Multiple-Volume Volumetric Rendering Using Graphic Accelerator

Raphael Leite Serapião
Curso de Ciência da Computação
UNIVALI – Campus Kobrasol
São José/SC - Brasil
raphael.lserapiao@gmail.com

Jeferson Vieira Ramos
Curso de Ciência da Computação
UNIVALI – Campus Kobrasol
São José/SC - Brasil
jefersonvramos@gmail.com

Anita Maria da Rocha Fernandes
Curso de Ciência da Computação
UNIVALI – Campus Kobrasol
São José/SC - Brasil
anita.fernandes@univali.br

***Abstract*—Real-time visualization of volumetric data sets demands high computational performance and large in-memory storage capacity. With the technological advance, the means used to acquire these sets, such as Computed Tomography or Magnetic Resonance Imaging, underwent improvements that resulted in a significant increase of the amount of information collected per sample, making it necessary to develop techniques to generate the visualization of this large quantity of information. The research presents in this paper has the purpose to develop a three-dimensional visualization tool to represent in real time volumetric data with the size of hundreds of megabytes. This tool is based on the ray casting algorithm using graphical acceleration features. To achieve this goal, a methodology was used that involved dividing the volume of data into smaller parts, compatible with the memory limitation of the video hardware. Then the data is used to compose a high quality final image representing the information contained in the volume. In order to evaluate and validate this methodology, comparative tests are performed between the standard rendering procedure and the procedure used in this research.**

Keywords—Three-dimensional visualization, Real time volumetric, Ray casting algorithm, Graphical acceleration features.

I. INTRODUÇÃO

Muitas vezes é necessário analisar uma grande quantidade de informações, ou estudar o comportamento de um determinado processo para reconhecer um padrão ou relação nos dados de uma amostra coletada. Fazer este tipo de análise trabalhando puramente com números pode ser uma tarefa muito cansativa e ineficiente. Entretanto, se de alguma maneira, esses números forem convertidos em uma forma visual, como uma imagem, os padrões e relações se tornam facilmente visíveis. Um exemplo disso pode ser visto na área médica, onde os médicos estão interessados em visualizar e quantificar diferentes estruturas do corpo humano durante a análise das imagens obtidas por Tomografia Computadorizada ou Imageamento por Ressonância Magnética. Atualmente, existe a possibilidade de fundir várias imagens, coletadas em regiões diferentes da amostra, em uma estrutura análoga a um "bloco de dados", que é chamada pela literatura, na maioria das vezes, de volume de dados [1].

De acordo com Bruckner [2], um volume de dados é uma entidade tridimensional (3D) que possui informações em seu interior. Volumes de dados podem ser obtidos por amostragens, simulação numérica, e por técnicas de modelagem. Com o passar dos anos, muitas técnicas foram desenvolvidas para renderização de dados volumétricos. Tradicionalmente a Renderização é uma forma simplificada de mencionar uma série de operações computacionais realizadas para exibir uma imagem tridimensional por meio de uma representação bidimensional (2D), na tela do computador. Atualmente esse conjunto de técnicas é chamado de visualização volumétrica, e se tornou uma importante área de pesquisa dentro do âmbito da visualização científica. Portanto, a visualização volumétrica, dentro do contexto da visualização científica, denota um conjunto de técnicas utilizadas para representar conjuntos tridimensionais de dados em imagens de plano 2D. Para serem proveitosas, essas técnicas devem oferecer representações compreensíveis dos dados, manipulação rápida dos mesmos, e uma renderização rápida [3]. O principal objetivo da visualização volumétrica é mostrar o interior desses volumes a fim de explorar de diversas maneiras a sua estrutura e facilitar sua compreensão [1].

No entanto, as mudanças tecnológicas permitiram que os volumes de dados se tornassem enormes, podendo ocupar vários gigabytes quando carregados em memória, sendo necessário um grande poder computacional para manter a performance interativa das aplicações. Por isso, o desenvolvimento e o refinamento de algoritmos de visualização volumétrica tornaram-se uma importante área de estudo [3].

