de textura. Para cada vértice do tetraedro, o valor escalar atribuído é também passado como a coordenada w. Note que diferentes estratégias, como outras primitivas geométricas, podem também ser usadas para enviar os tetraedros pelo pipeline de renderização.

No shader de geometria, o tetraedro é decomposto em triângulos e os três valores associados com cada vértice, a espessura l e o escalar da frente s_f e de trás s_b, são computados usando o mesmo esquema do algoritmo PT original (veja o Apêndice C para detalhes). Para computar a projeção do tetraedro corretamente, o HAPT utiliza a tabela verdade ternária da estratégia de classificação do PTINT (veja o Apêndice D para detalhes), determinando o caso correto de projeção com quatro produtos vetoriais e uma consulta a tabela de classificação.

Estes três valores (s_f, s_b e l) são computados dependendo do vértice projetado. Existem dois tipos de vértice projetado: vértice espesso e fino. O vértice espesso é definido como o ponto de entrada do tetraedro onde o raio atravessa a distância máxima l. Dependendo do caso de classificação, o vértice espesso pode não ser um dos quatro vértices do tetraedro e, neste caso, o seu valor escalar deve ser interpolado. Analogamente, pode ser necessário computar a distância l percorrida nos casos em que l não é o comprimento de uma das arestas do tetraedro. Excluindo o vértice espesso, todos os outros são os vértices do tetraedro projetado, nomeados vértices finos, e tendo $s_f = s_b$, que são os valores escalar originais, e $l = 0$, já que o raio não atravessa nenhuma distância nestas extremidades. No exemplo apresentado na Figura 3.15, v'_0, v'_1 e v'_2 são vértices finos, enquanto v'_3 é o vértice espesso. A tabela de classificação fornece não somente o número de triângulos que são gerados, mas também a ordem em que os vértices devem ser percorridos na geração dos triângulos, e os casos onde a distância l atravessada deve ser computada, e os valores escalares interpolados, para o vértice espesso.

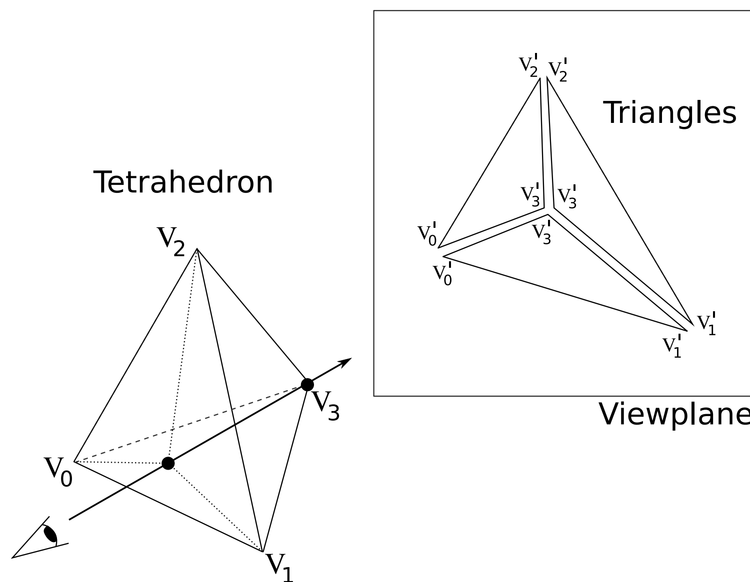


Figura 3.15: Exemplo de classe 1 de projeção do algoritmo PT original, onde a projeção do tetraedro (esquerda) no plano de visão (*viewplane*) gera três triângulos para renderização.

Os valores escalares de entrada e saída (s_f e s_b) e a distância atravessada (l), ou espessura, são armazenados como valores de cor RGB para cada vértice de todos os tetraedros na saída do shader de geometria, seja vértice espesso ou fino. Esta técnica é similar a usada no PTINT e IPTINT, porém no algoritmo HAPT ela serve para passar as informações dos triângulos gerados pelo shader de geometria para o shader de fragmento.

Integração do Raio Quando um triângulo é rasterizado (uma célula tetraedral pode ser decomposta em 1, 2, 3 ou 4 triângulos), os valores escalares e a espessura são interpolados por fragmento. Esta interpolação é uma aproximação dos valores de integração exatos para cada célula usada na equação da integral de iluminação. Os valores interpolados (s_f, s_b e l) são empregados por fragmento para computar a integração do raio através da técnica de pré-integração parcial, em contraste com o simples esquema de média dos escalares feito pelo PT original [6] e o algoritmo GATOR [8]. Nesta técnica a pré-computação da equação de integração não depende da função de transferência e, logo, pode ser pré-compilada dentro da aplicação. Tanto a implementação do algoritmo HAPT como o PTINT e suas duas variantes (RPTINT e IPTINT) usam a pré-integração parcial pré-compilada em suas aplicações. Dentro do shader de fragmento, uma tabela é acessada por dois índices computados usando a distância atravessada l e as cores da frente e de trás. Por sua vez, estas cores são extraídas prontamente da função de transferência usando os valores s_f e s_b. Note que a pré-integração parcial no shader de fragmento pode ser substituída por um método melhor ou mais rápido de integração.

Renderização de Iso-superfícies O fluxo de dados simples e flexível do HAPT permite efeitos adicionais que não são trivialmente implementáveis com outros métodos. Um exemplo é a renderização de iso-superfícies interativamente. Enquanto a maioria das abordagens (IPTINT incluso) são capazes de renderizar iso-superfícies por pixel (para cada fragmento determinar se a iso-superfícies passa por ali), HAPT possibilita não só essa estratégia, mas também uma abordagem no estilo *marching tetrahedra* [64], onde para cada tetraedro, um ou dois triângulos são gerados correspondendo a iso-superfície que corta aquele tetraedro. Usando o *marching tetrahedra*, normais podem ser computadas para fornecer efeitos de iluminação, diferentemente do IPTINT que usa o gradiente do campo escalar como normal improvisada da iso-superfície. O shader de geometria processa cada tetraedro separadamente e, logo, as normais computadas são restritas por triângulo, resultando em um iluminação plana (conhecida como *flat shading*). Para obter um efeito suave de iluminação de Phong, seria necessário ter a informação de adjacência dos vértices e computar normais por vértice ao invés de por triângulo, causando grande impacto no consumo de memória e desempenho de renderização.

Dentro do shader de geometria, a iso-superfície pode ser facilmente extraída do tetraedro e enviada para renderização como um ou dois triângulos adicionais aos triângulos resultantes da projeção do tetraedro. Este método tem duas vantagens principais: primeiro, a iso-superfície pode ser definida interativamente; e segundo, é possível misturar visualização volumétrica direta e indireta no mesmo pipeline, gerando uma visualização híbrida dos dados.

Renderização de Dados 4D Para demonstrar a flexibilidade do algoritmo HAPT melhor, nós descrevemos como este pode ser aplicado na visualização de dados que variam no tempo, também chamados de dados 4D (dados 3D ou volumétricos com a dimensão tempo). O dado 4D é uma sequência de quadros de animação, onde cada quadro é um volume estático.

Como mencionado anteriormente, o uso de VBOs é opcional e não é adequado para modelos que não caibam na memória da GPU; logo, para dados 4D, os quadros não são armazenados em memória mas sim enviados por fluxo como volumes estáticos separados.

Adicionalmente, nós aplicamos um teste prévio de descarte de tetraedros no shader de vértice para remover tetraedros vazios, i.e. células com todos os vértices mapeados para opacidade zero na função de transferência (esse método de descarte também é usado no algoritmo RPTINT, explicado na Seção 3.2). Note que o volume deve ter grandes regiões vazias para se beneficiar dos acessos adicionais à textura para descartar os tetraedros no shader de vértice.

Código Fonte O código do HAPT (com todos os métodos de ordenação), usando C/C++, *C for CUDA* e GLSL, é disponibilizado com esta tese em:

http://code.google.com/p/hapt.

Capítulo 4

Processamento de Malhas

"The most beautiful experience we can have is the mysterious – the fundamental emotion which stands at the cradle of true art and true science."
– Albert Einstein

Neste capítulo introduzimos a ideia de processamento de malhas ampliado por similaridade usando exemplares locais, método que chamamos de *SAMPLE – Similarity Augmented Mesh Processing using Local Exemplars*. O método introduzido propaga computação através da malha e é explicado na Seção 4.1. A propagação é feita empregando um espaço de similaridade, onde regiões similares são espacialmente próximas, como ilustrado na Figura 4.1. Nós investigamos duas aplicações onde o SAMPLE pode ser usado: transferência de detalhe e parametrização de superfícies; discutidas na Seção 4.2. Não obstante, qualquer tarefa de processamento de malhas no qual replicação de processamento seja adequada pode empregar o método apresentado nesta tese.

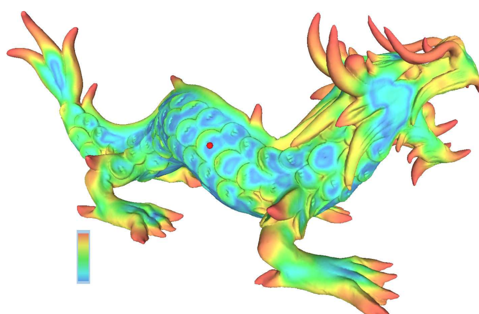


Figura 4.1: Exemplo de espaço de similaridade do SAMPLE. O modelo do XYZ RGB Asian Dragon é pintado por grau de similaridade (escala de cores no canto inferior esquerdo) com o vértice selecionado (círculo vermelho na escama do dragão), de mais similar (em azul) para menos similar (em vermelho).

4.1 SAMPLE

O método *Similarity Augmented Mesh Processing using Local Exemplars* (SAMPLE) foi submetido recentemente para a revista *Graphical Models*, e constitui a contribuição desta tese na área de processamento de malhas.

A ideia principal por trás do SAMPLE é fornecer um método para propagar computações de uma região exemplar para várias outras regiões similares de uma malha. Esta propagação evita computação redundante, permitindo a artistas gráficos usufruir da auto-similaridade de um modelo para reduzir o seu trabalho. Podemos citar como exemplo a pintura automática de regiões semelhantes com um mesmo padrão. O conceito de similaridade neste cenário depende das propriedades da malha utilizada pela tarefa de processamento a ser realizada. Se o processamento leva em consideração as propriedades locais de forma, e.g. normais e curvaturas, o descritor de similaridade deve codificar estas características. Quando propagamos um dado processamento pela malha, o descritor é usado para estabelecer as vizinhanças não-locais onde a computação pode ser replicada pela malha.

O método SAMPLE usa dois espaços para realizar a propagação por similaridade de uma tarefa de processamento de malhas: espaço *primal* e *dual*. O espaço primal define a vizinhança regular de um pedaço da superfície, dado pela conectividade da malha e sua geometria, e o espaço dual define a vizinhança de similaridade, dada pela distância entre descritores de similaridade. O espaço primal é amplamente usado por diversas técnicas de processamento de malhas, como a suavização Laplaciana de TAUBIN [65]. O espaço dual, por outro lado, é pouco usado, sendo normalmente empregado como uma vizinhança não-local para calcular média de valores, como em algumas técnicas de remoção de ruídos [49, 50]. Nós definimos e usamos o espaço dual de uma maneira mais abrangente. No nosso caso, os vizinhos duais definem correspondências entre regiões em adição aos vizinhos primais, auxiliando na replicação do processamento realizado em uma região exemplar para outras regiões próximas no espaço dual (veja um exemplo de espaço dual de similaridade na Figura 4.1).

Uma vez que o espaço primal e dual estão estabelecidos, a propagação do processamento de malhas pode ser realizada. Nós ilustramos nossa ideia aplicando o algoritmo de propagação na transferência de um mesmo detalhe de superfície para diversas regiões similares consistentemente, e usando a parametrização computada para um parte da malha para outras partes similares evitando redundância. Ambas as aplicações são bem adaptadas para propagação usando descritores de auto-similaridade baseados nas características de forma local, como explicado em detalhes na próxima seção. Entretanto, outras aplicações de processamento de malhas poderiam utilizar descritores diferentes, e.g. simetria planar refletiva [25] ou um descritor baseado em saliência [66].

Similaridade baseada em Vértices A vizinhança dual é estabelecida considerando a auto-similaridade de uma malha triangular, que por sua vez é definida pelas distâncias entre os descritores de forma dos vértices que constituem a malha. Estes descritores devem funcionar harmoniosamente com a tarefa de processamento de malhas considerada. No nosso cenário, escolhemos um descritor simples de similaridade baseado em medidas de distância Euclideana. Cada vértice tem um descritor deste tipo, o que permite o posicionamento do vértice no espaço dual. O descritor de similaridade no SAMPLE é um mapa de alturas computado usando a placa gráfica (através da técnica *Z-buffer*) ou lançando raios do plano tangente à superfície (veja um exemplo na Figura 4.2). O plano tangente que suporta o mapa é delineado pela posição e normal do vértice. O mapa de alturas é definido como uma grade regular sobre o plano tangente que é alinhada usando as duas direções principais de curvatura no vértice. A estimativa discreta destas propriedades geométricas diferencias não é sempre robusta. Contudo, esta limitação não apresenta impacto significante na eficiência dos descritores, pois o alinhamento dos eixos da grade pode ser descrito por uma simples rotação, e a comparação dos descritores no método SAMPLE é feita de maneira invariante rotacionalmente.

Para computar o mapa de alturas, consideramos uma sub-região quadrada do plano tangente, com lados de tamanho $\epsilon = 2.5\%$ da diagonal da caixa limitante tridimensional da malha. Esta sub-região é dividida em uma grade 16×16, guiando o lançamento de raios através de cada célula da grade. Nós descobrimos que estes valores (tamanho da grade e taxa de amostra) capturaram bem as propriedades locais de forma em nossos experimentos. A Figura 4.2 ilustra um exemplo de grade de um mapa de alturas na espinha do modelo XYZ RGB Asian Dragon.

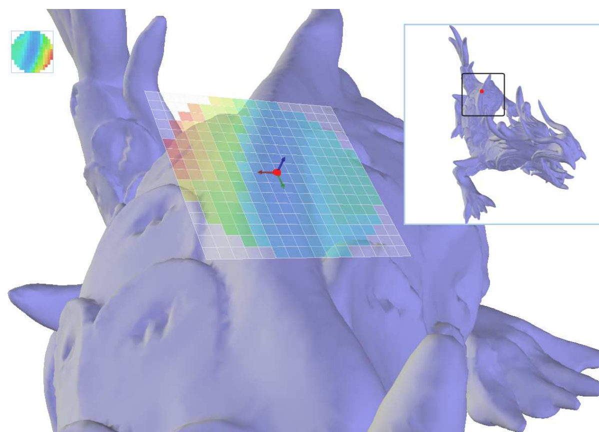


Figura 4.2: Exemplo de descritor de similaridade para um dado vértice (em vermelho). O descritor é um mapa de alturas (canto superior esquerdo) sobre o plano tangente do vértice. Os eixos vermelho e verde são as direções principais de curvatura no vértice, e o eixo azul é a normal do vértice. Os valores de altura são as distâncias Euclideanas do plano à malha, onde as cores azuis e vermelhas representam valores baixos e altos, respectivamente.

Para cada célula da grade, nós lançamos um raio perpendicular ao plano tangente em ambas as direções, encontrando o ponto de intersecção mais próximo da superfície e computando a distância do plano tangente para este ponto. A distância computada passa a ser o valor de altura daquela célula. As células do mapa de alturas sem um valor de intersecção e células fora de um raio ϵ do vértice são descartadas. O resultado é uma imagem circular de resolução 16×16 (veja um exemplo no canto superior esquerdo da Figura 4.2) descrevendo a forma da superfície ao redor do vértice. Esta abordagem é robusta em casos de malhas com triângulos malformados, sem variedade topológica e contendo buracos, uma vez que é baseada apenas na computação de intersecções raio-triângulo. Uma abordagem mais complexa usando propriedades geométrica diferenciais e discretas, tais como o HKS de SUN *et al.* [29], pode resultar em uma assinatura mais descritiva, porém nossos experimentos indicam que tais abordagens são geralmente sensíveis à qualidade da triangulação e singularidades topológicas.

Tendo o mapa de alturas de cada vértice, nós podemos medir a similaridade entre dois vértices quaisquer da malha computando a diferença por pixel das duas imagens relativas aos seus mapas de alturas. Infelizmente, isto leva a resultados pobres. O principal problema é que as direções principais de curvatura de cada vértice não alinham sempre o seu mapa de alturas apropriadamente (veja a Figura 4.3). Nós atacamos este problema usando uma função base que tem se provado útil em prover invariância rotacional no campo de visão computacional; as funções polinomiais ortogonais de Zernike.

A abordagem direta para o problema de alinhamento é simplesmente computar a diferença para toda rotação possível do mapa e escolher aquela com menor valor. Este valor passa a ser a medida de similaridade entre dois vértices. Apesar desta abordagem força bruta produzir o resultado correto, ela é muito ineficiente. Ao invés da abordagem força bruta, nós convertemos cada mapa de alturas em um conjunto de coeficientes de Zernike, nomeado assim pelo trabalho de ZERNIKE [67], e comparamos estes coeficientes ao invés dos pixels da imagem. Os polinômios de Zernike constituem uma base ortogonal para funções definidas no disco unitário. Cada polinômio de Zernike, V_p^q, tem uma ordem p associada a ele e uma repetição q, e eles são definidos sobre um domínio $D = \{(p,q) \mid q \in \mathbb{Z}, p \in \mathbb{Z}^{\geq 0}, |q| \leq p, |p-q| \in \mathbb{Z}^{\text{even}}\}$ como segue:

$$V_p^q(\rho, \theta) = R_p^q(\rho) e^{iq\theta} \tag{4.1}$$

onde R_p^q é o polinômio de base radial dado por:

$$R_p^q(\rho) = \sum_{\substack{k=|q| \\ |p-q| \text{even}}}^{p} \frac{(-1)^{\frac{p-k}{2}} \frac{p+k}{2}!}{\frac{p-k}{2}! \frac{k-q}{2}! \frac{k+q}{2}!} \rho^k \tag{4.2}$$

Para obter os coeficientes de Zernike de uma função $f(x, y)$ definida sobre o disco unitário, nós aplicamos a seguinte fórmula:

$$z_p^q(f) = \frac{p+1}{\pi} \iint_{x^2+y^2 \leq 1} (\bar{V_p}^q)(x, y) f(x, y) dx dy \tag{4.3}$$

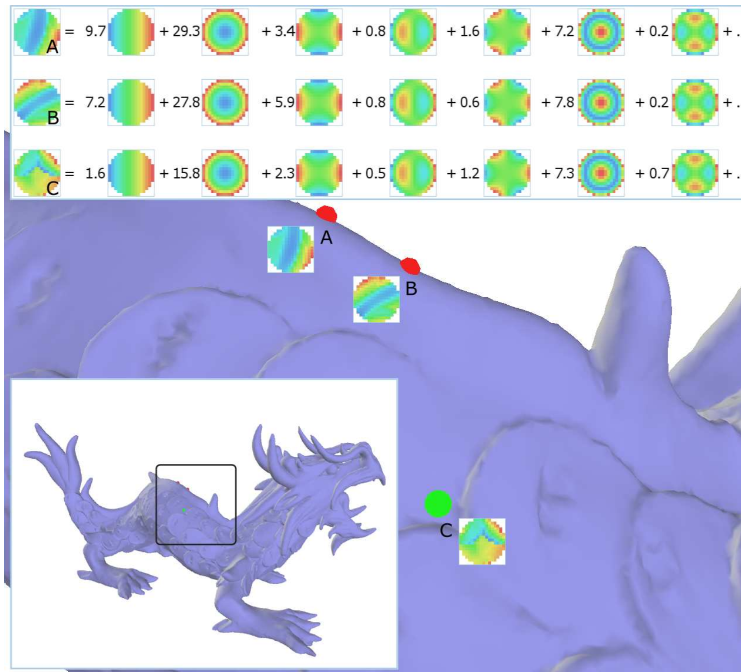


Figura 4.3: Expansão de Zernike para mapa de alturas. Somente as magnitudes dos valores complexos dos coeficientes (números nas três linhas de cima) e os polinômios de Zernike (imagens depois do sinal de igual) são mostradas. Apesar dos vértices A e B (em vermelho) serem similares, suas imagens relativas aos seus mapas de alturas têm alinhamento incorreto baseado nas direções principais de curvatura. Convertendo as imagens para coeficientes de Zernike, os mapas de alturas foram corretamente comparados. O vértice C (em verde) é também corretamente classificado como diferente dos vértices A e B.