Dentro deste contexto, este trabalho apresenta como foi tratado o problema, ocasionado quando o tamanho de um conjunto volumétrico de dados excede a capacidade de armazenamento de memória no hardware de vídeo, causando uma redução na performance interativa de aplicações baseadas em visualização volumétrica. Esse problema faz parte de uma área de estudo dentro da visualização científica, que busca métodos para aumentar a velocidade de renderização, considerando o uso eficiente dos recursos de aceleração gráfica. A solução apresentada refere-se a como manter a interatividade em aplicações baseadas em visualização volumétrica, utilizando para isso, métodos de divisão de dados e recursos de aceleração gráfica. Tal solução se baseou na hipótese de que o volume de dados pode ser dividido em partes (blocos) menores, sendo possível enviá-las para o processador gráfico individualmente, e em seguida, compor uma única imagem de alta qualidade representando o conjunto.

Nas próximas sessões serão apresentados os conceitos principais utilizados na pesquisa, a solução desenvolvida e os resultados obtidos.

II. REFERENCIAL TEÓRICO

Visualização Volumétrica

A visualização volumétrica denota um conjunto de técnicas de Computação Gráfica que permitem visualizar o interior de conjuntos tridimensionais de dados e extrair informações significativas de forma interativa. Ela possui numerosas aplicações em áreas como a fluidodinâmica computacional, a biomedicina, geofísica e química computacional. O seu principal objetivo envolve a projeção de um volume de dados em uma imagem de plano 2D, normalmente exibida na tela do computador [4]. De acordo com Kaufman, Cohen e Yagel [5], através dessas técnicas é possível visualizar e manipular de forma interativa as estruturas que estão localizadas no interior do volume, utilizando transformações, cortes, segmentações, translucidez, medições e similares. Com o passar do tempo, muitas técnicas de visualização de volumes foram criadas e atualmente elas se dividem em duas categorias principais chamadas de *surface-fitting* (ajuste de superfície) e *direct volume rendering* (renderização direta de volumes). Na *surface-fitting* estão as técnicas que operam na extração de iso-superfície por meio de primitivas geométricas. E na *direct volume rendering* encontram-se as técnicas que operam gerando a imagem diretamente a partir do volume de dados.

Conforme Kaufman [6], a maior parte dos primeiros métodos de visualização volumétrica utilizavam aproximações baseadas em *surface-fitting*, visto que os algoritmos utilizados nessas categorias são mais simples de implementar e possuem um menor custo computacional. Além disso, existem funcionalidades prontas no processador gráfico para tratar primitivas geométricas. Entretanto, nessa abordagem uma boa parte das informações é perdida por considerar apenas uma superfície. Acrescentando-se a isso alguns fenômenos amorfos como nuvens, fogo e neblina, não podem ser representados adequadamente por meio de superfícies. Em resposta a isso, as técnicas de *direct volume rendering* foram desenvolvidas, com o propósito de capturar todo o conjunto volumétrico de dados em uma única imagem *2D*. A *direct volume rendering* é mais adequada para projetar volumes de dados 3D, e consegue transportar mais informações do volume para a imagem do que as técnicas de *surface-fitting*. Entretanto, existe um custo computacional mais caro neste tipo de abordagem que pode aumentar o tempo de renderização, e por isso, métodos de otimização e hardwares dedicados foram desenvolvidos para aprimorar a interatividade das aplicações. Para este trabalho utilizou-se técnicas de *direct volume rendering*.

Direct Volume Rendering

As técnicas de *direct volume rendering* surgiram a partir de um esforço para representar todos os dados contidos no volume em uma única imagem 2D. Estas técnicas conseguem extrair mais informações e oferecem maior flexibilidade para tratar dados 3D do que as técnicas de *surface-fitting*. Para Bruckner [2], essa categoria de visualização oferece uma grande flexibilidade, pois é possível usá-la para obter uma visão completa de todo o volume de dados, e através do ajuste em uma função de transferência, focar em características particulares dos dados. A função de transferência é aplicada em uma etapa de classificação de dados, onde tem o papel de decidir qual cor e nível de opacidade será atribuído a cada *voxel* com base em seu respectivo valor. Ela torna possível visualizar estruturas localizadas no interior do volume, deixando as camadas externas de materiais translúcidas, criando um efeito análogo a "ver através" do volume.

De acordo com Callahan et al [7], no passado as técnicas de *direct volume rendering* eram lentas e pesadas para serem amplamente utilizadas, porém com os avanços de hardware e software, elas passaram a ser uma categoria de visualização muito atrativa para a comunidade científica. Conforme a literatura [3,6, 7], as técnicas de *direct volume rendering* podem ser divididas em: *object-order*, *image-order* e *domain-based*. Considerando *object-order*, a contribuição de cada *voxel* para formar o *pixel* na tela é calculada, e a união de todas as contribuições constituirá a imagem final. O algoritmo *splatting*, o qual utiliza uma impressão no plano da imagem para cada *voxel*, espalhando a intensidade do valor deste *voxel* sobre uma vizinhança de *pixels*, é um exemplo de *object-order*.