Onde $\bar{V_p^q}$ denota o conjugado complexo de V_p^q. Os polinômios de Zernike formam a base na qual a imagem f pode ser projetada. O resultado desta projeção são os momentos de Zernike da imagem, no qual as magnitudes têm a característica de serem invariantes rotacionalmente [68]. Para nossas imagens de mapa de alturas (com resolução 16^2), nós projetamos $f(x, y)$ em um polinômio de Zernike com 25 coeficientes (correspondendo às repetições não-negativas e até a 8ª ordem polinomial) resultando em um vetor $\mathbf{z}_i$ de momentos de Zernike para cada vértice v_i.

Nós notamos que 25 coeficientes são suficientes para representar o descritor de um vértice. A Figura 4.3 mostra um exemplo de duas imagens de mapas de alturas (A e B) com um alinhamento incorreto dado pelas direções principais de curvatura, enquanto que os coeficientes de Zernike das imagens têm valores próximos, classificando corretamente os dois vértices como similares na espinha do modelo XYZ RGB Asian Dragon. O tempo de processamento gasto na computação força bruta é três ordens de magnitude maior que a computação usando coeficientes de Zernike. Mais especificamente, empregar coeficientes de Zernike reduz a medida de similaridade de malhas inteiras de horas para segundos.

Similaridade baseada em Regiões Com as duas técnicas explicadas anteriormente, o método SAMPLE possui uma melhor medida de similaridade entre vértices. O problema remanescente é no espaço dual resultante ao medir similaridades entre vértices isolados. Construir o espaço de similaridade desta maneira pode apresentar discrepâncias em vértices próximos e ignorar propriedades de forma distintas em favor de regiões planas (veja exemplo na esquerda da Figura 4.4). Nós resolvemos este problema considerando vizinhanças inteiras ao invés de vértices isolados quando construímos o espaço dual. Para cada vértice, nós aplicamos pesos Gaussianos aos coeficientes de Zernike, somando os coeficientes de vértices dentro de uma vizinhança de raio fixo ϵ. O corte do filtro Gaussiano (chamado *Gaussian filter cut-off*) é configurado para estar a uma distância de 2σ, onde o parâmetro do filtro σ é dado por $\epsilon/2$ em nosso contexto, para incluir todos os vértices dentro da região considerada pelo mapa de alturas.

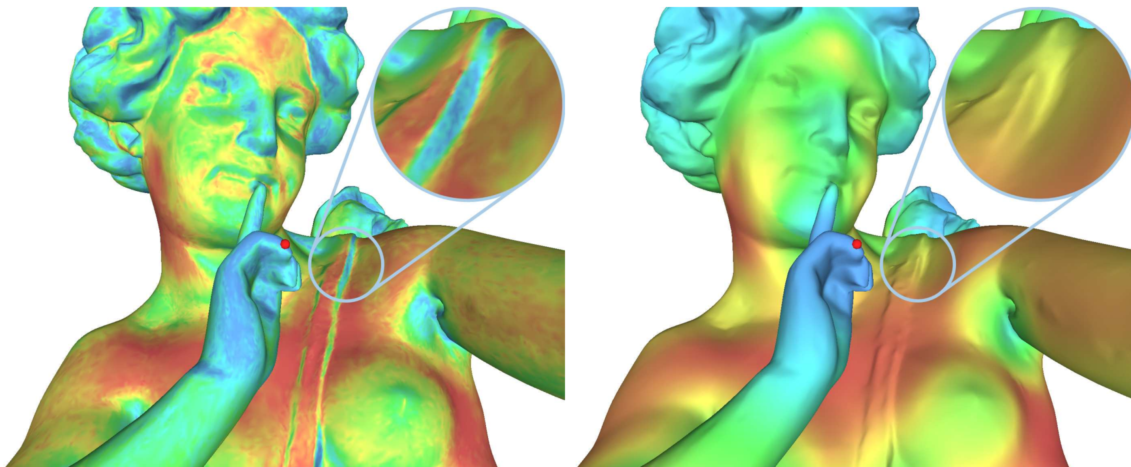


Figura 4.4: Diferença entre os métodos de comparação baseado em vértices (esquerda) e em regiões (direita). A medida de similaridade baseada em vértices isolados produz a comparação incorreta do vértice selecionado (em vermelho) com a região da alça da aljava próxima do ombro (canto superior direito). O modelo é pintado de mais similar (em azul) para menos similar (em vermelho).

Lembre que nós definimos $\mathbf{z}_i$ para ser o vetor de coeficientes de Zernike para o vértice v_i. Nosso método de similaridade baseada em regiões usa a vizinhança no espaço primal de v_i, denotado por $\mathcal{N}(v_i, \sigma)$, que inclui vértices dentro de um distância σ de v_i. Para aplicar nossa comparação usando pesos Gaussianos, nós consideramos um novo vetor de coeficientes de Zernike para cada vértice definido como segue:

$$\mathbf{z}_i^\sigma = \frac{\sum_{v_j \in \mathcal{N}(v_i, 2\sigma)} \mathbf{z}_j e^{-\frac{\left|v_i - v_j\right|^2}{2\sigma^2}}}{\sum_{v_j \in \mathcal{N}(v_i, 2\sigma)} e^{-\frac{\left|v_i - v_j\right|^2}{2\sigma^2}}} \tag{4.4}$$

O $\mathbf{z}_i^\sigma$ é definido como uma média usando pesos Gaussianos dos vetores de coeficientes de Zernike de todos os vértices dentro de um raio 2σ de v_i. Estes vetores recém formados de coeficientes de Zernike com pesos Gaussianos substituem os vetores originais, resultando na geração de um espaço dual suave e de alta qualidade. As Figuras 4.4 e 4.5 ilustram os modelos *3DScanCo Angel* (de uma estátua de anjo) e *Dama* usando os métodos de similaridade baseada em vértices e em regiões.

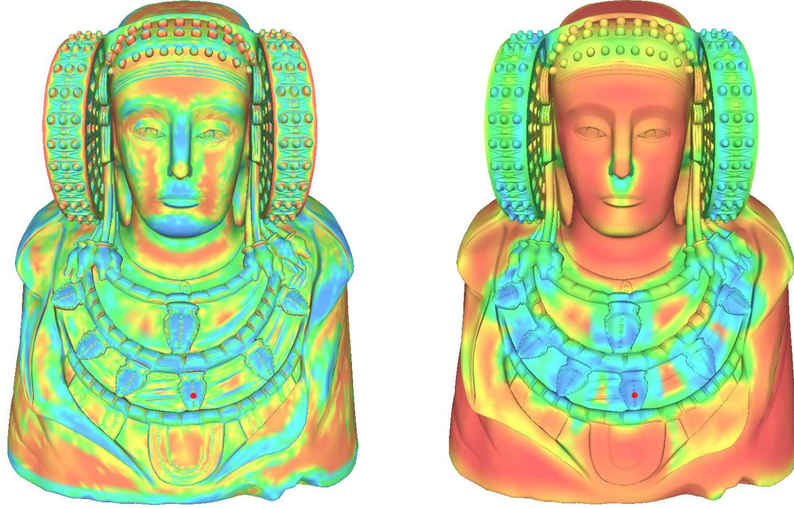


Figura 4.5: O espaço dual do modelo Dama ilustra também a diferença da comparação baseada em vértices (esquerda) e em regiões (direita). Cor azul excessiva na imagem de esquerda mostra que a técnica baseada em vértice não é tão discriminativa como a técnica baseada em regiões na imagem da direita.

Estabelecendo o Espaço Dual Depois de aplicar as técnicas explicadas anteriormente, nós obtemos 25 coeficientes para cada vértice, que constituem as coordenadas de posicionamento no espaço dual. Este espaço é o nosso espaço de similaridade e pode ser simplesmente visto como $\mathbb{R}^{25}$, onde vértices similares podem ser encontrados por vizinhos mais próximos neste espaço, através de uma métrica Euclideana, para qualquer vértice.

A distância de similaridade dada por s no espaço dual entre vértices v_i e v_j é definida da seguinte forma:

$$s(v_i, v_j) = d(\mathbf{z}_i^\sigma, \mathbf{z}_j^\sigma) \tag{4.5}$$

onde $d(\cdot, \cdot)$ denota a métrica de distância Euclideana. O espaço dual de uma malha depende somente dos coeficientes de Zernike com pesos Gaussianos de cada vértice, $\mathbf{z}_i^\sigma$, que podem ser pré-computados e armazenados em uma estrutura de dados auxiliar para medidas de similaridade. Os vizinhos duais de um vértice v_i são definidos como o conjunto de vértices sobre um limite τ onde $s(v_i, v_j) < \tau$. A Figura 4.6 mostra alguns vizinhos no espaço dual de um vértice na escama do XYZ RGB Asian Dragon, os vértices na vizinhança primal são ilustrados apenas para comparação.

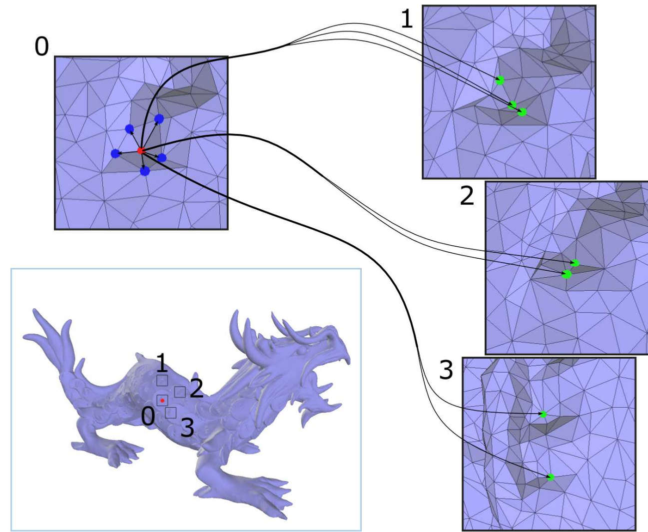


Figura 4.6: Ilustração do espaço primal e dual. Os vizinhos imediatos primais do vértice selecionado (em vermelho) na escama do dragão são mostrados em azul (na caixa 0). Enquanto que alguns dos vizinhos duais são mostrado em verde (nas caixas 1, 2 e 3).

Propagando Processamento de Malhas O algoritmo SAMPLE propaga processamento de malhas usando o espaço dual e um *exemplar local*. Uma região exemplar é qualquer região da malha com propriedades desejadas, ou seja, uma região que será usada para guiar a propagação de uma tarefa de processamento alvo. Por exemplo, para pintar todas as escamas do XYZ RGB Asian Dragon com o mesmo desenho, uma das escamas pode ser selecionada como região exemplar e, ao passo que a pintura ocorre, todas as regiões localmente similares à região exemplar são pintadas com o mesmo padrão (veja a Figura 4.7). A desvantagem desta abordagem é que ela se baseia no tamanho da região exemplar local, que é diretamente

relacionado com o tamanho do descritor por mapa de alturas explicado anteriormente. Se a região exemplar local é muito menor ou maior que o mapa de alturas, o espaço dual falha em capturar as características distintas da forma local, fornecendo incorretamente os vizinhos por similaridade para serem usados na propagação do processamento. Nestes casos, o espaço dual deve ser recomputado considerando um valor ϵ escolhido para coincidir com o tamanho da região desejada e produzir o resultado correto.

O espaço dual desempenha o papel principal na ideia de propagação de processamento de malhas. Ele é responsável por descrever correspondências de similaridade entre vértices, que são usados para propagar computação através de regiões similares. Depois que um exemplar local é escolhido, seus vizinhos duais são determinados empregando uma dada distância limite τ do vértice central do exemplar no espaço dual. Logo, operações realizadas no exemplar são replicadas para outras regiões semelhantes. A distância limite τ de similaridade define quais regiões serão afetadas pela propagação. Valores de limite pequenos afetam poucas regiões, enquanto valores grandes afetam um número maior de regiões.

Embora o método de comparação usado para construir o espaço dual considere regiões na malha, a vizinhança dual é definida entre vértices e não regiões. Os vizinhos duais de um dado vértice podem estar próximos deste vértice no espaço primal. Este cenário compromete a propagação do processamento de malhas. Nós resolvemos este problema considerando somente vértices vizinhos no espaço dual cuja regiões primais não se sobreponham uma com as outras. Isto é, o tamanho da região exemplar define uma distância mínima no espaço primal exigida entre pares de vértices a serem considerados para a propagação do processamento.

Depois destas considerações, nós podemos agora descrever como a propagação de processamento de malhas entre o exemplar local e cada região alvo é realmente executada. Nós definimos um sistema de coordenadas local (chamado de *local coordinate frame – LCF*) para cada vértice sujeito à propagação, a fim de replicar o processamento consistentemente. Este LCF é baseado nos mesmos vetores usados para computar e alinhar o descritor por mapa de alturas no plano tangente do vértice, e ele é usado para atribuir coordenadas locais (u, v, w) para cada vértice dentro das regiões afetadas. Estas coordenadas substituem as coordenadas originais (x, y, z), ao passo que regiões similares são processadas, criando um sistema de coordenadas comum para toda a propagação. O tamanho de cada região afetada é definido pela região exemplar, onde a forma local desejada reside. Considerando que a nossa medida de similaridade é invariante rotacionalmente e os mapas de alturas das regiões afetadas podem não estar alinhados, nós rotacionamos as coordenadas (u, v) por um ângulo α, escolhido para minimizar a diferença de similaridade entre as regiões afetadas e o exemplar. Apesar de existir uma técnica para recuperar o

ângulo de rotação usando momentos de Zernike, técnica de REVAUD *et al.* [69], nós não utilizamos esta técnica, pois, em nossos experimentos, este método produz mínimos locais para o ângulo α indesejáveis, degradando a nossa propagação. Pelo número de regiões que iremos realizar a propagação ser pequeno, nós computamos o ângulo α usando a abordagem força bruta discutida anteriormente, i.e. computamos a diferença entre os mapas de alturas para todas as possíveis rotações e escolhemos aquela que resulta no valor mínimo.

Finalmente, as coordenadas locais são usadas para encontrar os vértices mais próximos na região exemplar de um vértice na região afetada. Estes vértices próximos são usados para interpolar um valor aproximado de qualquer processamento feito no exemplar. A qualidade desta aproximação depende do processamento e condição de amostragem da malha, i.e. uma malha amostrada mais regularmente produz melhores resultados do que uma malha irregularmente amostrada. Na próxima seção, esta ideia é ilustrada usando SAMPLE em duas aplicações.

4.2 Aplicações

Nesta seção serão apresentadas duas aplicações do método SAMPLE para ilustrar o uso do espaço dual e as ideias de propagação. A primeira (veja a Figura 4.7) replica a parametrização computada em uma região exemplar para várias outras regiões similares, preservando a escala e orientação local do domínio de parametrização. A segunda (veja a Figura 4.8) propaga uma dada máscara de detalhe para regiões similares, preservando consistência local.

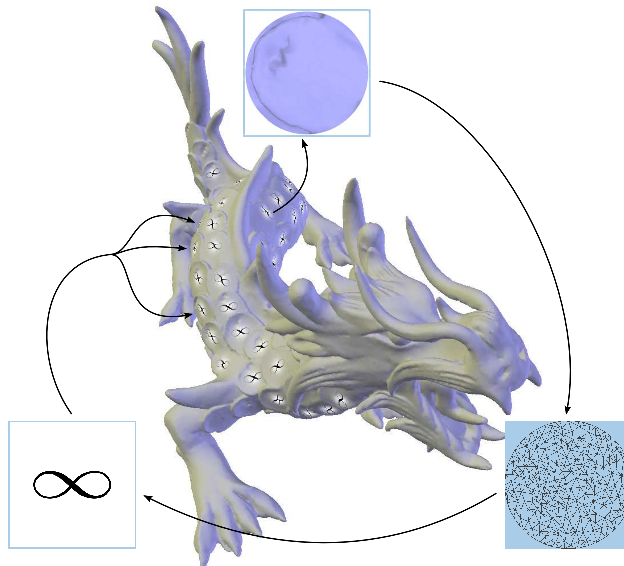


Figura 4.7: Aplicação do SAMPLE: parametrização de superfície. Uma das escamas do dragão é selecionada (topo); o exemplar é parametrizado usando um algoritmo padrão (direita); uma imagem (esquerda) é usada para texturizar escamas similares.

A primeira aplicação visa reutilizar uma parametrização local empregando o espaço dual para replicar os parâmetros computados. Nós realizamos uma simples parametrização, usando o algoritmo conhecido como *Mean Value Coordinates* de FLOATER [70], na região exemplar selecionada, e visitamos um número de regiões similares que residem dentro de uma distância limite no espaço dual. Para cada região similar, valores da parametrização são inferidos da região exemplar usando a coordenada local (u, v, w). Desta maneira, o mesmo domínio de parametrização é compartilhado entre regiões com forma local similar. A Figura 4.7 mostra o processo de replicação usado no modelo XYZ RGB Asian Dragon.

Note que, na Figura 4.7, um pequeno número de escamas não foram afetadas pelo processo. Isto ocorre pois as regiões correspondentes a estas escamas estavam fora de uma distância limite τ especificada da escama exemplar. Nós escolhemos esta distância limite visando selecionar a maior parte das escamas sem produzir falso positivos, i.e. regiões similares que não eram escamas. Estas regiões indesejadas tendem a aparecer em altos valores de τ. Note também que, apesar do procedimento de replicação induzir um erro de posicionamento máximo de 0.047 dentro do disco unitário de parametrização, as texturas não estão visivelmente distorcidas.

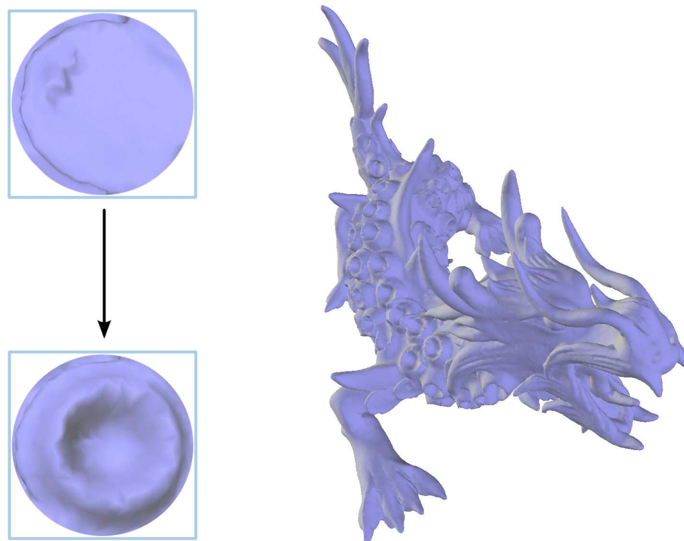


Figura 4.8: Aplicação do SAMPLE: transferência de detalhe. Uma região exemplar (canto superior direito) de uma escama do dragão é editada para parecer uma cratera (canto inferior esquerdo). Nosso algoritmo SAMPLE propaga o resultado da edição naturalmente para todas as escamas similares da malha.

A segunda aplicação visa usar o espaço dual para propagar uma edição na malha para todas as regiões com forma próxima de maneira significativa. Nós usamos uma simples máscara de edição, uma extrusão Gaussiana com um raio fixo (mostrada no canto inferior esquerdo da Figura 4.8), aplicada para a mesma escama do dragão usada na primeira aplicação. Nós também usamos o mesmo limite de similaridade

τ para afetar as escamas idênticas, ilustrando que o nosso algoritmo SAMPLE pode ser empregado de maneira similar para uma grande variedade de tarefas de processamento de malhas. Nesta aplicação, a malha resultante da propagação é equivalente a malha que resultaria se todas as regiões similares ao exemplar local fossem alteradas da mesma forma. Isto ilustra como o sistema SAMPLE pode ser usado para reduzir o trabalho de um artista, que de outra forma teria que editar manualmente cada região similar para obter o mesmo resultado.

Capítulo 5

Resultados

"It is through science that we prove,
but through intuition that we discover."
– Jules Henri Poincaré

Neste capítulo serão apresentados resultados referentes aos algoritmos desta tese. Nos dois capítulos anteriores foram descritos 4 algoritmos aprimorados de visualização volumétrica e introduzido 1 método de processamento de malhas. As 5 técnicas apresentadas foram:

- VF-Ray-GPU – *Visible-Face Driven Ray Casting implemented on the GPU*;
- RPTINT – *Regular Projected Tetrahedra with Partial Pre-Integration*;
- IPTINT – *Improved Projected Tetrahedra with Partial Pre-Integration*;
- HAPT – *Hardware-Assisted Projected Tetrahedra*;
- SAMPLE – *Similarity Augmented Mesh Processing using Local Exemplars.*

A implementação de todas as técnicas foi realizada usando as linguagens C/C++, *C for CUDA* e GLSL; empregando as bibliotecas OpenGL [1], GLUT [71], CGAL [72], *Boost* [73], VCGLib [74] e LCGtk [75]. Medidas de desempenho foram realizadas usando diferentes computadores e placas gráficas, explicitadas em cada uma das técnicas.

Na Seção 5.1 são apresentados os resultados referentes aos algoritmos de visualização volumétrica. E na Seção 5.2 são apresentados os resultados referentes ao método de processamento de malhas.

5.1 Visualização Volumétrica

Os experimentos realizados para os algoritmos de visualização volumétrica apresentados nesta tese utilizaram os seguintes dados volumétricos irregulares: *Blunt Fin* (*blunt*); *Combustion Chamber* (*comb*); *Liquid Oxygen Post* (*post*); *Super Phoenix* incrementado (*spx+*); *Delta Wing* (*delta*); *Human Torso* (*torso*); *F-16 Jet Simulation* (*f16*); *Langley Fighter* normal (*fighter*) e incrementado (*fighter+*). Além destes dados também foram utilizados os seguintes dados volumétricos regulares: *Fuel Injection* (*fuel*); *Electron Distribution Probability* (*neghip*); *Tooth CAT scan* (*tooth*); *Foot X-Ray* (*foot*); *Skull CAT scan* (*skull*); e *Aneurism X-Ray* (*aneurism*). Por último utilizamos um dado volumétrico 4D (que varia no tempo) chamado: *time-varying Turbulent Jet* (*turbjet*). Os volumes utilizados nesta tese foram adquiridos de diferentes fontes, entretanto a maior parte está disponível em:

http://www.volvis.org.

A validação dos algoritmos apresentados foi realizada comparando-os com os seguintes algoritmos do estado da arte:

- VF-Ray – Algoritmo original *Visible-Face Driven Ray Casting* [51];
- PTINT – Algoritmo original *Projected Tetrahedra with Partial Pre-Integration* [23];
- GATOR – *GPU-Accelerated Tetrahedra Renderer* [8];
- VICP (GPU) (CPU) – *View-Independent Cell Projection* (implementado em GPU e CPU) [16];
- VICP (Balanced) – *VICP* com balanceamento GPU-CPU [11];
- HARC – *Hardware-Based Ray Casting* [20];
- HARC (INT) – *HARC* com pré-integração parcial [21];
- HAVIS – *Hardware-Accelerated Volume and Isosurface Rendering Based on Cell-Projection* [7];
- HAVS - *Hardware-Assisted Visibility Sorting* [17].

Os algoritmos utilizados para comparação foram implementados ou adquiridos de seus respectivos autores (veja o Capítulo 2 para uma explicação geral destes algoritmos). Esta tese utiliza um ou mais algoritmos listados dependendo da técnica a ser comparada. No final desta seção, uma comparação direta entre as técnicas apresentadas no Capítulo 3 é delineada.

VF-Ray-GPU Os experimentos do VF-Ray-GPU foram conduzidos em um Intel Pentium Core 2 Duo 6400 com 2.13 GHz por processador e 2 GB RAM. A placa gráfica utilizada foi uma nVidia GeForce 8800 Ultra com 768 MB de memória.

A validação do nosso algoritmo foi realizada contra algoritmos de traçado de raios recentes em GPU. O VF-Ray-GPU foi também comparado com a versão original do VF-Ray em CPU com o intuito de: primeiro, examinar o impacto no desempenho do algoritmo apresentado usando a GPU; e segundo, analisar a diferença no consumo de memória ao utilizar as estruturas de dados adaptadas para a GPU apresentadas nesta tese na Seção 3.1.

Os volumes utilizados nos experimentos foram: "blunt", "post", "fighter+" e o "f16". Na Tabela 5.1 são mostradas as seguintes propriedades dos dados volumétricos testados: o número de vértices (*# Verts*), faces, faces externas e tetraedros (*# Tet*); onde K e M significam mil e milhão, respectivamente. Como pode ser observado nesta tabela, foram utilizados dois dados menores, "blunt" e "post", e dois dados maiores, "fighter+" e "f16". A resolução da imagem final gerada para os volumes menores foi de 512×512 e para os maiores 2048×2048.

Volume	# Verts	# Faces	# Externas	# Tet
blunt	41 K	381 K	13 K	187 K
post	109 K	1 M	27 K	513 K
f16	1.1 M	12.9 M	309 K	6.3 M
fighter+	1.9 M	22.1 M	334 K	11.2 M

Tabela 5.1: Propriedades dos volumes testados.

Os seguintes algoritmos foram usados para comparação: VICP de WEILER *et al.* [16] é um algoritmo híbrido que projeta cada célula do volume, realizando a integração dentro de cada célula usando traçado de raios; HARC de WEILER *et al.* [20] é baseado no algoritmo de BUNYK *et al.* [10] com o adicional de guardar a informação de faces e adjacência em GPU e computar a contribuição de cada célula acessando uma textura 3D com valores de pré-integração; HARC (INT) de ESPINHA e CELES [21] é uma extensão do algoritmo HARC com pré-integração parcial que emprega estruturas de dados alternativas para reduzir o consumo de memória; VF-Ray de RIBEIRO *et al.* [51] é a versão original do algoritmo apresentado nesta tese em CPU, explicada no Apêndice B.

Na Tabela 5.2, o algoritmo apresentado VF-Ray-GPU é comparado com outros algoritmos em placa gráfica, usando os seguintes aspectos de memória: número de bytes por tetraedro (*Bytes/Tet*); número de bytes por pixel (*Bytes/Pixel*); e o total de megabytes gasto em técnicas de pré-integração normal ou parcial (*Pre-Int*). Estes números indicam o requerimento de memória de cada algoritmo, dado o tamanho do volume visualizado e a precisão da imagem gerada.

Algoritmo	*Bytes/Tet*	*Bytes/Pixel*	*Pre-Int*
VICP	456	–	16
HARC	160	96	16
HARC (INT)	96	96	1
VF-Ray-GPU	38	–	–

Tabela 5.2: Aspectos de memória do algoritmo VF-Ray-GPU e outros algoritmos recentes em placa gráfica.

Na Tabela 5.3 são mostrados os resultados de tempo e memória para os algoritmos VICP, HARC, HARC (INT) e VF-Ray-GPU, para o dois dados menores. Os algoritmos em GPU comparados falharam ao visualizar os dados maiores, devido a limitação de memória, impedindo a comparação deles com o VF-Ray-GPU para estes dados. Duas colunas para cada volume resumem o espaço de memória usado (em kilobytes) e o tempo gasto (em milisegundos) na renderização de um quadro (considerando uma janela de visualização de 512×512).

Algoritmo	blunt		post	
	Memória (KB)	*Tempo (ms)*	*Memória (KB)*	*Tempo (ms)*
VICP	118,524	190	249,928	546
HARC	72,267	18	123,245	33
HARC (INT)	22,636	32	50,248	51
VF-Ray-GPU	7,029	186	19,494	370

Tabela 5.3: Comparação do algoritmo VF-Ray-GPU e outros algoritmos recentes em placa gráfica.

Na Tabela 5.4, o algoritmo VF-Ray-GPU e VF-Ray original em CPU são comparados. Nesta tabela são mostrados espaço de memória (em kilobytes) e tempo de execução total (em milisegundos), para os dois dados volumétricos maiores. Como pode ser observado na tabela, o algoritmo apresentado nesta tese executa por volta de 7 vezes mais rápido que o original, e usa menos de 50% da memória. O ganho de desempenho é devido a natureza paralela do algoritmo de traçado de raios, enquanto as estruturas de dados apresentadas são responsáveis pela redução do espaço de memória gasto em placa gráfica.

Volume	Memória (MB)		Tempo (ms)	
	CPU	GPU	CPU	GPU
fighter+	876	426	58326	10035
f16	499	239	51235	8815

Tabela 5.4: Comparação entre o algoritmo VF-Ray original em CPU e o algoritmo apresentado nesta tese VF-Ray-GPU.

Na Figura 5.1 são apresentados resultados de renderização do nosso algoritmo. Os quatro volumes utilizados para teste foram renderizados.

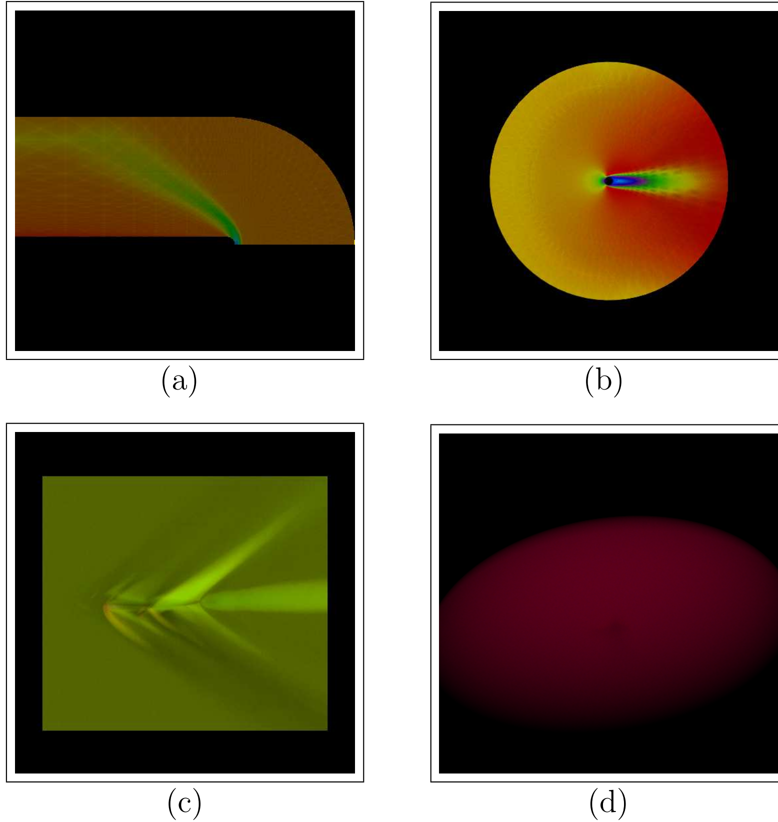


Figura 5.1: Imagens geradas pelo algoritmo VF-Ray-GPU dos seguintes volumes testados: “blunt” (a), “post” (b), “fighter+” (c) e “f16” (d).

RPTINT O desempenho do algoritmo RPTINT foi medido em um Intel Pentium IV 3.6 GHz, 2 GB RAM, com uma placa gráfica nVidia GeForce 6800 256 MB conectada em um barramento PCI Express 16x.

Os resultados dos dados volumétricos regulares usados para teste são mostrados na Tabela 5.5. As medidas de desempenho foram feitas usando uma janela de 512×512 pixels com o volume em constante rotação. Este procedimento de rotação é importante para mudar as classes de projeção, uma vez que renderizações paralelas geram um número menor de triângulos.

Volume	# Verts	# Tet	*fps*	*M Tet/s*
fuel	262 K	1.2 M	70.78	88.5 (5.99)
tooth−	1 M	5 M	1.22	13.1 (6.30)
tooth	10 M	52 M	0.24	12.7 (6.61)
foot	16 M	83 M	0.81	67.7 (6.23)
skull	16 M	83 M	0.61	51.7 (6.25)
aneurism	16 M	83 M	2.42	201 (5.35)

Tabela 5.5: Medida de desempenho do algoritmo RPTINT para diferentes dados volumétricos regulares. O resultado “tooth−” corresponde à tomografia original do dente “tooth” cortada com a nossa ferramenta de corte (discutida na Seção 3.2).

Nós utilizamos 5 dados volumétricos para medir o RPTINT. Uma simulação de injeção de combustível (fuel), uma tomografia computadorizada de um dente (tooth) e um crânio (skull), e o raio-X de um pé humano (foot) e um aneurisma (aneurism). Quatro modelos renderizados com nosso algoritmo são mostrados na Figura 5.2 (enquanto que o modelo "skull" foi mostrado na Figura 1.2).

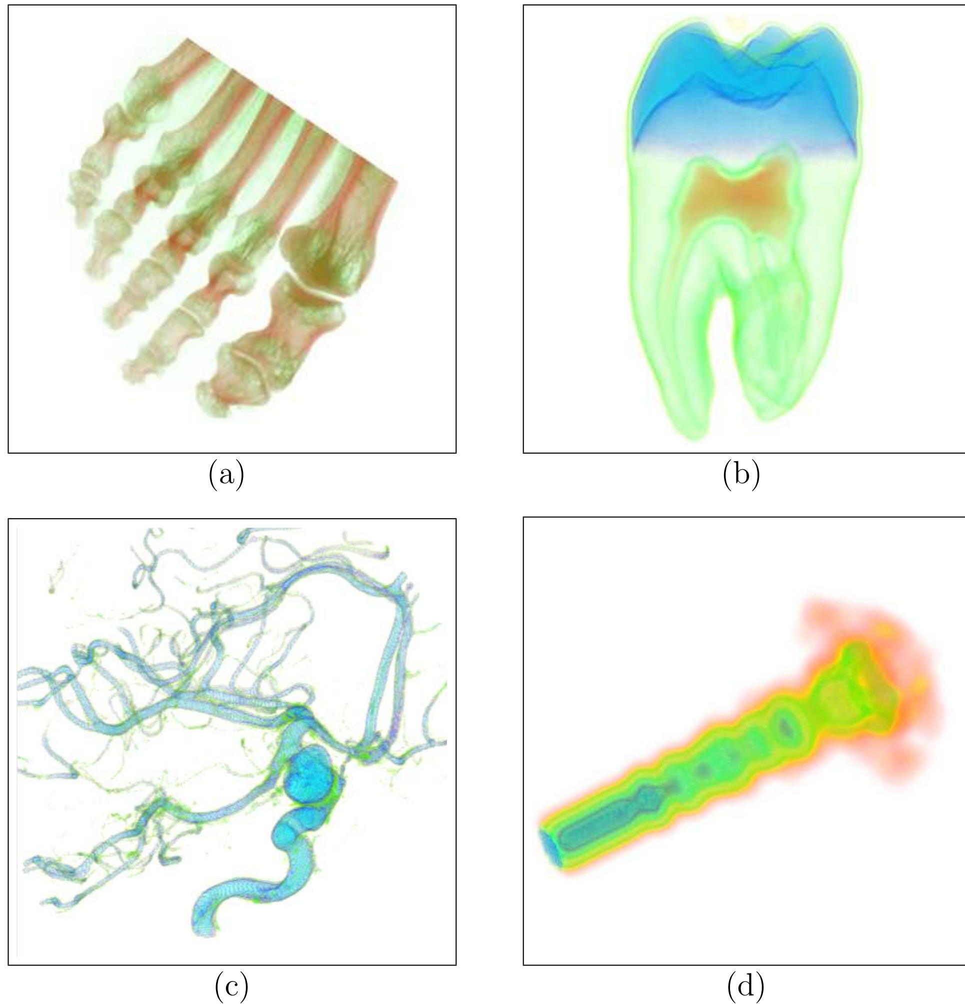


Figura 5.2: Imagens geradas pelo algoritmo RPTINT dos seguintes volumes regulares: "foot" (a), "tooth" (b), "aneurism" (c) e "fuel" (d).

Na Tabela 5.5, o número de vértices (# Verts) e tetraedros (# Tet) dependem das dimensões do volume regular original. As medidas de desempenhos são dadas em quadros por segundo (*fps*) e milhões de tetraedros por segundo (*M Tet/s*) para cada volume. Na verdade, esta última coluna contém dois valores: o primeiro é o valor nominal do número de tetraedros por segundo, incluindo aqueles que não contribuem para a geração da imagem final, enquanto que o segundo é o número efetivo, i.e. inclui apenas os tetraedros que foram realmente renderizados. Este número efetivo (dado entre parênteses) conta somente os tetraedros responsáveis por gerar a imagem final, uma vez que as *volunits* com opacidade zero são descartadas.

O tempo de organização dos vetores, usados para enviar a informação do dado volumétrico para GPU, e renderização do volume são dados na Tabela 5.6 (correspondendo ao terceiro e quarto passo do algoritmo RPTINT). No nosso algoritmo, o tempo médio gasto em renderização corresponde a mais que 60% do tempo total, demonstrando que o RPTINT gera imagens de dados volumétricos com um desempenho próximo ao limite da placa gráfica.

Volume	Organização	Renderização	% Total
fuel	0.005 s	0.009 s	64.28 %
tooth	1.377 s	6.484 s	82.47 %
foot	4.556 s	9.074 s	66.57 %
skull	4.606 s	8.593 s	65.10 %
aneurism	0.199 s	0.210 s	51.34 %

Tabela 5.6: Tempo gasto (em segundos) nos dois passos finais do RPTINT: organização dos vetores de vértice, cor, normal e contadores; e renderização do volume. A porcentagem de tempo gasto com renderização em relação ao tempo total é dada na última coluna.

IPTINT Medidas de desempenho do IPTINT foram feitas no mesmo PC usado no RPTINT, um Intel Pentium IV 3.6 GHz, 2 GB RAM, com uma placa gráfica nVidia GeForce 6800 256 MB conectada em um barramento PCI Express 16x.

Os dados volumétricos utilizados nos testes do IPTINT foram: “blunt”, “comb”, “post”, “spx+”, “fuel” e “neghip”. Imagens geradas pelo nosso algoritmo destes volumes podem ser vistas nas Figuras 5.3, 5.4 e 5.5.

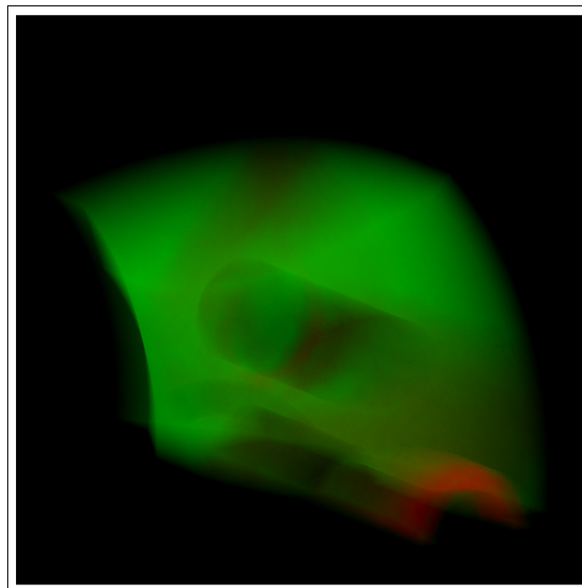
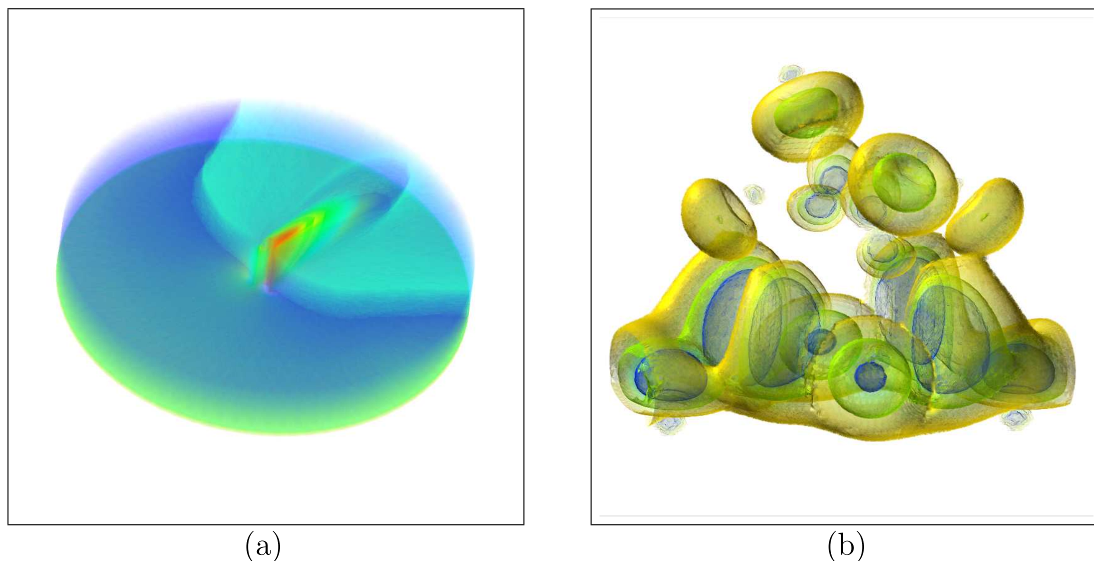


Figura 5.3: Volume “spx+” gerado pelo IPTINT.

Os tempos são dados usando uma janela de 512×512 e considerando que o volume está em constante rotação. A Tabela 5.7 compara o nosso algoritmo IPTINT com diversos algoritmos de visualização volumétrica em placa gráfica do estado da arte.