No que se refere ao *image-order*, tem-se algoritmos como o de *ray casting* (utilizado neste trabalho), nos quais os raios são lançados a partir dos *pixels* no plano da imagem em direção ao volume, e a contribuição dos *voxels* ao longo da direção deste raio é usada para compor a cor ao pixel correspondente. Quanto ao *domain-based*, os dados 3D são transformados em outro domínio alternativo, como compressão, *wavelet* ou domínio de frequência, no qual uma projeção é gerada diretamente [8].

Passos Utilizados na Direct Volume Rendering

De acordo com Elvins [3], a maior parte dos algoritmos baseados em *direct volume rendering*, costuma seguir um conjunto similar de passos para alcançar a renderização. Tais passos começam com um fluxo comum que se inicia com a aquisição dos dados de uma amostra e termina com a reprodução de uma imagem final na tela. A seguir são apresentados cada um dos passos.

a) Aquisição de dados: nos casos onde as amostras são obtidas por tomografia computadorizada ou imageamento por ressonância magnética por exemplo, os dados são segmentados em *slices* (fatias), que são planos de seções transversais pertencentes a um objeto. Em seguida é necessário agrupar os *slices* de uma forma que seja possível manipulá-los em conjunto, formando assim um agrupamento volumétrico de dados [3]. Quando o conjunto de dados é construído, a razão das dimensões deve ser proporcional a razão das dimensões do elemento avaliado ou simulado. Isso pode envolver interpolações de valores entre *slices* adjacentes para construir novos, replicações de *slices* existentes ou interpolações para estimar valores faltantes [3].

b) Classificação dos dados: é uma etapa onde funções de transferência são empregadas a fim de mapear os valores dos *voxels* (como a densidade de um tecido do corpo humano, por exemplo) para propriedades visuais, tais como cor e opacidade. Estas funções utilizam o valor encontrado nos *voxels* para gerar a visualização de uma parte específica do volume, de modo que a região de interesse ganhe destaque na imagem final, enquanto as outras regiões fiquem mais translúcidas [1]. Neste trabalho foram usadas as funções de

transferência de opacidade e de cor. As funções de transferência de opacidade especificam a opacidade de acordo com a intensidade do *voxel*. Desta forma é possível determinar quanta energia luminosa é absorvida pelo *voxe*l, e identificar se é possível enxergar através dele ou não. Esta função é aplicada para definir quais estruturas serão transparentes, semi transparentes ou opacas dentro de um volume de dados. Já a função de transferência de cor transforma o valor existente de intensidade do *voxel* em uma cor, já que de uma forma geral, o valor encontrado em um *voxel* não está associado a uma cor específica.

c) Iteração: é a etapa cujo objetivo é compor a imagem que representará a estrutura contida no volume de dados. Essa imagem pode ser criada de duas maneiras: iterando nos *pixels* da imagem pelo método de ordenamento por imagem ou iterando nos *voxels* do volume pelo método de ordenamento por objeto [3]. Este trabalho utilizou o método de ordenamento por imagem. De acordo com Kaufman [6], a iteração por ordenamento de imagem geralmente é executada em uma linha de varredura, onde raios são lançados da posição do observador (câmera) atravessando os *pixels* no plano da imagem na tela. A contribuição dos *voxels* que estão ao longo deste raio é calculada e utilizada para dar cor ao *pixel* atual. Um dos algoritmos mais conhecidos de *direct volume rendering* que utiliza este método é o *ray casting*. Ray et al [4], utilizam a palavra "raio" para descrever uma equação dada por um ponto inicial (câmera do espectador) e uma direção (normal). Quando o raio atingir o volume, a cor do pixel será calculada pelo recolhimento dos valores que estiverem em sua direção em um número finito de posições.

d) Composição: é responsável por somar a contribuição de cor e opacidade das regiões amostradas durante a passagem do raio até atingir a cor final do *pixel* na tela. A composição na ordem frente para trás possibilita que o raio seja encerrado caso o nível de opacidade desejada seja atingido. A composição de trás para frente pode ser utilizada para simplificar os cálculos, entretando não é possível encerrar o raio antes que ele complete seu trajeto. A informação de cor produzida pela composição é mantida em memória (memória de vídeo, no caso de uso de aceleração gráfica), onde os dados de quadros de imagens são enviados para tela por completo [4].