Algoritmo	blunt	post
PTINT	10.76 *fps*	4.35 *fps*
IPTINT	4.97 *fps*	2.09 *fps*
GATOR	4.07 *fps*	1.51 *fps*
VICP (GPU)	5.20 *fps*	1.93 *fps*
VICP (CPU)	1.82 *fps*	0.57 *fps*
VICP (Balanced)	4.10 *fps*	1.11 *fps*
HARC	4.47 *fps*	8.63 *fps*
HARC (INT)	4.94 *fps*	5.93 *fps*
HAVIS	2.36 *fps*	0.79 *fps*
HAVS (k=2)	6.09 *fps*	3.09 *fps*
HAVS (k=6)	3.45 *fps*	2.09 *fps*

Tabela 5.7: Comparação do algoritmo IPTINT e outras abordagens. O algoritmo base PTINT [23] é idêntico a executar o IPTINT sem suporte a renderização de iso-superfícies, apenas usando a pré-integração parcial.



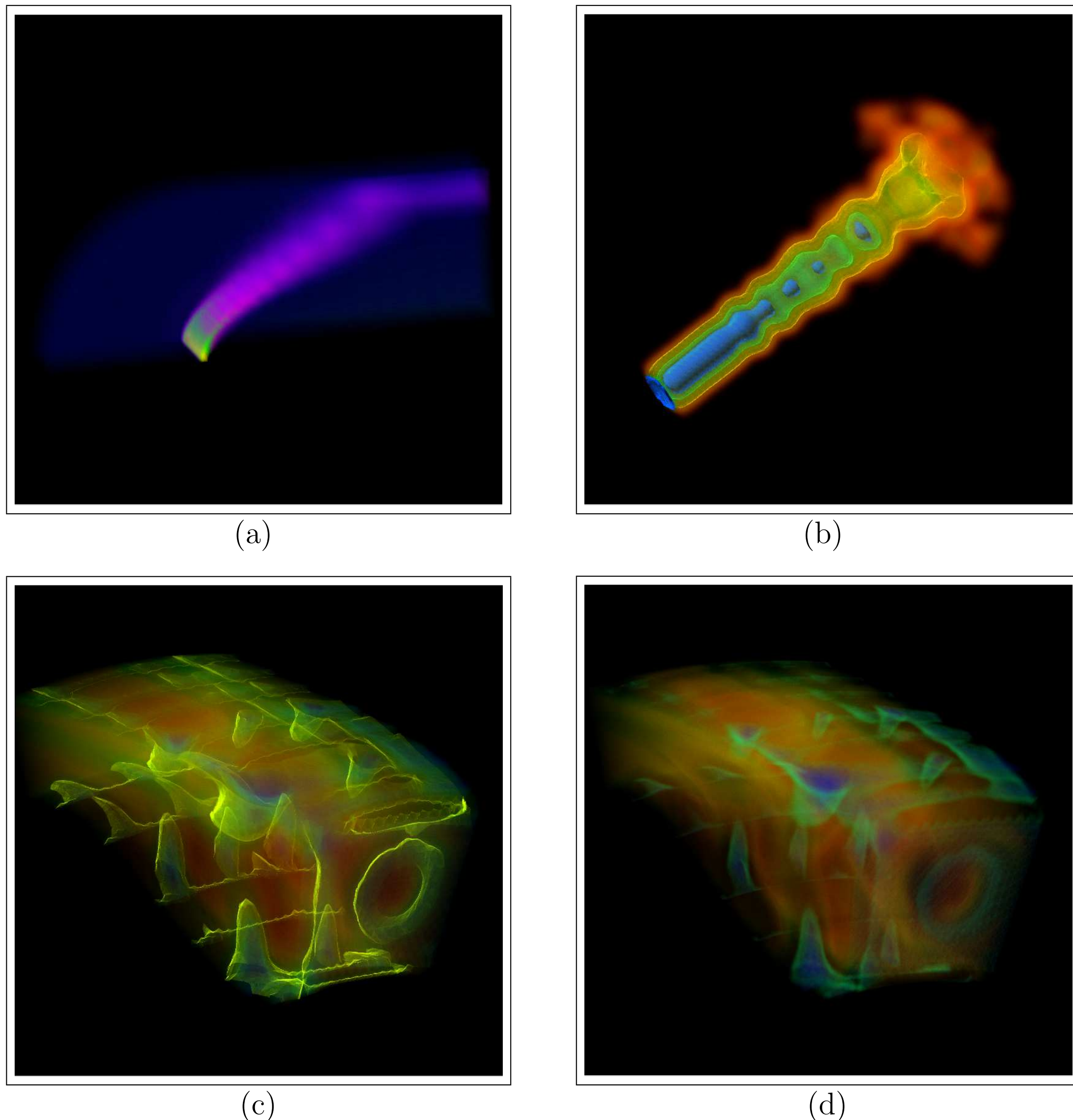
(a) (b)

Figura 5.4: Imagens geradas pelo IPTINT dos seguintes volumes: "post" (a) e "neghip" (b).

A Tabela 5.8 especifica o número de vértices (# Verts) e tetraedros (# Tet) de cada volume testado, o número médio de quadros por segundo (*fps*) e de tetraedros por segundo (*Tet/s*) do nosso algoritmo IPTINT em três variantes: com o método original de média de escalares (*Básico*) do PT [6]; usando a técnica de pré-integração parcial (*+INT*) de MORELAND e ANGEL [11]; e tanto realizando pré-integração parcial quanto renderização de iso-superfícies (*+ISO*).

Volume	Tamanho		*Básico*		*+INT*		*+ISO*	
	# Verts	# Tet	*fps*	*M Tet/s*	*fps*	*M Tet/s*	*fps*	*M Tet/s*
blunt	40 K	187 K	12.75	2.38	10.76	2.01	4.97	0.93
comb	47 K	215 K	11.12	2.38	8.98	1.92	3.71	0.79
post	110 K	513 K	4.91	2.51	4.35	2.23	2.09	1.07
spx+	150 K	828 K	4.68	2.55	4.52	2.47	1.23	1.02
fuel	262 K	1.25 M	26.01	2.10	22.99	1.86	9.95	0.80
neghip	262 K	1.25 M	3.59	2.34	3.11	2.03	1.27	0.83

Tabela 5.8: Tamanho dos volumes testados e medida de desempenho do algoritmo IPTINT para três diferentes renderizações: método básico de média de escalares; técnica de pré-integração parcial; e com pré-integração e extração de iso-superfícies.



(a) (b) (c) (d)

Figura 5.5: Imagens geradas pelo IPTINT dos volumes: “blunt” (a), “fuel” (b) e “comb” (c|d). Todas as imagens foram geradas usando pré-integração parcial, porém para o “fuel” e o “comb” (c) as iso-superfícies foram também renderizadas.

Nas Tabelas 5.7 e 5.8 é importante notar a diferença entre 2 dos dados volumétricos testados. Para o "blunt", nosso algoritmo IPTINT é competitivo com os outros algoritmos mesmo fazendo uma renderização híbrida do volume com iso-superfícies. Entretanto, para o "post", o IPTINT perde para os algoritmos de traçado de raios até mesmo se considerarmos a versão básica sem pré-integração e iso-superfícies. Esta característica pode ser atribuída ao fato que, para alguns pontos de vista, o modelo tem uma área pequena de pixels na janela, enquanto que, para abordagens de projeção de células, esta área de pixels é irrelevante.

As diferentes renderizações podem ser vistas nas Figuras 5.3 e 5.4. O dado "spx+" mostrado na Figura 5.3 foi renderizado com o método básico de média de escalares (equivalente a usar o algoritmo PT original). Ao passo que, na Figura 5.4, a imagem do "post" (a) foi gerada aplicando apenas a técnica de pré-integração parcial (equivalente a usar o algoritmo PTINT original) enquanto no caso do "neghip" (b) pré-integração e iluminação de iso-superfícies foram usadas.

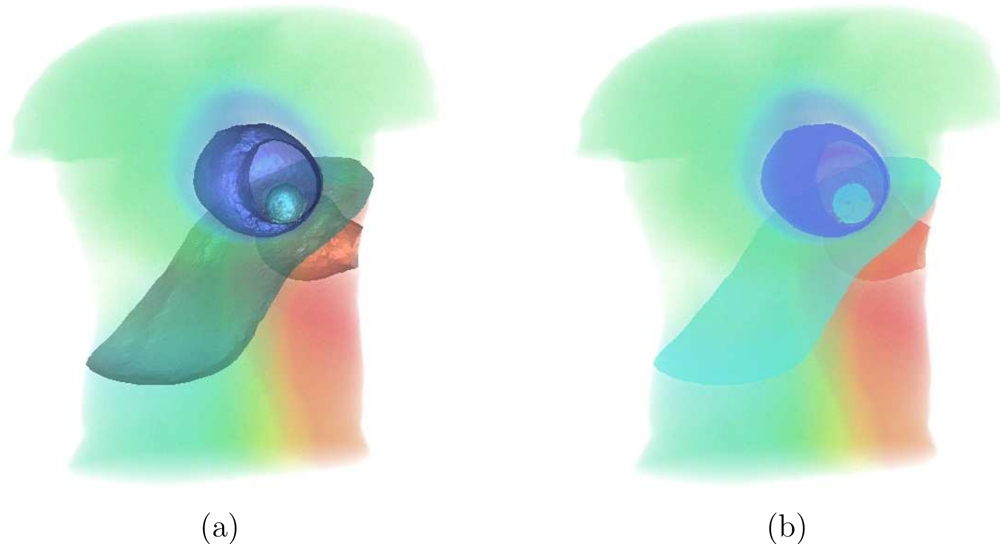
HAPT Nós testamos o algoritmo HAPT com os seguintes dados volumétricos irregulares: "blunt"; "post"; "spx+"; "delta"; "torso"; e "fighter". Adicionalmente, utilizamos o dado "turbjet" que varia no tempo (4D) para ilustrar a flexibilidade do nosso algoritmo. As Figuras 5.6, 5.7, 5.8 e 5.9 mostram alguns destes dados. Os tamanhos dos dados e os tempos de renderização são apresentados na Tabela 5.9. Os tempos são dados usando uma janela de resolução 512×512 pixels e considerando que o modelo está em constante rotação. Todas as medidas de tempo foram realizadas em um Intel Xeon E5345 CPU com 2.33 GHz por processador e 4 GB de RAM, usando uma GeForce 8800 GTX com 768 MB de VRAM (memória de GPU).

Volume	Tamanho		*DVR*		*ISO*		*DVR + ISO*	
	# Verts	# Tet	*fps*	*M Tet/s*	*fps*	*M Tet/s*	*fps*	*M Tet/s*
blunt	40 K	187 K	19.2	3.59	25.5	4.78	7.7	1.44
post	110 K	513 K	8.1	4.15	11.9	6.10	3.0	1.51
spx+	150 K	828 K	7.4	6.11	8.2	6.76	1.9	1.57
delta	211 K	1 M	4.5	4.52	6.0	6.01	1.5	1.51
torso	168 K	1.08 M	5.6	6.08	7.2	7.78	1.7	1.82
fighter	256 K	1.40 M	4.2	5.83	5.0	7.06	1.1	1.60
turbjet	212 K	1.01 M	17.5	17.67	n/a	n/a	n/a	n/a

Tabela 5.9: Tamanho dos volumes testados e medida de desempenho do algoritmo HAPT aplicando a técnica de visualização volumétrica direta (*DVR*), renderização de iso-superfícies (*ISO*) ou ambas.

Para estes resultados nós utilizamos visualização volumétrica direta, ou renderização de iso-superfície ou ambas (veja a Figura 5.6 e 5.7). Para os dois últimos casos, nós fixamos um número máximo de quatro iso-superfícies no framework do HAPT.

O problema é que o shader de geometria requer a especificação do número máximo de vértices de saída, o qual independente de usado completamente tem um impacto expressivo no desempenho. Este problema implica em um gasto significativo de tempo quando utilizamos ambos os métodos (duas últimas colunas da Tabela 5.9). Mesmo que seja improvável que quatro iso-superfícies cortem um mesmo tetraedro, nós contamos com este caso para deixar os resultados consistentes, o que deixa espaço para melhorias no algoritmo HAPT sem limitá-lo consideravelmente.



(a) (b)

Figura 5.6: Dado volumétrico "torso" renderizado com visualização volumétrica direta e indireta, as iso-superfícies são renderizadas com (a) e sem (b) iluminação.

A Tabela 5.10 compara o algoritmo HAPT com outros algoritmos baseados em GPU (porém sem extrair iso-superfícies) usando o volume "spx+". Os tempos dedicados para ordenação e renderização são detalhados nas colunas correspondentes.

Algoritmo	*Ordenação*	*Renderização*	*fps*	*M Tet/s*
$HAPT^Q$	0.03 s	0.09 s	7.4	6.11
$HAPT^B$	0.04 s	0.09 s	6.9	5.73
$HAPT^S$	0.08 s	0.09 s	5.4	4.50
$HAPT^M$	0.13 s	0.09 s	4.4	3.61
$HAVS^2$	0.09 s	0.11 s	5.0	4.14
$HAVS^6$	0.09 s	0.12 s	4.7	3.94
PTINT	0.19 s	0.20 s	2.4	2.06
GATOR	0.08 s	0.83 s	1.1	0.93
HARC	n/a	0.22 s	4.6	3.82
HARC (INT)	n/a	0.28 s	3.5	2.90

Tabela 5.10: Comparação de desempenho de visualização volumétrica direta do modelo "spx+" (828 K Tet) para diferentes algoritmos e critérios. Variações do HAPT são referentes à ordenação usada: **Q**uicksort; **B**itonic-sort; **S**TL-sort; e **M**PVONC. No caso do HAVS é para o tamanho do seu k-buffer: $k = \mathbf{2}$ e $k = \mathbf{6}$.

O algoritmo PTINT original usa uma ordenação simples por fatia durante a rotação do volume. Nos resultados apresentados aqui, nós avaliamos apenas a ordenação completa em CPU; apesar desta ordenação do PTINT ser equivalente ao método *STL-based* usado no GATOR e no HAPT^S, o PTINT ainda transfere os dados da GPU para CPU neste passo. Além disso, em ambos os casos para o HAVS, o tempo de ordenação conta apenas a pré-ordenação realizada em CPU, enquanto que para as abordagens do HARC não existe passo de ordenação, uma vez que algoritmos de traçado de raios atravessam o volume usando uma estrutura auxiliar de adjacência.

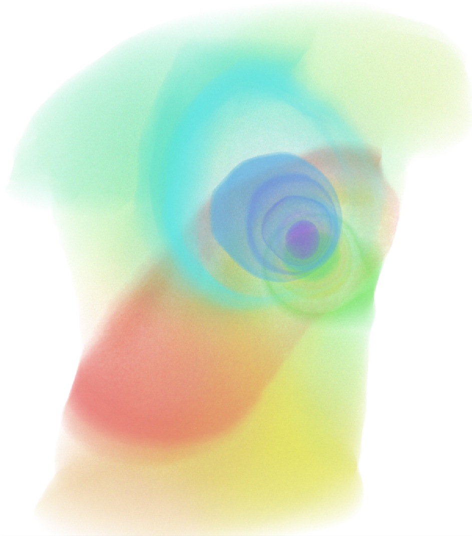


Figura 5.7: Visualização volumétrica direta do dado “torso” (sem renderização de iso-superfícies).

Uma observação importante é que o HAPT tem melhor tempo de renderização que os algoritmos comparados, incluindo abordagens de projeção de células, traçado de raios ou híbridas, e o HAPT não armazena o dado volumétrico em memória da GPU como feito pelo PTINT e HARC. Por outro lado, HAVS e GATOR renderiza por fluxo de maneira similar ao nosso algoritmo, porém requerendo múltiplos passos ou envio de dados redundantes. A principal variação no desempenho do HAPT vem da forma do tetraedro, o qual influencia a probabilidade da célula cair em cada uma das classes de projeção, ditando o número de triângulos gerados.

A Tabela 5.11 especifica quadros e tetraedros por segundo para os dados “torso” e “fighter” usando HAPT e outros métodos para comparação. Todos os resultados foram conduzidos usando apenas visualização volumétrica direta, limpando o pipeline da placa gráfica a cada quadro para não usufruir de renderização por fluxo contínuo. O fluxo do dado volumétrico usa VBOs para o HAPT, HAVS e GATOR (PTINT e HARC leem o volume da memória de textura e não podem se beneficiar de VBOs e fluxo contínuo). Esta comparação mostra que o HAPT é rápido mesmo quando lida com dados com mais de um milhão de células.

Algoritmo	"torso" 1082 K Tet		"fighter" 1403 K Tet	
	fps	*M Tet/s*	*fps*	*M Tet/s*
$HAPT^Q$	5.6	6.08	4.2	5.83
$HAPT^B$	4.3	4.68	3.6	5.09
$HAPT^S$	3.9	4.25	2.9	4.10
$HAPT^M$	1.6	1.73	1.2	1.62
$HAVS^2$	3.7	4.01	2.9	4.12
$HAVS^6$	3.3	3.60	2.7	3.89
PTINT	1.3	1.47	0.9	1.31
GATOR	0.7	0.76	0.4	0.56
HARC	4.8	5.19	3.8	5.33
HARC (INT)	3.9	4.22	3.0	4.21

Tabela 5.11: Comparação do algoritmo HAPT e outras abordagens para os volumes "torso" e "fighter".

Uma maneira de testar o desempenho de renderização por fluxo do nosso algoritmo para dados volumétricos grandes (dezenas de milhões de células) é renderizar dados que variam no tempo como uma sequência de volumes estáticos. Nós testamos o HAPT usando renderização por fluxo contínuo e um teste prévio de descarte de tetraedros com opacidade zero, explicado no Capítulo 3, para renderizar o dado "turbjet" (veja a Figura 5.8). Este dado 4D consiste de 150 quadros com 1 milhão de células por quadro, logo a animação completa do "turbjet" é composta de 150 milhões de células diferentes (veja a Tabela 5.9 para mais detalhes). Para este tamanho de dado, algoritmos que precisam guardar o volume inteiro em memória da GPU, como o PTINT (e a versão melhorada IPTINT apresentada nesta tese) e o HARC, são forçados a transferir em partes a animação da CPU para a GPU, impactando o desempenho excessivamente.

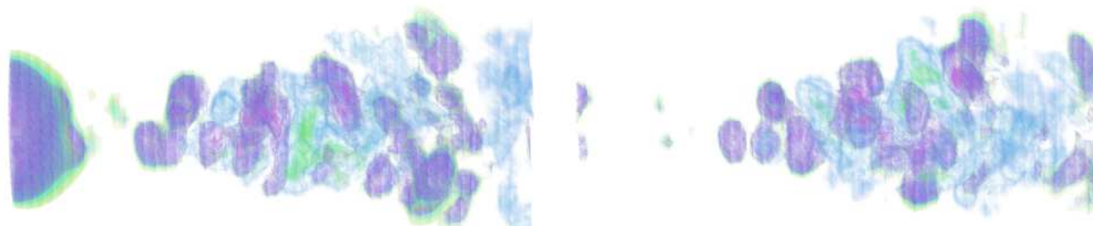
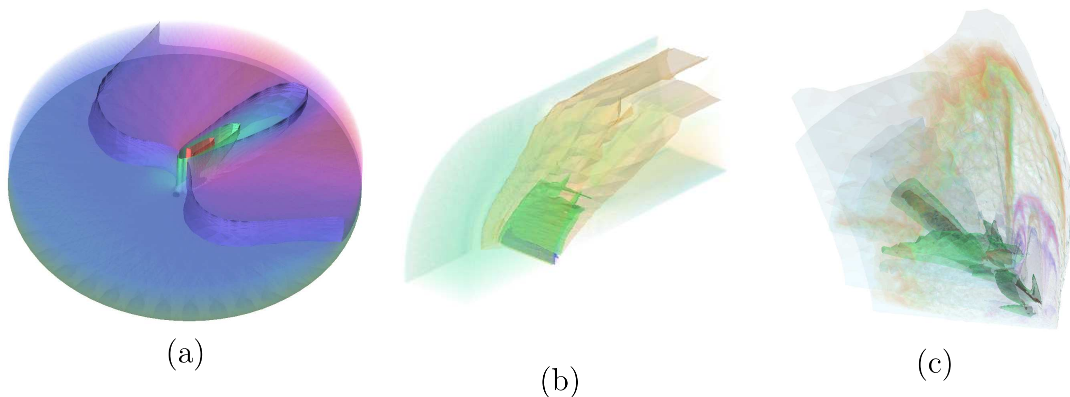


Figura 5.8: Dois dos 150 quadros do volume que varia no tempo "turbjet" (com 1 M Tet por quadro) renderizado usando o algoritmo HAPT. Alterando o framework do HAPT no shader de vértice, no intuito de realizar um teste prévio de descarte de tetraedros, o HAPT foi capaz de renderizar interativamente esta sequência na velocidade de 17 quadros por segundo.

GATOR e PTINT são os métodos mais similares ao HAPT já que eles também se baseiam no algoritmo PT original. Entretanto, ambos os métodos dependem de estratégias auxiliares para forçar a placa gráfica renderizar tetraedros. Em contraste, o HAPT evita ao mesmo tempo a redundância de dados quando envia o volume para a GPU imposta pelo GATOR, e a armazenagem do volume em memória da placa gráfica como realizada pelo PTINT. Nossa abordagem também evita o método em dois passos do PTINT, renderizando o volume em um único passo do pipeline gráfico.

O uso da pré-integração parcial melhora a qualidade de renderização do volume do GATOR e do algoritmo PT original, colocando o nosso algoritmo HAPT na mesma categoria de qualidade do PTINT. Quando comparado com abordagens de traçado de raios, como o HARC, e um esquema de interpolação mais precisa, como o HAVS, o HAPT apresenta os problemas de renderização, advindos da interpolação de escalares e espessura nas extremidades da projeção, herdados da ideia do PT. Este problema fica mais em evidência quando renderizamos volumes regulares convertidos para tetraedros, como foi feito na sequência do "turbjet" mostrada na Figura 5.8.



(a) (b) (c)

Figura 5.9: Visualização volumétrica direta e indireta com iso-superfícies iluminadas dos seguintes dados: "post" (a); "blunt" (b); e "fighter" (c).

Em termos de consumo de memória, o HAPT requer uma quantidade pequena e fixa de memória da GPU para guardar a função de transferência, a tabela de classificação (mesma TTT usada no PTINT) e a tabela de pré-integração parcial. Note que nenhuma estrutura de dados é relacionada com o tamanho do volume a ser renderizado, levando a um uso total de memória da placa gráfica de 10 KB. Quando utilizamos VBOs para renderizar por fluxo, nossa abordagem consome memória de GPU proporcional ao tamanho do dado. A opção econômica em memória é enviar as primitivas por fluxo sem usar VBOs, perdendo por volta de 10% de desempenho, porém evitando limitações de memória. Esta opção de usar memória da GPU ou não, renderizando primitivas ou usando VBOs, pode também ser explorada pelos métodos GATOR e HAVS, já que eles também se baseiam em renderização por fluxo. Em contraste com estes métodos, o HARC e o HARC (INT) requerem uma quantidade

significante de memória de 144 Bytes/Tet e 96 Bytes/Tet, respectivamente. Além disso, mesmo um algoritmo que gasta pouca memória, como o PTINT, precisaria de 3 GB de memória da GPU para armazenar a sequência inteira do “turbjet”.

Embora de natureza diferente, o HAVS é ainda o mais popular renderizador de volumes irregulares. Um ponto a favor do HAVS é que ele realiza uma ordenação mais apurada, quando comparado com os três métodos de ordenação por centróide apresentado nesta tese ($HAPT^Q$, $HAPT^B$ e $HAPT^S$). Contudo, o algoritmo HAPT oferece um pipeline flexível, tem um ganho de aproximadamente 50% de desempenho no caso do $HAPT^Q \times HAVS^6$, e não requer renderização em múltiplos passos. Esta última característica pode impactar o desempenho geral de renderização dependendo da placa gráfica.

Adicionalmente, nossa estratégia dissocia completamente ordenação de renderização, o que leva a alguns benefícios sobre algoritmos anteriores. PTINT depende de trazer os dados de volta para a CPU para serem ordenados entre passos. HAVS tem a ordenação acoplada com renderização e depende de dois passos de ordenação, um em CPU e outro em GPU. HARC, como qualquer método de traçado de raios, não requer a ordenação das células do volume, mas, por outro lado, depende de estruturas de dados auxiliares que aumentam a quantidade de acesso a memória e, consequentemente, diminuem o desempenho de renderização. Além disso, o volume deve ser armazenado na memória de textura da GPU, limitando o tamanho do volume possível de ser renderizado.

Comparação entre os algoritmos Quando comparamos os algoritmos de visualização volumétrica apresentados nesta tese, vemos que o VF-Ray-GPU não apresenta os problemas de renderização, herdados da ideia do algoritmo PT original, ao interpolar valores escalares e distância percorrida pelo raio nas extremidades da projeção dos tetraedros. Entretanto, as abordagens baseadas no PT tendem a serem mais rápidas e baratas em termos de consumo de memória. RPTINT pode tratar apenas dados regulares, tornando-o mais rápido que os demais quando renderiza este tipo de volume. HAPT é o algoritmo mais rápido de visualização volumétrica de dados irregulares, porém sem permitir a iluminação de iso-superfície usando o modelo de Phong como apresentado no IPTINT. O algoritmo HAPT gera os triângulos das iso-superfícies durante a renderização do volume no shader de geometria, o que impossibilita o cômputo de normais por vértice necessário na iluminação de Phong. Por outro lado, o IPTINT ilumina as iso-superfícies considerando os vetores gradientes do campo escalar como normais e possibilitando a aplicação de um modelo melhor de iluminação.

5.2 Processamento de Malhas

O método de processamento de malhas introduzido nesta tese apresenta o conceito de espaço dual de uma malha, definindo um novo descritor de similaridade e como usá-lo para posicionar os vértices da malha no espaço dual. Adicionalmente ao uso deste espaço de similaridade para ampliar tarefas de processamento de malhas, as distâncias Euclideanas entre pontos neste espaço podem também revelar simetrias refletivas. Estas simetrias podem ser vistas como um subconjunto da classe mais geral de similaridades que foi apresentada no nosso método SAMPLE. A Figura 5.10 retrata o modelo Stanford Armadillo e seu espaço dual com respeito a um de seus vértices. Note os padrões simétricos que aparecem naturalmente ao colorir os vértices por ordem de distância no espaço dual.

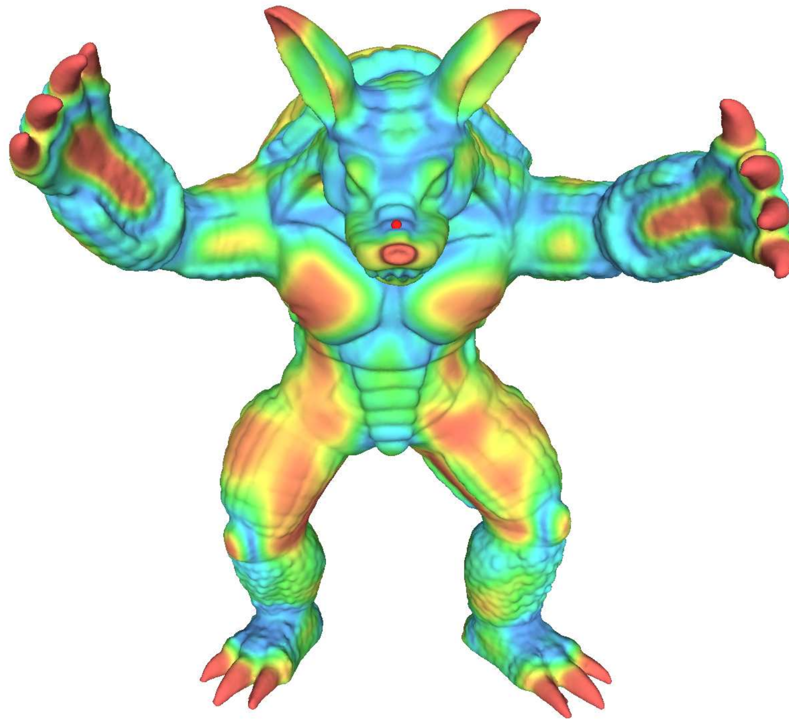


Figura 5.10: Exemplo de espaço dual para um dado vértice (em vermelho). A região ao redor do vértice é usada para mapear a malha inteira usando a vizinhança de similaridade. Regiões com rugas similares à região do vértice selecionado no nariz são pintadas de azul, como aquelas nos braços e pernas, enquanto regiões mais dissimilares são gradualmente pintadas de verde para vermelho.

Outro resultado interessante é o modelo 3DScanCo Angel, mostrado na Figura 5.11. Diferentemente do Armadillo, o Angel não exibe um simples plano de simetria, porém ele ainda contém uma estrutura simétrica inerente. O espaço dual para este modelo também destaca os padrões simétricos quando colorimos os vértices por distância de similaridade.

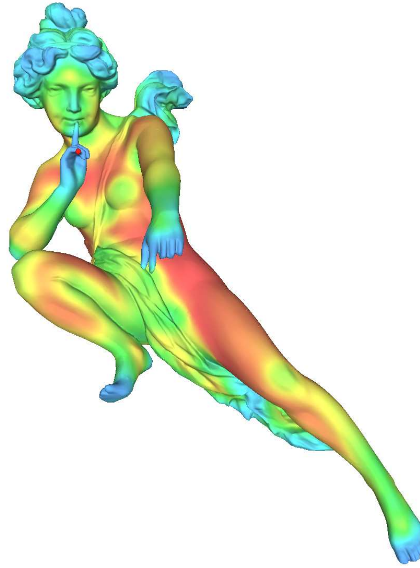


Figura 5.11: Exemplo de simetria usando o espaço dual. Para um vértice na mão direita (em vermelho), o modelo é pintado de regiões próximas (em azul) para distantes (em vermelho) no espaço dual.

Enquanto o espaço dual pode revelar simetrias globais, considerar somente os k vizinhos mais próximos neste espaço pode ser uma aplicação interessante. Em particular, quando $k = 1$, o espaço dual fornece um resultado próximo à intuição. A Figura 5.12 ilustra como a parametrização da superfície ao redor de um dos olhos do modelo Armadillo pode ser duplicada consistentemente para o outro olho.

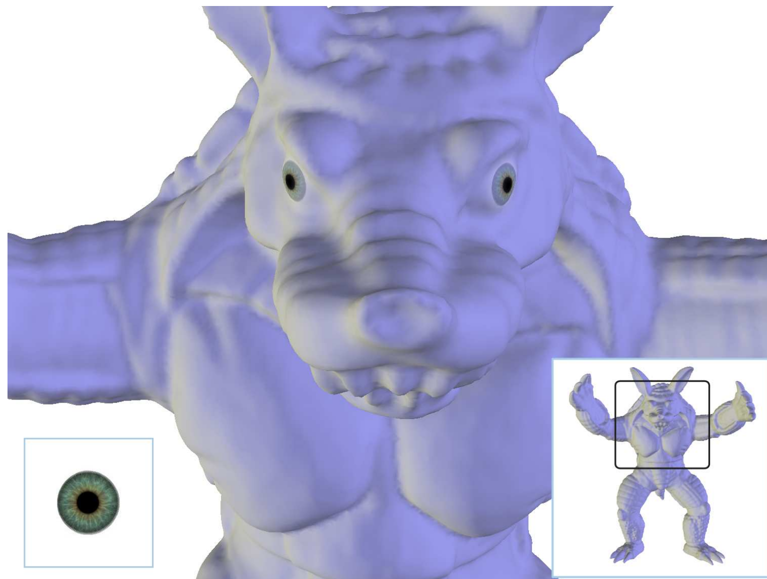


Figura 5.12: Exemplo de aplicação do vizinho dual imediato. Os vértices no centro de cada olho são vizinhos imediatos no espaço dual, escolher qualquer um dos vértices e aplicar o nosso método SAMPLE conduz ao mapeamento automático de uma textura (canto inferior esquerdo) para ambos os olhos.

Analisar os k vizinhos mais próximos no espaço de similaridade é também interessante para revelar regiões simétricas em modelos com uma grande quantidade de detalhes. Considere por exemplo o modelo *XYZ RGB Thai Statue* mostrado na Figura 5.13. Mesmo com um grande número de detalhes na superfície, nosso espaço dual é capaz de capturar similaridades pequenas usando $k = 0.5\%$ do total de vértices da malha.

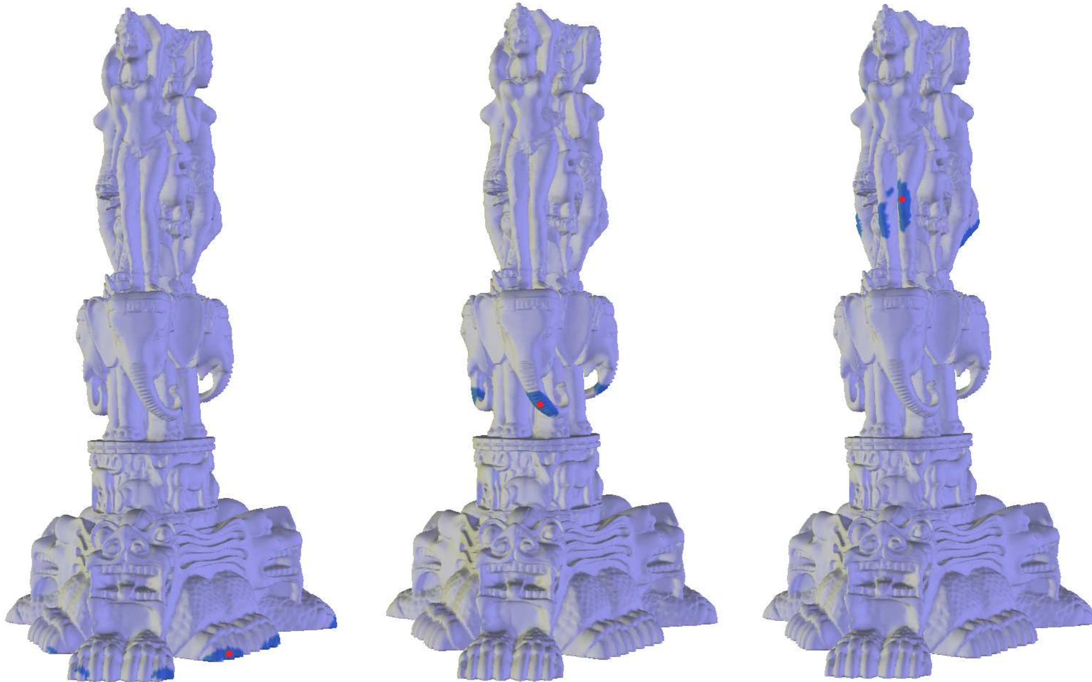


Figura 5.13: Similaridades mais próximas (em azul) do modelo XYZ RGB Thai Statue (com 125 K vértices). Três vértices (em vermelho) são usados neste exemplo: na parte inferior da estátua (esquerda) encontrando alguns dedos de pés; encontrando as três trompas dos elefantes no meio da estátua; e no topo (direita) encontrando alguns joelhos.

O tempo de pré-computação para construção do espaço dual é mostrado na Tabela 5.12 e depende das seguintes computações: do descritor por mapa de alturas, o qual é um procedimento padrão linear no número de vértices; da expansão em coeficientes de Zernike da imagem de cada mapa de alturas; e da computação dos coeficientes de Zernike com pesos Gaussianos, o qual é um somatório em uma vizinhança fixa também linear no número de vértices. A computação dos coeficientes de Zernike requer, para o modelo XYZ RGB Asian Dragon (com 108 K vértices), 18.6 segundos em um 2.33 GHz Intel Xeon E5345 CPU com 4 GB RAM. Além disso, o modelo Stanford Armadillo (com 172 K vértices) e 3DScanCo Angel (com 237 K vértices) requerem 28.8 e 40.2 segundos, respectivamente, provendo validação empírica que a computação dos coeficientes de Zernike também escala linearmente no número de vértices.

Modelo	# Verts	# Faces	Descritor	Zernike	Pesos Gaussianos
Dama	106 K	207 K	3.20 min	5.53 seg	2.47 min
Asian Dragon	108 K	216 K	2.56 min	5.54 seg	1.83 min
Thai Statue	125 K	250 K	2.98 min	6.58 seg	2.07 min
Armadillo	172 K	345 K	6.12 min	9.01 seg	5.34 min
Angel	237 K	474 K	17.76 min	12.15 seg	18.45 min

Tabela 5.12: Tempo de computação para estabelecer o espaço dual. As primeiras duas colunas mostram o número de vértices e faces de cada modelo, enquanto as três últimas mostram o tempo gasto em cada passo de pré-computação do nosso algoritmo (onde *min* é minutos e *seg* segundos).

O tempo gasto no cômputo dos vizinhos duais para um dado vértice, usando o espaço dual pré-computado, depende da avaliação das diferenças de similaridade e busca de vizinho mais próximo. Nós aplicamos um método de ordenação padrão para as diferenças de similaridade, tornando a busca por vizinhos trivial. A computação e ordenação das diferenças para um vértice selecionado requer 0.02 segundos para o modelo XYZ RGB Asian Dragon. O tempo de propagação, por outro lado, depende principalmente da tarefa de processamento de malhas que se deseja propagar. Nos nossos experimentos, a parametrização de uma escama do dragão consumiu 0.01 segundos, enquanto a propagação das 57 escamas similares consumiu 0.26 segundos. Finalmente, a operação de transferência de detalhes realizada nas mesmas escamas consumiu 0.29 segundos.

Capítulo 6

Conclusões

"Not only is the universe stranger than we imagine, it is stranger than we can imagine."
– Arthur Eddington

Nesta tese nós apresentamos algumas abordagens em GPU eficientes para visualização volumétrica e introduzimos a ideia de processamento de malhas ampliado por similaridade usando exemplares locais. Cada abordagem de visualização volumétrica visa oferecer vantagens em um contexto específico: VF-Ray-GPU é um algoritmo econômico em memória capaz de renderizar dados volumétricos grandes usando traçado de raios em GPU; RPTINT especializa o algoritmo PT original para dados regulares em placa gráfica, melhorando o desempenho consideravelmente; IPTINT adapta o PT usando uma abordagem em dois passos em GPU, adicionando visualização volumétrica indireta através de detecção de iso-superfícies e interpolação de gradientes; HAPT é eficiente por evitar múltiplos passos de renderização, e flexível possibilitando misturar e combinar facilmente outras estratégias, como métodos de ordenação e técnicas de renderização. Os primeiros dois algoritmos foram publicados em conferências internacionais: o VF-Ray-GPU no *Volume Graphics* 2008 [53]; e o RPTINT no *GRAPP* 2007 [54]. Os dois últimos algoritmos foram publicados na revista internacional *Computer Graphics Forum*, o IPTINT em 2008 [57] e o HAPT em 2010 em uma edição especial pela conferência EuroVis 2010. Nós disponibilizamos o código fonte junto com cada artigo na esperança de tornar as contribuições de pesquisa desta tese mais reproduzíveis.

O trabalho apresentado de processamento de malhas foi realizado durante o programa de doutorado sanduíche do CNPq que fiz com o Professor Varshney na Universidade de Maryland no ano de 2009. Neste trabalho, definimos uma medida de similaridade, baseada na imagem do mapa de alturas por vértice, para ser nosso

descritor de superfícies em um novo conceito de espaço dual. O espaço dual é usado para propagar o processamento de malhas feito em uma região da malha para regiões similares, usando parametrização e edição como duas aplicações de exemplo. Nosso método, chamado SAMPLE, foi recentemente submetido para a revista Graphical Models.

Nós acreditamos que o nosso método SAMPLE mostra resultados promissores e apresenta diversos caminhos para pesquisas futuras. Uma direção futura de pesquisa é considerar formulações diferentes para o descritor local de superfície. O mapa de alturas que usamos no momento pode ser visto como uma função escalar parametrizada no plano tangente do vértice central. Pode ser possível formar descritores mais informativos considerando diferentes parametrizações, como *authalic* ou *conformal*, para a região ao redor do vértice central. Outra possibilidade é considerar funções escalares diferentes, como curvatura ou saliência, para serem avaliadas sobre o domínio de parametrização. Um terceiro caminho interessante para pesquisa futura é estender o nosso descritor de superfícies para levar em consideração regiões em múltiplas escalas. O sucesso de descritores multi-escala no campo de processamento de imagens sugere que tal abordagem pode ter benefícios significantes para a presente abordagem de única escala implementada em nosso método SAMPLE.

Referências Bibliográficas

[1] SHREINER, D., WOO, M., NEIDER, J., et al. *OpenGL(R) Programming Guide (Red Book)*. 7th ed. London, UK, Addison-Wesley Professional, 2009. ISBN: 978-0321552624. Disponível em: <`http://www.opengl.org/documentation/red_book/`>.

[2] ROST, R. J., LICEA KANE, B., GINSBURG, D., et al. *OpenGL(R) Shading Language (Orange Book)*. 3rd ed. London, UK, Addison-Wesley Professional, 2009. ISBN: 978-0321637635. Disponível em: <`http://www.3dshaders.com/home/`>.