e) Visualização: existem dois métodos de projeção comumente utilizados para exibir a visualização de volumes no escopo da Computação Gráfica: a projeção ortográfica (também chamada de paralela) e a projeção por perspectiva [9]. A maioria dos algoritmos baseados em *direct volume rendering* utiliza a projeção ortográfica ao invés da projeção em perspectiva. Isso ocorre embora a projeção em perspectiva ofereça maior noção de profundidade, na projeção ortográfica, a incerteza de interpretação dos dados é menor porque não exisem as distorções nos dados causadas por uma transformação de perspectivas.

Aceleração Gráfica

A aceleração gráfica em hardware tornou-se um requerimento importante para a maioria das aplicações 3D em tempo real. Historicamente, a aceleração gráfica iniciava apenas no fim do *pipeline* de renderização, durante uma etapa chamada rasterização, usada para determinar os locais dos *pixels* no espaço de tela para cada triângulo. Após sucessivas evoluções no hardware, os aceleradores gráficos 3D evoluiram de um processador de função fixa não programável, para um poderoso processador programável, onde os desenvolvedores podem implementar seus próprios algoritmos para processamento de vértices e fragmentos [10]. O principal objetivo da aceleração gráfica é alcançar a renderização em tempo real, onde as imagens são criadas rapidamente no computador, fazendo com que o espectador se sinta imerso em um processo dinâmico de visualização. A taxa de exibição de uma imagem é mensuada em FPS (*Frames Per Second*, ou Quadros por Segundo), e se refere a quantidade de vezes que o computador projeta essa imagem na tela em um intervalo de um segundo. Uma aplicação gráfica 3D já pode ser considerada em tempo real, caso seja capaz de renderizar imagens em uma taxa acima de 15 FPS. Entretanto, taxas mais altas como 72 FPS são indicadas para que as diferenças de atualização se tornem imperceptíveis ao olho humano [9].

Dentro do contexto da aceleração gráfica, tem-se a GPU (*Graphics Processing Unit*), que é um dispositivo dedicado a renderização gráfica que oferece grande largura de banda de memória e alto poder de processamento. Com o passar do tempo a GPU tornou-se um processador flexível, sendo amplamente utilizada como uma plataforma de aceleração gráfica [11]. Segundo Goodnight, Wang e Humphreys [11], as GPUs atuais superam consistentemente as CPUs em termos de poder computacional. Um dos fatores cruciais para este sucesso é que a arquitetura utilizada nas GPUs foi desenvolvida especialmente para o processamento paralelo de dados. No processamento paralelo, cada elemento de um conjunto de dados é computado concorrentemente, ou seja, o mesmo algoritmo é executado simultaneamente para todos os elementos em um fluxo de dados. Para Owens et al [10], a GPU sempre foi um processador de amplos recursos computacionais. No entanto, segundo o autor, houve um avanço significativo na arquitetura desses processadores, quando eles passaram a passaram a ter a capacidade de processar vértices e fragmentos programáveis. Isso possibilitou que programadores pudessem implementar transformações personalizadas, algoritmos próprios para controle de luminosidade e textura, além de possibilidade de implementar *shaders*. *Shaders* são os meios primários pelo qual as GPUs são controladas. O *vertex shader* possibilita que várias operações (incluindo transformações e deformações) sejam executadas em cada vértice de primitiva. De forma similar, o *fragment shader* processa cada fragmento individualmente, fazendo com que complexas equações sejam calculadas por *pixel* [9]. O *pipiline* de renderização gráfica pode ser considerado o componente central das aplicações gráficas em tempo real [9]. Sua principal função é renderizar uma imagem 2D, dada uma câmera virtual, objetos 3D, fontes de luz, equações, iluminação, texturas, e outros componentes gráficos. A Figura 1 apresenta um modelo de pipeline de renderização gráfica e a sua divisão em três estágios conceituais: aplicação, geometria e rasterização.

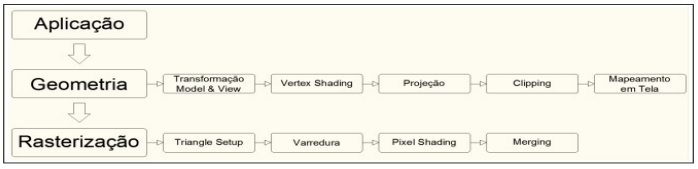

Fig. 1. Modelo de construção básica do *pipeline* de renderização [9].