[3] MARROQUIM, R., MAXIMO, A. "Introduction to GPU Programming with GLSL", *Tutorials of the XXII Brazilian Symposium on Computer Graphics and Image Processing*, pp. 3–16, 2009, doi: http://doi.ieeecomputersociety.org/10.1109/SIBGRAPI-Tutorials.2009.9.

[4] "CUDA Environment – Compute Unified Device Architecture". . nVidia CUDA, 2007. Disponível em: <`http://www.nvidia.com/object/cuda_home.html`>.

[5] 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.

[6] SHIRLEY, P., TUCHMAN, A. "A Polygonal Approximation to Direct Scalar Volume Rendering", *SIGGRAPH Comput. Graph.*, v. 24, n. 5, pp. 63–70, 1990. ISSN: 0097-8930, doi: http://doi.acm.org/10.1145/99308.99322.

[7] RÖTTGER, 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, doi: http://doi.ieeecomputersociety.org/10.1109/VISUAL.2000.885683.

[8] WYLIE, B., MORELAND, K., FISK, L. A., et al. "Tetrahedral Projection Using Vertex Shaders", *Volume Visualization and*

Graphics, IEEE Symposium on, v. 0, pp. 7–12, 2002, doi: http://doi.ieeecomputersociety.org/10.1109/SWG.2002.1226504.

[9] 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.

[10] 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, doi: http://doi.ieeecomputersociety.org/10.1109/DAGSTUHL.1997.10002.

[11] MORELAND, K., ANGEL, E. "A Fast High Accuracy Volume Renderer for Unstructured Data". In: *VVS '04: Proceedings of the 2004 IEEE Symposium on Volume visualization and graphics*, pp. 13–22, Piscataway, NJ, USA, 2004. IEEE Press. ISBN: 0-7803-7641-2, doi: http://dx.doi.org/10.1109/VV.2004.2.

[12] MORELAND, K. D. *Fast High Accuracy Volume Rendering*. Tese de Doutorado, University of New Mexico, July 2004. Disponível em: <`http://www.cs.unm.edu/~kmorel/`>.

[13] SILVA, C. T., MITCHELL, J. S. B., WILLIAMS, P. L. "An Exact Interactive Time Visibility Ordering Algorithm for Polyhedral Cell Complexes". In: *VVS '98: Proceedings of the 1998 IEEE symposium on Volume visualization*, pp. 87–94, New York, NY, USA, 1998. ACM. ISBN: 1-58113-105-4, doi: http://doi.acm.org/10.1145/288126.288170.

[14] KRAUS, M., ERTL, T. "Cell-Projection of Cyclic Meshes". In: *VIS '01: Proceedings of the conference on Visualization '01*, pp. 215–222, Washington, DC, USA, 2001. IEEE Computer Society. ISBN: 0-7803-7200-X, doi: http://doi.ieeecomputersociety.org/10.1109/VISUAL.2001.964514.

[15] COOK, R., MAX, N., SILVA, C. T., et al. "Image-Space Visibility Ordering for Cell Projection Volume Rendering of Unstructured Data", *IEEE Transactions on Visualization and Computer Graphics*, v. 10, n. 6, pp. 695–707, Nov.-Dec. 2004. ISSN: 1077-2626, doi: http://doi.ieeecomputersociety.org/10.1109/TVCG.2004.45.

[16] WEILER, M., KRAUS, M., ERTL, T. "Hardware-Based View-Independent Cell Projection", *Volume Visualization and Graphics, IEEE Symposium on*, v. 0, pp. 13–22, 2002, doi: http://doi.ieeecomputersociety.org/10.1109/SWG.2002.1226505.

[17] CALLAHAN, S., IKITS, M., COMBA, J., et al. "Hardware-Assisted Visibility Ordering for Unstructured Volume Rendering", *IEEE Transactions on Visualization and Computer Graphics*, v. 11, n. 3, pp. 285–295, 2005, doi: http://dx.doi.org/10.1109/TVCG.2005.46.

[18] BLINN, J. F. "Light Reflection Functions for Simulation of Clouds and Dusty Surfaces". In: *SIGGRAPH '82: Proceedings of the 9th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 21–29, New York, NY, USA, 1982. ACM Press. ISBN: 0-89791-076-1, doi: http://doi.acm.org/10.1145/800064.801255.

[19] 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.

[20] 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*, pp. 333–340, 2003. ISBN: 0-7695-2030-8/03, doi: http://dx.doi.org/10.1109/VISUAL.2003.1250390.

[21] 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.

[22] KRAUS, M., QIAO, W., EBERT, D. S. "Projecting Tetrahedra Without Rendering Artifacts". In: *VIS '04: Proceedings of the 15th IEEE conference on Visualization '04*, pp. 27–34, Washington, DC, USA, 2004. IEEE Computer Society. ISBN: 0-7803-8788-0, doi: http://dx.doi.org/10.1109/VIS.2004.85.

[23] MARROQUIM, R., MAXIMO, A., FARIAS, R., et al. "GPU-Based Cell Projection for Interactive Volume Rendering", *Computer Graphics and Image Processing, Brazilian Symposium on*, v. 0, pp. 147–154, 2006. ISSN: 1530-1834, doi: http://doi.ieeecomputersociety.org/10.1109/SIBGRAPI.2006.22.

[24] KAZHDAN, M., CHAZELLE, B., DOBKIN, D., et al. "A Reflective Symmetry Descriptor for 3D Models", *Algorithmica*, v. 38, n. 1, pp. 201–225, 2003. ISSN: 0178-4617, doi: http://dx.doi.org/10.1007/s00453-003-1050-5.

[25] PODOLAK, J., SHILANE, P., GOLOVINSKIY, A., et al. "A Planar-Reflective Symmetry Transform for 3D Shapes". In: *SIGGRAPH '06: Proceedings of the 33th annual conference on Computer graphics and interactive techniques*, pp. 549–559, New York, NY, USA, 2006. ACM. ISBN: 1-59593-364-6, doi: http://doi.acm.org/10.1145/1179352.1141923.

[26] XU, K., ZHANG, H., TAGLIASACCHI, A., et al. "Partial Intrinsic Reflectional Symmetry of 3D Shapes". In: *SIGGRAPH Asia '09: ACM SIGGRAPH Asia 2009 papers*, pp. 1–10, New York, NY, USA, 2009. ACM. ISBN: 978-1-60558-858-2, doi: http://doi.acm.org/10.1145/1661412.1618484.

[27] RUSTAMOV, R. M. "Laplace-Beltrami eigenfunctions for deformation invariant shape representation". In: *SGP '07: Proceedings of the Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, pp. 225–233, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association. ISBN: 978-3-905673-46-3, doi: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.86.6567.

[28] OVSJANIKOV, M., SUN, J., GUIBAS, L. J. "Global Intrinsic Symmetries of Shapes." *Comput. Graph. Forum*, v. 27, n. 5, pp. 1341–1348, 2008, doi: http://dx.doi.org/10.1111/j.1467-8659.2008.01273.x.

[29] SUN, J., OVSJANIKOV, M., GUIBAS, L. J. "A Concise and Provably Informative Multi-Scale Signature Based on Heat Diffusion", *Comput. Graph. Forum*, v. 28, n. 5, pp. 1383–1392, 2009, doi: http://dx.doi.org/10.1111/j.1467-8659.2009.01515.x.

[30] GOLOVINSKIY, A., PODOLAK, J., FUNKHOUSER, T. "Symmetry-Aware Mesh Processing". In: *Proceedings of the 13th IMA International Conference on Mathematics of Surfaces*, pp. 170–188, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN: 978-3-642-03595-1, doi: http://dx.doi.org/10.1007/978-3-642-03596-8_10.

[31] ZELINKA, S., GARLAND, M. "Similarity-Based Surface Modelling using Geodesic Fans". In: *SGP '04: Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pp. 204–213, New York, NY, USA, 2004. ACM. ISBN: 3-905673-13-4, doi: http://doi.acm.org/10.1145/1057432.1057460.

[32] GAL, R., COHEN OR, D. "Salient Geometric Features for Partial Shape Matching and Similarity", *ACM Trans. Graph.*, v. 25, n. 1, pp. 130–150, 2006. ISSN: 0730-0301, doi: http://doi.acm.org/10.1145/1122501.1122507.

[33] PODOLAK, J., GOLOVINSKIY, A., RUSINKIEWICZ, S. "Symmetry-Enhanced Remeshing of Surfaces". In: *SGP '07: Proceedings of the Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, pp. 235–242, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association. ISBN: 978-3-905673-46-3, doi: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.83.7217.

[34] PALACIOS, J., ZHANG, E. "Rotational symmetry field design on surfaces", *ACM Trans. Graph.*, v. 26, n. 3, pp. 55, 2007. ISSN: 0730-0301, doi: http://doi.acm.org/10.1145/1276377.1276446.

[35] RAY, N., VALLET, B., LI, W., et al. "N-Symmetry Direction Field Design", *ACM Trans. Graph.*, v. 27, n. 2, pp. 1–13, 2008. ISSN: 0730-0301, doi: http://doi.acm.org/10.1145/1356682.1356683.

[36] KIM, Y., LEE, C. H., VARSHNEY, A. "Vertex-Transformation Streams", *Graph. Models*, v. 68, n. 4, pp. 371–383, 2006. ISSN: 1524-0703, doi: http://dx.doi.org/10.1016/j.gmod.2006.03.005.

[37] KAZHDAN, M., FUNKHOUSER, T., RUSINKIEWICZ, S. "Shape Matching and Anisotropy", *ACM Trans. Graph.*, v. 23, n. 3, pp. 623–629, 2004. ISSN: 0730-0301, doi: http://doi.acm.org/10.1145/1015706.1015770.

[38] KAZHDAN, M., FUNKHOUSER, T., RUSINKIEWICZ, S. "Symmetry Descriptors and 3D Shape Matching". In: *SGP '04: Proceedings of the Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, pp. 115–123, New York, NY, USA, 2004. ACM. ISBN: 3-905673-13-4, doi: http://doi.acm.org/10.1145/1057432.1057448.

[39] TAL, A., ZUCKERBERGER, E. "Mesh Retrieval by Components". In: *GRAPP*, pp. 142–149, 2006, doi: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.6.1900.

[40] LIU, R., ZHANG, H., SHAMIR, A., et al. "A Part-Aware Surface Metric for Shape Analysis", *Computer Graphics Forum (Proceedings of Eurographics)*, v. 28, n. 2, pp. 397–406, 2009, doi: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.151.655.

[41] BERNER, A., BOKELOH, M., WAND, M., et al. "A Graph-Based Approach to Symmetry Detection". In: *Symposium on Volume and Point-Based Graphics*, pp. 1–8, Los Angeles, CA, 2008. Eurographics Association, doi: http://dx.doi.org/10.2312/VG/VG-PBG08/001-008.

[42] GATZKE, T., GRIMM, C., GARLAND, M., et al. "Curvature Maps For Local Shape Comparison", *International Conference on Shape Modeling and Applications*, pp. 244–253, June 2005, doi: http://dx.doi.org/10.1109/SMI.2005.13.

[43] KARNI, Z., GOTSMAN, C. "Spectral Compression of Mesh Geometry". In: *SIGGRAPH '00: Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 279–286, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co. ISBN: 1-58113-208-5, doi: http://doi.acm.org/10.1145/344779.344924.

[44] GRINSPUN, E., SCHRÖDER, P., DESBRUN, M. *Discrete Differential Geometry: An Applied Introduction.* Boston, US, ACM SIGGRAPH 2006 Courses, 2006. Disponível em: <`http://ddg.cs.columbia.edu/`>.

[45] VALLET, B., LÉVY, B. "Spectral Geometry Processing with Manifold Harmonics", *Computer Graphics Forum (Proceedings of Eurographics)*, v. 27, n. 2, pp. 251–260, 2008, doi: http://dx.doi.org/10.1111/j.1467-8659.2008.01122.x.

[46] SIMARI, P., KALOGERAKIS, E., SINGH, K. "Folding meshes: Hierarchical mesh segmentation based on planar symmetry". In: *SGP '06: Proceedings of the Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, pp. 111–119, Aire-la-Ville, Switzerland, Switzerland, 2006. Eurographics Association. ISBN: 30905673-36-3, doi: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.61.5100.

[47] CHENG, Z., DANG, G., JIN, S. "A Meaningful Mesh Segmentation Based on Local Self-similarity Analysis", *10th IEEE International Conference on Computer-Aided Design and Computer Graphics*, pp. 288–293, Oct. 2007, doi: http://dx.doi.org/10.1109/CADCG.2007.4407896.

[48] MITRA, N. J., GUIBAS, L. J., PAULY, M. "Partial and approximate symmetry detection for 3D geometry", *ACM Trans. Graph.*, v. 25, n. 3, pp. 560–568, 2006. ISSN: 0730-0301, doi: http://doi.acm.org/10.1145/1141911.1141924.

[49] YOSHIZAWA, S., BELYAEV, A., SEIDEL, H. "Smoothing by Example: Mesh Denoising by Averaging with Similarity-Based Weights". In: *International Conference on Shape Modeling and Applications*, p. 9, Washington, DC, USA, 2006. IEEE Computer Society. ISBN: 0-7695-2591-1, doi: http://dx.doi.org/10.1109/SMI.2006.38.

[50] SCHALL, O., BELYAEV, A., SEIDEL, H. "Feature-preserving non-local denoising of static and time-varying range data". In: *SPM '07: Proceedings of the ACM Symposium on Solid and Physical Modeling*, pp. 217–222, New York, NY, USA, 2007. ACM. ISBN: 978-1-59593-666-0, doi: http://doi.acm.org/10.1145/1236246.1236277.

[51] RIBEIRO, S., MAXIMO, A., BENTES, C., 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.

[52] PINA, A., BENTES, C., FARIAS, R. "Memory Efficient and Robust Software Implementation of the Raycast Algorithm". In: *WSCG'07: The 15th Int. Conf. in Central Europe on Computer Graphics, Visualization and Computer Vision*, 2007.

[53] MAXIMO, A., RIBEIRO, S., BENTES, C., et al. "Memory Efficient GPU-Based Ray Casting for Unstructured Volume Rendering". In: *VG-PBG*, 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.

[54] MAXIMO, A., MARROQUIM, R., FARIAS, R., et al. "GPU-Based Cell Projection for Large Structured Data Sets". In: *GRAPP (GM/R)*, pp. 312–322. INSTICC – Institute for Systems and Technologies of Information, Control and Communication, 2007. ISBN: 978-972-8865-71-9, doi: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.86.6833.

[55] MAXIMO, A. *Projeção de Células baseada em GPU para Visualização Interativa de Volumes*. Dissertação de Mestrado, UFRJ – Federal University of Rio de Janeiro, RJ, Brazil, Nov 2006. Disponível em: <`http://www.lcg.ufrj.br/Members/andream`>.

[56] WILLIAMS, P. L. "Visibility-Ordering Meshed Polyhedra", *ACM Trans. Graph.*, v. 11, n. 2, pp. 103–126, 1992. ISSN: 0730-0301, doi: http://doi.acm.org/10.1145/130826.130899.

[57] MARROQUIM, R., MAXIMO, A., FARIAS, R., et al. "Volume and Isosurface Rendering with GPU-Accelerated Cell Projection", *Computer Graphics Forum*, v. 27, pp. 24–35, 2008. ISSN: 0167-7055, doi: http://dx.doi.org/10.1111/j.1467-8659.2007.01038.x.

[58] KLEIN, T., STEGMAIER, S., ERTL, T. "Hardware-Accelerated Reconstruction of Polygonal Isosurface Representations on Unstructured Grids". In: *PG '04: Proceedings of the 12th Pacific Conference on Computer Graphics and Applications*, pp. 186–195, Washington, DC, USA, 2004. IEEE Computer Society. ISBN: 0-7695-2234-3, doi: http://doi.ieeecomputersociety.org/10.1109/PCCGA.2004.1348349.

[59] PASCUCCI, V. "Isosurface Computation Made Simple: Hardware Acceleration, Adaptive Refinement and Tetrahedral Stripping". In: *In Joint Eurographics - IEEE VGTC Symposium on Visualization (VisSym)*, pp. 293–300, 2004, doi: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.2.6156.

[60] MAXIMO, A., MARROQUIM, R., FARIAS, R. "Hardware-Assisted Projected Tetrahedra", *Computer Graphics Forum*, v. 29, pp. 903–912, 2010. ISSN: 0167-7055, doi: http://dx.doi.org/10.1111/j.1467-8659.2009.01673.x.

[61] CEDERMAN, D., TSIGAS, P. "A Practical Quicksort Algorithm for Graphics Processors". In: *ESA '08: Proceedings of the 16th annual European symposium on Algorithms*, pp. 246–258, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN: 978-3-540-87743-1, doi: http://dx.doi.org/10.1007/978-3-540-87744-8_21.

[62] PLAUGER, P. J., LEE, M., MUSSER, D., et al. *C++ Standard Template Library*. Upper Saddle River, NJ, USA, Prentice Hall PTR, 2000. ISBN: 0134376331.

[63] "CUDA SDK Bitonic Sort Example". . nVidia™ CUDA, 2007. Disponível em: <`http://developer.download.nvidia.com/compute/cuda/sdk`>.

[64] TREECE, G. M., PRAGER, R. W., GEE, A. H. "Regularised Marching Tetrahedra: Improved Iso-Surface Extraction", *Computers & Graphics*, v. 23, n. 4, pp. 583–598, 1999, doi: http://dx.doi.org/10.1016/S0097-8493(99)00076-X.

[65] TAUBIN, G. "A Signal Processing Approach to Fair Surface Design". In: *SIGGRAPH '95: Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, pp. 351–

358, New York, NY, USA, 1995. ACM. ISBN: 0-89791-701-4, doi: http://doi.acm.org/10.1145/218380.218473.

[66] LEE, C. H., VARSHNEY, A., JACOBS, D. W. "Mesh Saliency", *ACM Trans. Graph.*, v. 24, n. 3, pp. 659–666, 2005. ISSN: 0730-0301, doi: http://doi.acm.org/10.1145/1073204.1073244.

[67] ZERNIKE, F. "Beugungstheorie des Schneidenverfahrens und seiner verbesserten Form, der Phasenkontrastmethode", *Physica 1*, pp. 689–704, 1934.

[68] KHOTANZAD, A., HONG, Y. H. "Rotation invariant pattern recognition using Zernike moments". In: *9th International Conference on Pattern Recognition*, v. 1, pp. 326–328, Nov 1988, doi: http://dx.doi.org/10.1109/ICPR.1988.28233.

[69] REVAUD, J., LAVOUE, G., BASKURT, A. "Improving Zernike Moments Comparison for Optimal Similarity and Rotation Angle Retrieval", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. 31, pp. 627–636, 2008. ISSN: 0162-8828, doi: http://doi.ieeecomputersociety.org/10.1109/TPAMI.2008.115.

[70] FLOATER, M. S. "Mean Value Coordinates", *Comput. Aided Geom. Des.*, v. 20, n. 1, pp. 19–27, 2003. ISSN: 0167-8396, doi: http://dx.doi.org/10.1016/S0167-8396(02)00002-5.

[71] "OpenGL Utility Toolkit". . GLUT, 2010. Disponível em: <`http://www.opengl.org/resources/libraries/glut`>.

[72] "CGAL, Computational Geometry Algorithms Library". . CGAL, 2010. Disponível em: <`http://www.cgal.org/`>.

[73] "Boost C++ Libraries". . Boost, 2010. Disponível em: <`http://www.boost.org/`>.

[74] "Visual Computing Lab Library". . VCGLib, 2010. Disponível em: <`http://vcg.sourceforge.net/`>.

[75] "Toolkit do Laboratório de Computação Gráfica". . LCGtk, 2010. Disponível em: <`http://code.google.com/p/lcgtk/`>.

[76] 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.

[77] PHONG, B. T. "Illumination for Computer Generated Pictures", *Commun. ACM*, v. 18, n. 6, pp. 311–317, 1975. ISSN: 0001-0782, doi: http://doi.acm.org/10.1145/360825.360839.

Apêndice A

Algoritmos Desenvolvidos

Algoritmo VF-Ray-GPU [53] (*Visible-Face Driven Ray Casting implemented on the GPU*) – código fonte: http://code.google.com/p/vfray; doi: http://doi.ieeecomputersociety.org/10.2312/VG/VG-PBG08/155-162.

Algoritmo RPTINT [54] (*Regular Projected Tetrahedra with Partial Pre-Integration*) – código fonte: http://code.google.com/p/rptint; doi: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.86.6833.

Algoritmo IPTINT [57] (*Improved Projected Tetrahedra with Partial Pre-Integration*) – código fonte: http://code.google.com/p/ptint; doi: http://dx.doi.org/10.1111/j.1467-8659.2007.01038.x.

Algoritmo HAPT [60] (*Hardware-Assisted Projected Tetrahedra*) – código fonte: http://code.google.com/p/hapt; doi: http://dx.doi.org/10.1111/j.1467-8659.2009.01673.x.

Algoritmo SAMPLE [submetido] (*Similarity Augmented Mesh Processing using Local Exemplars*) – código fonte e publicação ainda não disponíveis.

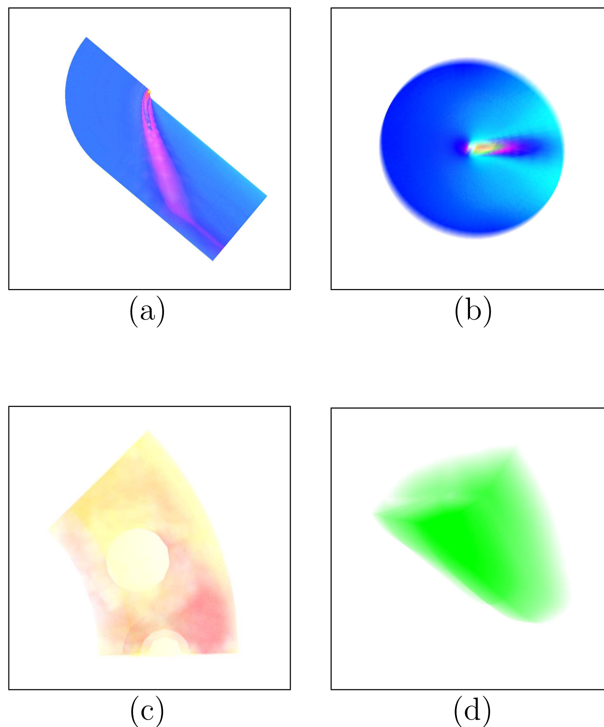
Apêndice B

Algoritmo VF-Ray

Visible-Face Driven Ray Casting [51] – http://code.google.com/p/vfray

O algoritmo VF-Ray forma a base do algoritmo VF-Ray-GPU apresentado por esta tese na Seção 3.1. Este apêndice complementa a explicação apresentada com figuras e tabelas referentes ao VF-Ray.

A Figura abaixo mostra 4 resultados de renderização usando o VF-Ray: o *Blunt Fin* (a) e o *Oxygen Post* (b) são experiências da interação do oxigênio em um ambiente; o *SPX* (c) apresenta áreas de possíveis vazamentos de um reator; e o *Delta Wing* (d) são experiência com uma asa delta:



(a) (b)

(c) (d)

Os experimentos realizados no algoritmo VF-Ray foram conduzidos em um Intel Pentium IV 3.6 GHz com 2 GB de RAM e 1 MB de memória cache L2. O algoritmo foi escrito em C/C++ ANSI sem utilizar nenhuma biblioteca gráfica em particular.

Na tabela seguinte são apresentados resultados de consumo de memória (em Mega Bytes), na renderização de um quadro em diferentes resoluções, do algoritmo VF-Ray para os quatro volumes exibidos na página anterior. Os percentuais apresentados nesta tabela são a razão do consumo de memória de outros algoritmos sobre o VF-Ray. Os seguintes algoritmos foram testados: *ME-Ray – Memory Efficient Ray Casting* e *EME-Ray – Enhanced Memory Efficient Ray Casting* de PINA *et al.* [52]; algoritmo *Bunyk* [10]; e *ZSweep* de FARIAS *et al.* [76].

Volume	Resolução	Memória (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%
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%
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%
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%

Na próxima tabela são apresentados resultados de desempenho (em segundos), na renderização de um quadro em diferentes resoluções, do algoritmo VF-Ray e outros algoritmos de visualização. Os volumes utilizados e os algoritmos comparados são os mesmos da tabela anterior.

Volume	Resolução	Tempo (s)	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%
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%
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%
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%

O algoritmo VF-Ray (*Visible-Face Driven Ray Casting*) foi publicado com o título *Memory-Aware and Efficient Ray-Casting Algorithm* na conferência internacional SIBGRAPI 2007 [51].

Apêndice C

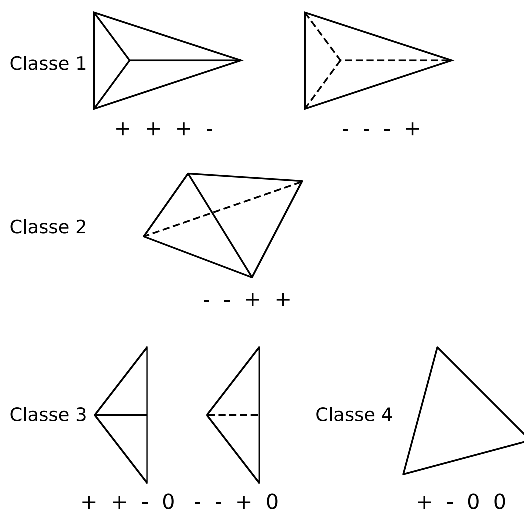
Algoritmo PT

Projected Tetrahedra [6] – http://doi.acm.org/10.1145/99308.99322

O algoritmo PT forma a base principal dos algoritmos de projeção de células apresentados nesta tese. Este apêndice explica o algoritmo original PT de SHIRLEY e TUCHMAN [6].

O algoritmo *Projected Tetrahedra* (PT) consiste em projetar tetraedros no plano da imagem, e compor as projeções em ordem de visibilidade. No caso de volumes com células diferentes de tetraedros, cada célula deve ser dividida em tetraedros. No algoritmo PT original é apresentado um método para tetraedrizar um cubo (ou hexaedro) visando tratar malhas regulares. Esta ideia é utilizada nesta tese no algoritmo RPTINT apresentado na Seção 3.2.

A forma dos tetraedros projetados é classificada em quatro classes diferentes, como mostrado na Figura abaixo. As classes representam as quatro possíveis silhuetas de projeção do tetraedro no plano da imagem. A notação $+$, $-$ e 0 usada é explicada a seguir. Note que, as classes 3 e 4 são casos degenerados das classes 1 e 2, onde um dos vértices é projetado sobre uma aresta (classe 3) ou sobre outro vértice (classe 4).

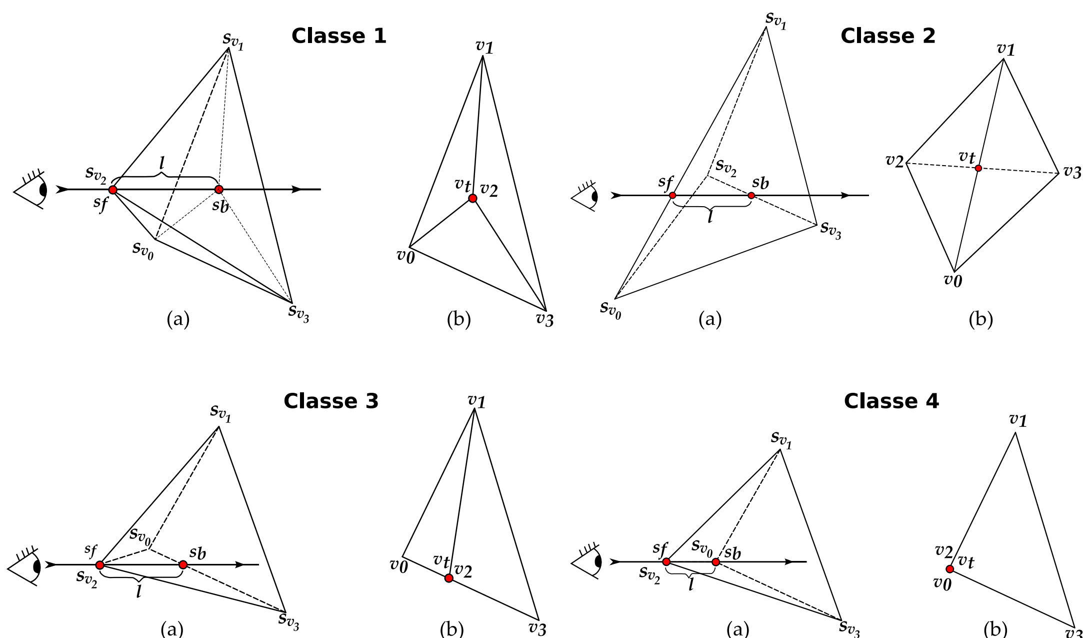


A classificação das células é feita baseada nas normais das faces do tetraedro. A célula é classificada comparando a direção e o sentido das normais em relação ao *vetor de visão*, i.e. vetor do vértice ao ponto de vista. As faces são marcadas com a notação mostrada na Figura da página anterior, dependendo se:

- A normal aponta na direção do ponto de vista: $+$;
- A normal aponta na direção oposta do ponto de vista: $-$;
- Caso a normal seja perpendicular ao vetor de visão: 0.

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

Na Figura abaixo é ilustrado um caso para cada classe de projeção. O tetraedro (a) é mostrado projetado no plano da imagem (b) em cada caso. Os vértices v_i são as projeções dos vértices originais do tetraedro, onde s_{v_i} é o valor escalar do vértice. O vértice espesso é definido como v_t e seus atributos são: a espessura l da célula; os valores escalares de entrada s_f e saída s_b.



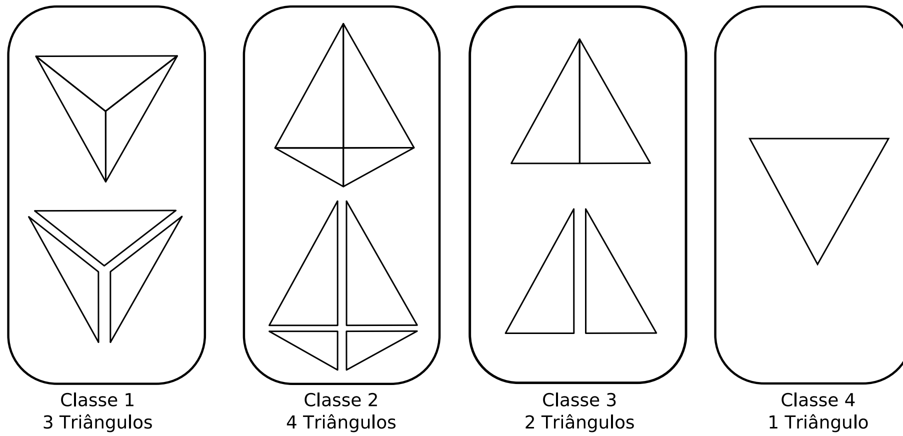
Analisando a Figura acima, para a projeção classe 4, o vértice espesso v_t é definido como v_2 ou v_0, pois esses dois vértices são colineares em relação ao vetor de visão. Em consequência, $s_f = s_{v_2}$, $s_b = s_{v_0}$ e l é igual ao tamanho da aresta dos vértices v_2 e v_0 no tetraedro original,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

Para as outras classes é necessário computar os atributos do vértice espesso v_t. A espessura l é computada realizando a projeção inversa de v_t, encontrando os pontos de entrada e saída, e computando a distância euclidiana desses pontos. No caso da classe 1, uma interpolação bilinear deve ser realizada:

$$v_t = v_0 + u(v_1 - v_0) + t(v_3 - v_0). \tag{C.1}$$

As coordenadas (x, y) de v_t são dadas pela projeção. No algoritmo PT, a Equação acima é utilizada para computar a coordenada z de v_t. Por exemplo, na projeção classe 1 mostrada na Figura da página anterior, a espessura l é computada pela distância entre o vértice do tetraedro original v_2 e a interpolação dos vértices v_0, v_1 e v_3. Os escalares de entrada e saída s_f e s_b do vértice espesso v_t são computados por interpolação dos escalares do tetraedro original.

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



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

Os algoritmos de projeção de células apresentados nesta tese utilizam a ideia do algoritmo PT. No próximo apêndice será apresentado a base da adaptação do algoritmo PT em GPU, o algoritmo PTINT [23].

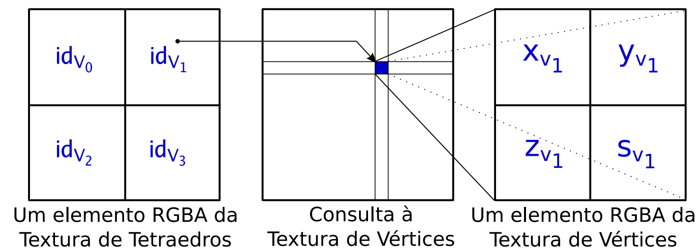
Apêndice D

Algoritmo PTINT

PT with Partial Pre-Integration [23] – http://code.google.com/p/ptint

O algoritmo PTINT forma a base dos algoritmos RPTINT e IPTINT apresentados nesta tese nas Seções 3.2 e 3.3. Este apêndice complementa a explicação apresentada com figuras e tabelas referentes ao PTINT.

Para acessar em GPU as coordenadas dos vértices dos tetraedros de um volume, o algoritmo PTINT usa duas texturas: de *Tetraedros* e de *Vértices*. O acesso à Textura de Vértices é feito usando a Textura de Tetraedros da seguinte forma:



Para classificar a projeção de um tetraedro, o algoritmo PTINT usa quatro testes de produto vetorial (mostrados abaixo). A função *sign* do GLSL [2] é similar à notação empregada pelo PT e retorna -1, 0 ou 1 dependendo se o argumento é menor que, igual a ou maior que zero, respectivamente.

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

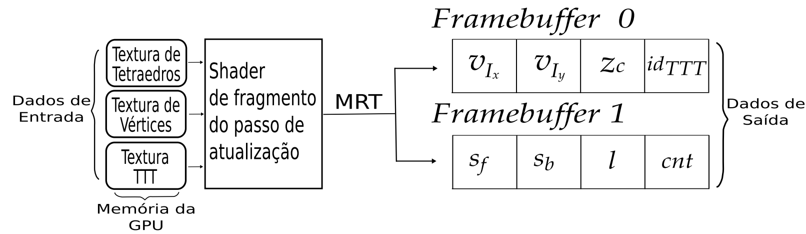
$$\begin{aligned} teste_1 &= sign((vec_1 \times vec_2).z) + 1 \\ teste_2 &= sign((vec_1 \times vec_3).z) + 1 \\ teste_3 &= sign((vec_2 \times vec_3).z) + 1 \\ teste_4 &= sign((vec_4 \times vec_5).z) + 1 \end{aligned}$$

Após os testes de classificação, o algoritmo PTINT usa a tabela TTT (*Ternary Truth Table*), apresentada a seguir, para determinar o caso da classe de projeção.

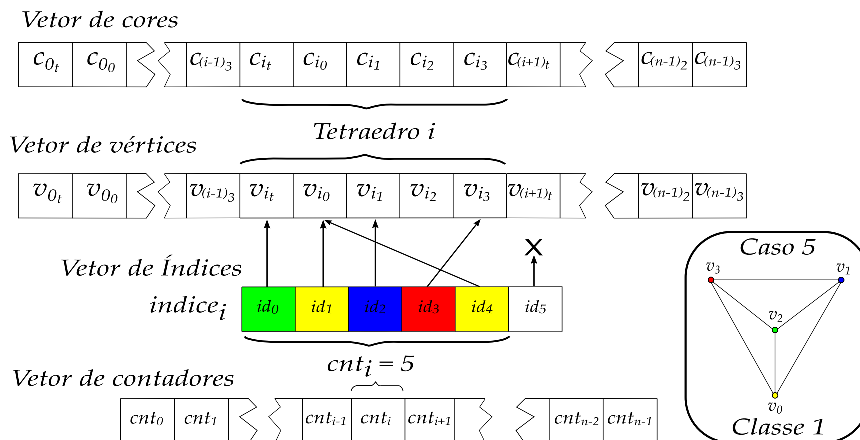
id_{ttt}	$teste_{1234}$	$Caso$	$RGBA$	id_{ttt}	$teste_{1234}$	$Caso$	$RGBA$
0	0 0 0 0	12	0-3-2-1	41	1 1 1 2	–	–
1	0 0 0 1	18	0-3-2-1	42	1 1 2 0	–	–
2	0 0 0 2	6	0-3-2-1	43	1 1 2 1	–	–
3	0 0 1 0	25	1-0-3-2	44	1 1 2 2	49	0-1-2-3
4	0 0 1 1	40	2-3-1-0	45	1 2 0 0	34	3-2-0-1
5	0 0 1 2	23	1-0-2-3	46	1 2 0 1	–	–
6	0 0 2 0	8	0-2-3-1	47	1 2 0 2	–	–
7	0 0 2 1	20	0-2-3-1	48	1 2 1 0	47	0-2-1-3
8	0 0 2 2	14	0-2-3-1	49	1 2 1 1	–	–
9	0 1 0 0	27	2-1-0-3	50	1 2 1 2	–	–
10	0 1 0 1	–	–	51	1 2 2 0	37	3-0-2-1
11	0 1 0 2	–	–	52	1 2 2 1	44	1-2-3-0
12	0 1 1 0	46	0-3-2-1	53	1 2 2 2	35	3-0-1-2
13	0 1 1 1	–	–	54	2 0 0 0	4	0-3-1-2
14	0 1 1 2	–	–	55	2 0 0 1	16	0-3-1-2
15	0 1 2 0	32	2-1-3-0	56	2 0 0 2	10	–
16	0 1 2 1	41	1-3-0-2	57	2 0 1 0	–	–
17	0 1 2 2	30	2-3-1-0	58	2 0 1 1	–	1-2-0-3
18	0 2 0 0	2	1-3-0-2	59	2 0 1 2	21	–
19	0 2 0 1	–	–	60	2 0 2 0	–	–
20	0 2 0 2	–	–	61	2 0 2 1	–	–
21	0 2 1 0	22	1-3-0-2	62	2 0 2 2	1	1-2-0-3
22	0 2 1 1	–	–	63	2 1 0 0	29	2-0-1-3
23	0 2 1 2	–	–	64	2 1 0 1	42	1-3-2-0
24	0 2 2 0	9	0-2-1-3	65	2 1 0 2	31	2-0-3-1
25	0 2 2 1	15	0-2-1-3	66	2 1 1 0	–	–
26	0 2 2 2	3	0-2-1-3	67	2 1 1 1	–	–
27	1 0 0 0	36	3-2-1-0	68	2 1 1 2	45	0-3-1-2
28	1 0 0 1	43	1-2-0-3	69	2 1 2 0	–	–
29	1 0 0 2	38	3-1-2-0	70	2 1 2 1	–	–
30	1 0 1 0	–	–	71	2 1 2 2	28	2-3-0-1
31	1 0 1 1	–	1-3-0-2	72	2 2 0 0	13	0-1-3-2
32	1 0 1 2	48	–	73	2 2 0 1	19	0-1-3-2
33	1 0 2 0	–	–	74	2 2 0 2	7	0-1-3-2
34	1 0 2 1	–	0-2-1-3	75	2 2 1 0	24	1-3-2-0
35	1 0 2 2	33	0-2-1-3	76	2 2 1 1	39	2-3-0-1
36	1 1 0 0	50	0-2-1-3	77	2 2 1 2	26	1-2-3-0
37	1 1 0 1	–	3-2-1-0	78	2 2 2 0	5	0-1-2-3
38	1 1 0 2	–	1-2-0-3	79	2 2 2 1	17	0-1-2-3
39	1 1 1 0	–	3-1-2-0	80	2 2 2 2	11	0-1-2-3
40	1 1 1 1	–	–	–	–	–	–

Na tabela acima, os valores 0, 1 e 2 para as colunas com rótulo $teste_{1234}$ são todos os 81 possíveis resultados dos 4 testes de classificação apresentados na página anterior. Estes resultados formam o índice id_{TTT} da tabela TTT.

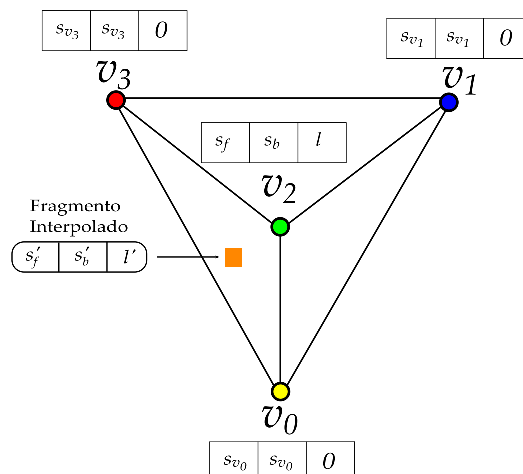
O primeiro passo do algoritmo PTINT é realizado usando shader de fragmento com o seguinte esquema de entrada/saída:



A estrutura de dados principal do PTINT é baseada em vetores de vértices para a função *glMultiDrawElements* do OpenGL [1]. Na Figura abaixo, os índices ilustram o caso 5 (da classe 1), onde a ordem correta para renderizar o tetraedro i é $v_{i_t} - v_{i_0} - v_{i_1} - v_{i_3} - v_{i_0}$. Note que v_{i_2} é o vértice espesso e suas coordenadas são copiadas entre o primeiro e segundo passo do algoritmo para v_{i_t}.



Na Figura abaixo é ilustrado um fragmento de entrada do segundo passo do algoritmo PTINT. Os valores interpolados usam o exemplo classe 1 mostrado na Figura anterior. Note que, exceto pelo vértice espesso, todos os outros vértices são renderizados com os valores originais do volume: s_{v_i}.

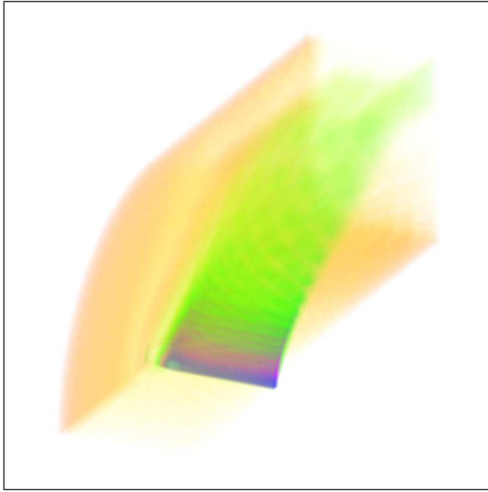


Nas próximas páginas serão apresentadas imagens e medidas de desempenho do algoritmo PTINT. As medidas foram conduzidas em um Intel Pentium IV 3.6 GHz com 2 GB de RAM, utilizando a placa gráfica nVidia GeForce 6800 com 256 MB e barramento PCI Express 16x. A implementação do algoritmo PTINT foi escrita em C/C++ utilizando OpenGL 2.0 com GLSL em Linux.

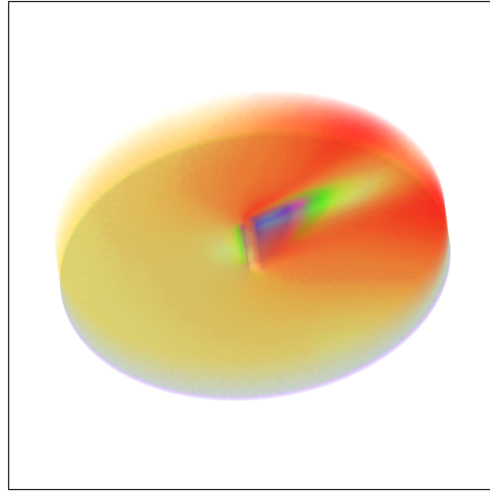
Nas imagens seguintes são exibidos exemplos de renderização de seis dados volumétricos usando o algoritmo PTINT:

- o *Blunt Fin* (a) e o *Oxygen Post* (b) são experiências da interação do oxigênio em um ambiente;
- o *SPX* (c) apresenta áreas de possíveis vazamentos de um reator, enquanto que o *Combustion Chamber* (d) mostra diferentes temperaturas em uma câmara de combustão;
- o *Fuel Injection* (e) simula a injeção de combustível em uma câmara de combustão;
- e o *Brain Tomography* (f) é o resultado de uma tomografia computadorizada do cérebro.

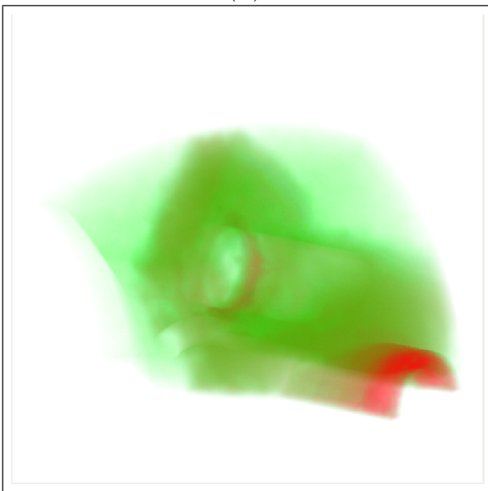
Nas tabelas seguintes, resultados são apresentados envolvendo estes seis volumes. Os resultados visam comparar o algoritmo PTINT com outros algoritmos do estado da arte, sejam baseados em projeção de células ou traçado de raios.



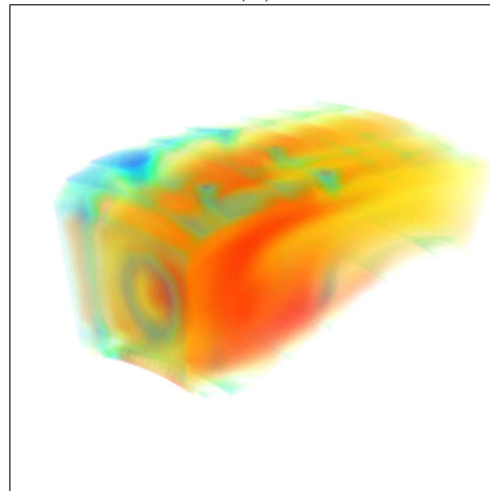
(a)



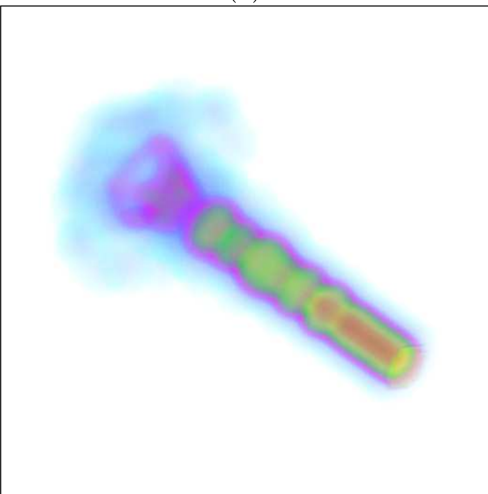
(b)



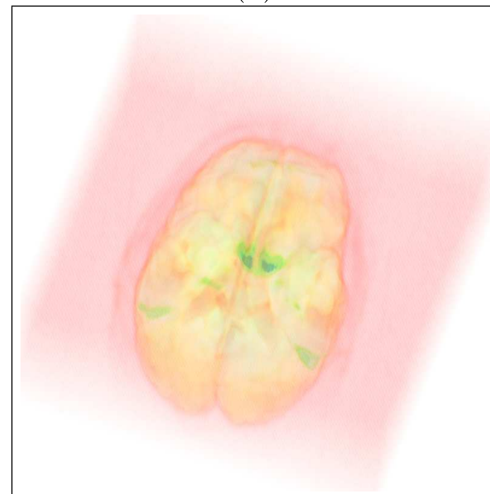
(c)



(d)



(e)



(f)

Na tabela seguinte são apresentados comparações de desempenho do algoritmo PTINT com outros algoritmos de visualização volumétrica. Os seguintes algoritmos de projeção de células e de traçado de raios foram testados: *PTINT* – algoritmo apresentado [55]; *GATOR – GPU Accelerated Tetrahedra Renderer* [8]; *VICP – View-Independent Cell Projection* (implementado em GPU e CPU) [16]; *VICP (Balanced)* – VICP balanceado [11]; *HARC – Hardware-Based Ray Casting* [20]; *HARC (INT) – HARC* com Pré-Integração Parcial [21]; *HAVIS – Hardware-Accelerated Volume and Isosurface Rendering Based on Cell-Projection* [7].

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

Na próxima tabela são apresentados alguns resultados do algoritmo PTINT para diferentes dados volumétricos. Os tempos consideram o volume em constante rotação, e o plano da imagem utilizado para visualização com 512×512 pixels.

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

O algoritmo PTINT (*Projected Tetrahedra with Partial Pre-Integration*) foi publicado com o título *GPU-Based Cell Projection for Interactive Volume Rendering* na conferência internacional SIBGRAPI 2006 [23].

Glossário

A

Absorção mais emissão – Modelo denominado *absortion plus emission* por Max [5].
Análise de Componentes Principais – Método estatístico que projeta um número de variáveis possivelmente correlacionadas em um número menor de variáveis não correlacionadas chamadas de componentes principais (*Principal Component Analysis*).
Anéis de vizinhança – Vizinhança local direta entre vértices ligados por arestas (*neighborhood rings*).
Auto-similaridade – Propriedade de uma malha de possuir regiões similares entre si (*self-similarity*).

B

Base Variedade Harmônica – Método de conversão de geometria de uma malha para espaço de frequência (*Manifold Harmonic Basis.*
Bloco – Relativo a um conjunto de threads (*block*) em CUDA (veja CUDA), concentra os recursos de processamento de um multi-processador na placa gráfica.
Buffer – Espaço de armazenamento temporário para escrita e leitura de dados em trânsito.
Buffer de Dados – Área de memória (*data buffer*) alocada na memória principal compartilhada pela CPU e GPU. Também conhecida por antigos nomes: *vertex buffer object* (VBO) e *pixel buffer object* (PBO).

C

Cache – Memória especializada para reduzir tempo de acesso à memória principal, armazenando dados prováveis de serem requisitados pelo processador mais próximo dele.

Caixa limitante – Retângulo mínimo que contém um objeto (*bounding box*).
Caminho do raio – Intersecções realizadas ao longo de um raio (*ray traversal*) no algoritmo traçado de raios (veja Traçado de Raios).
Célula do volume – Região de valor constante dentro do volume (*volume cell*).
Chip – Circuito eletrônico miniaturizado usado para computar.
Coeficiente de extinção – Valor que exprime a quantidade de luz perdida por um raio ao atravessar uma célula (*extinction coefficient*).
Computação Gráfica – Área da ciência da computação responsável pela geração de imagens computacionais a partir de modelos (*computer graphics*).
Conjunto visível – Conjunto de pixels (veja Pixels) de uma face visível (veja Face visível) do volume (*visible set*).
Corte – Operação de descarte de geometria não visível (*clipping*).
CUDA – Arquitetura unificada de processamento paralelo em GPU (veja GPU) pela nVidia [4] (*Compute Unified Device Architecure*).

D

Dataset – Modelo ou dado em questão.
Dados 4D – Veja Dados dinâmicos.
Dados dinâmicos – Dados volumétricos que variam no tempo (*time-varying data*) também chamados de Dados 4D.
Desvio – Condição de desvio em código (*branch*).
De-frente-para-trás – Relativo a renderizar objetos mais próximos do ponto de vista até os mais distantes (*front-to-back order*).
De-trás-para-frente – Relativo a renderizar objetos mais distantes do ponto de vista até os mais próximos (*back-to-front order*).

E

Elemento de figura – Elemento (por exemplo cor) em uma parte mínima da imagem na tela (veja Quadro).
Elemento de textura – Elemento (por exemplo cor $RGBA$) armazenado em uma textura (*texel ou texture element*).
Elemento de volume – Elemento (por exemplo valor escalar) em uma parte mínima do volume (*voxel ou volume element*).
Entrada/Saída – Denominação referindo à entrada e saída de dados (*input/output* ou I/O).
Escanear – Estrangeirismo derivado da palavra inglesa *scan*, que significa: *esquadrinhar*.

Espaço de imagem – Referente a tela, guiado pelos pixels da imagem.
Espaço de memória – Área de memória utilizada (*memory footprint*).
Espaço de objeto – Referente ao modelo, guiado pelos elementos do dado em questão.
Espessura da célula – Profundidade percorrida pelo raio dentro de uma célula (*thickness*).

F

Face da borda – Face na fronteira ou borda de um modelo (*boundary face*).
Face externa – Face que pertence apenas a uma célula do modelo (*external face*), também chamada de face da borda (veja Face da borda).
Face interna – Face compartilhada por duas células do modelo (*internal face*).
Face visível – Face externa ou da borda (veja Face externa) que está visível dado um ponto de vista (*visible face*).
Fatia – Imagem 2D (*slice*) de uma série de imagens que formam um dado regular, também chamado de imagem 3D.
Fluxo – Entrada/Saída de dados (*stream*) em um kernel (veja Kernel).
Frame buffer – Relativo ao resultado do pipeline gráfico (veja Quadro) enviado à tela ou a um render alvo (*render target*).
Função de bases radiais – Função de valor real que depende somente da distância para o centro (*radial basis function* – RBFs).
Função de transferência – Tabela que relaciona valores escalares a cor e opacidade (*transfer function*).
Função de mistura – Função que determina como os fragmentos serão combinados na geração do pixel final (*blending function*).
Função de profundidade – Função que determina quais fragmentos serão descartados e qual será mantido no pixel final (*depth function*).

G

GATOR – Algoritmo *GPU-Accelerated Tetrahedra Renderer* de Wylie *et al.* [8].
Geodésica – Menor distância entre dois pontos em uma superfície considerando a curvatura da superfície (*geodesic*).
GLSL – Linguagem de programação (*OpenGL Shading Language*) em placa gráfica (veja Shader).
GPGPU – Conceito de programação genérica (*General Purpose GPU*) em GPU (veja GPU).
GPU – Nome da unidade de processamento gráfico, ou placa gráfica (*Graphics*

Processing Unit).
Grade – Matriz (*grid*) de blocos (veja Bloco) em CUDA (veja CUDA) que executa um determinado kernel (veja Kernel).
Grafo base – Classificação isomorfa à projeção de um tetraedro (*basis graph*) no algoritmo GATOR (veja GATOR).

H

HARC – Algoritmo *Hardware-Assisted Ray Casting* de Weiler *et al.* [20].

I

Integral de iluminação – Integral de linha (*volume rendering integral*) para o cálculo de visualização volumétrica (veja Visualização volumétrica).
Intensidade aritmética – Razão entre operações aritméticas e acesso à memória (*arithmetics intensity*) no contexto de programação em GPU (veja GPU).
Iso-superfície – Superfície de mesmo valor escalar dentro de um volume (*iso-surface*).
Iso-valor – Valor escalar (*iso-value*) associado a iso-superfície (veja Iso-superfície).

K

Kernel – Núcleo de processamento paralelo de fluxo (veja Fluxo) em CUDA (veja CUDA).

L

Laplace-Beltrami – Operador linear de geometria diferencial definido pelo divergente do gradiente que forma uma função-base (*function basis*).
Leque de triângulos – Primitiva geométrica (*triangle fan*) que descreve uma série de triângulos conectados por um vértice em comum.
Leque geodésico – Amostras usando uma distância geodésica (veja Geodésica) fixa de um dado vértice (*geodesic fan*).
Linha de execução – Relativo a tarefa (*thread*) mais simples a ser executa em CUDA (veja CUDA).

M

Malha – Descrição geométrica de um objeto 3D por pontos e faces triangulares (*mesh*).
Mapa de curvaturas – Conjunto de valores de curvaturas com o objetivo de descrever uma dada região (*curvature map*).
Mapeamento de textura – Relativo a associar elementos de textura (veja Texel) à uma geometria (*texture mapping*).
Matriz de projeção – Matriz responsável pela projeção dos vértices (*projection matrix*).
Matriz de transformações – Matriz responsável por transformações afins dos vértices (*modelview matrix*): rotação; escala; e translação.
Memória de textura – Memória da placa gráfica (*texture memory*) usada para armazenar texturas.
Montagem de primitivas – Tarefa realizada pela GPU (veja GPU) que agrega vértices formando primitivas (*primitive assembly*), e.g. triângulos.

N

Não-local – Orientação não-local remete a uma orientação global na estruturação de dados de um modelo (*non-local*).

O

Objeto de frame buffer – Relativo a anexar texturas (*color attachments*) ao frame buffer (*frame buffer objects – FBO*).
Ordenação por visibilidade – Técnica para ordernar células do volume de forma a compô-las corretamente na integração para gerar a imagem final. (*visibility ordering*).

P

Phong – O modelo de iluminação de Phong [77] (chamado de *Phong Shading*) refere-se a um modelo matemático que permite estimar a cor de um pixel baseado em contribuições de reflexões difusa e especular, bem como uma parcela devida a iluminação ambiente.
Pixel – Veja Elemento de figura.
Plano de visão – Relativo a imagem final que será gerada (*view plane*).
Ponto de vista – Ponto onde o observador se encontra (*view point*).

Pré-integração – Técnica de pré-computação (*pre-integration*) dos valores de integral de iluminação (veja Integral de iluminação).
Processamento de Malhas – Área da computação gráfica responsável pela manipulação ou análise geométrica de malhas triangulares em modelos 3D (*mesh processing* ou *geometry processing*).
Programa em GPU – Programa (*shader*) a ser carregado e executado em GPU (veja GPU).
Projeção de células – Método de visualização volumétrica que projeta cada célula do volume (*cell projection*) no plano de visão (veja Plano de Visão).
Projeção direta – Outro nome dado (*direct projection*) ao método projeção de células (veja Projeção de células).
PT – Algoritmo *Projected Tetrahedra* de Shirley e Tuchman [6].
PTINT – Algoritmo *Projected Tetrahedra with Partial Pre-Integration* de Marroquim *et al.* [23].

Q

Quadro – Relativo a imagem final gerada (*frame*) pela placa gráfica.
Quadro por segundo – Medida de desempenho de geração de imagens por segundo (*frames per second* ou *fps*).

R

Raio de visão – Raio que parte do observador até o modelo (*viewing ray*).
Rasterizar – Estrangeirismo derivado da palavra inglesa *raster*, que significa: *um padrão de linhas de pontos próximos que formam uma imagem.*
Renderizar – Estrangeirismo derivado da palavra inglesa *render*, que significa: *desenhar.*
Renderização em múltiplos alvos – Opção de renderização de placas gráficas modernas (*multiple render targets – MRT*) onde mais de um frame buffer (veja Frame buffer) pode ser utilizado para renderização.
Re-triangulação – Aplicação que visa refazer a malha triangulada, conhecida como *remeshing.*

S

Scanner – Equipamento de captura de dados em forma de imagem (2D) ou modelo geométrico (3D).

Shader – Programa em GPU no contexto de programação em placa gráfica (veja GPU).
Shader de fragmento – Programa em GPU responsável por operações nos fragmentos ou pré-pixels (*fragment shader*).
Shader de geometria – Programa em GPU responsável por operações nas primitivas (*geometry shader*).
Shader de vértice – Programa em GPU responsável por operações nos vértices (*vertex shader*).
SIMD – Arquitetura de processador onde uma instrução é executada em múltiplos dados (*Single Instruction Multiple Data*).
Somente-leitura – Relativo à forma de acesso que permite apenas leitura da memória (*read-only*).

T

Tabela de dispersão – Tabela onde os dados estão dispersados por uma função de indexação (*hash table*).
Tetraedrizar – O verbo tetraedrizar (usado neste trabalho) significa transformar em tetraedros.
Texel – Veja Elemento de textura.
Thread – Responsável por executar o programa kernel (veja Kernel) compartilhando um micro-processador da GPU.
Traçado de raios – Método de visualização volumétrica que lança, para cada pixel, um raio que atravessa o volume computando a cor final no pixel (*ray casting*).

U

Uniforms – Relativo as variáveis uniformes (veja Variável uniforme) na linguagem GLSL.

V

Variável uniforme embutida – Relativo às variáveis uniformes (veja Variável uniforme) padrão do GLSL (*built-in uniform variables*), como por exemplo a matriz de transformações *glModelViewMatrix*.
Variável uniforme – Variáveis uniformes são armazenadas em registradores somente-leitura compartilhados por todos os processadores da placa gráfica.
Vértice espesso – Ponto projetado (chamado de *thick vertex*) onde um dado raio

percorre a distância máxima dentro do tetraedro.
Vértice fino – Ponto projetado (chamado de *thin vertex*) onde um dado raio percorre distância zero dentro do tetraedro.
Vetor de cor – Vetor otimizado de renderização que armazena as cores dos vértices (*color array*).
Vetor de normal – Vetor otimizado de renderização que armazena as normais dos vértices (*normal array*).
Vetor de vértice – Vetor otimizado de renderização que armazena as coordenadas dos vértices (*vertex array*).
Vetor de visão – Vetor do observador para o modelo (*viewing vector*).
VF-Ray – Algoritmo *Visible Face Driven Ray-Cast* de Ribeiro *et al.* [51].
VICP – Algoritmo *View Independent Cell Projection* de Weiler *et al.* [16].
Visualização volumétrica – Área da computação gráfica responsável pela geração de imagens 2D a partir de dados volumétricos 3D (*volume rendering*).
Visualização volumétrica direta – Outro nome dado (*direct volume rendering*) à área de visualização volumétrica (veja Visualização volumétrica).
Visualização volumétrica indireta – Visualização de iso-superfícies (veja Iso-superfície) do dado volumétrico (*indirect volume rendering*).
Voxel – Veja Elemento de volume.