De acordo com Akenime-Moller, Haines e Hoffman [9], os estágios *do pipeline* de renderização podem ser assim descritos:

a) Estágio de Aplicação: o desenvolvedor tem o controle total do que acontece nesse estágio, e as mudanças realizadas nela podem afetar o desempenho nos estágios subsequentes. Por exemplo, uma configuração no estágio de aplicação pode reduzir o número de triângulos a serem renderizados. O seu principal objetivo é fornecer toda a geometria que será renderizada no estágio seguinte, chamado de geometria. Essa geometria se traduz nas primitivas de renderização, como por exemplo, pontos, linhas e triângulos que possam porventura aparecer na imagem final. Além disso, neste estágio, é feito o tratamento de entradas, como por exemplo, teclado e mouse.

b) Estágio de Geometria: é responsável pela maioria das operações por polígonos e por vértices. Este estágio é dividido em cinco estágios funcionais: *model and view transform* que emprega matrizes de transformação para reposicionar, orientar e projetar objetos em uma cena 3D; *vertex shading* que envolve computar uma equação em vários pontos do modelo, para reproduzir sua representação de forma realista; projeção, cujo objetivo é realizar os cálculos de projeção; *clipping*, cujo principal objetivo é garantir que só as primitivas que estão totalmente ou parcialmente dentro do volume de visualização seja enviadas para o estágio de rasterização; e finalmente *screen mapping*, cujo objetivo é mapear o valor em ponto flutuante das coordenadas x e y de cada vértice para um valor discreto.

c) Estágio de Rasterização: utiliza como entrada os vértices eu foram projetados e transformados no estágio de geometria. Nesse ponto do *pipeline*, os vértices já estão mapeados em coordenadas de tela, e carregam algumas propriedades como cores e coordenadas de um objeto [12]. O estágio de rasterização *é subdividido em: triangle setup, triangle traversal, pixel* shading e *merging* [9].

Pipeline em GPU

A GPU implementa os estágios de geometria e rasterização do *pipeline* conceitual. Estes são também divididos em vários estágios em hardware com níveis diferentes de configuração e programabilidade. A Figura 2 mostra os vários estágios do *pipeline* em GPU com cores de acordo com quão programáveis ou configuráveis eles são. Os estágios são coloridos de acordo com o nível de controle do usuário durante sua operação. Estágios verdes são totalmente programáveis. Estágios amarelos são configuráveis, mas não programáveis. Estágios azuis são complementamente fixos em sua funcionalidade, não permitindo nenhum tipo de controle.

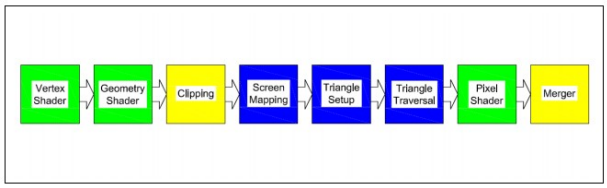

Fig. 2. Implementação em GPU do *pipeline* de renderização.

III. Algoritmos de Visualização Volumétrica

Sistemas de visualização volumétrica são usados para criar imagens de alta qualidade de volumes de dados, usualmente para ao propósito de obter entendimento sobre algum problema científico. No entanto, a maioria das técnicas de visualização de volumes se baseiam em um conjunto de algoritmos fundamentais para renderização. Dentre estes estão o *Opaque Cube*, *Marching Cubes*, *Ray Casting* e *Slatting* [3]. Este trabalho utilizou o algoritmo de *Ray Casting*, o qual será detalhado a seguir.

Ray Casting é um algoritmo para renderização direta de volumes. As técnicas para visualização de volumes através de *Ray Casting* são muito importantes no âmbito da visualização científica e permitem a visualização de dados científicos que podem ser obtidos por diversas maneiras, como por exemplo, tomografia computadorizada ou simulação numérica.

De acordo com Mensmann, Ropinski e Hinrichs [13], o *Ray Casting* emprega usualmente técnicas de programação em GPU em sua implementação. Uma de suas principais vantagens é que tal algoritmo pode ser totalmente implementado e executado usando o estágio de *fragment shader* da GPU, possibilitando resultados interativos para volumes de dados razoavelmente grandes. Além disso, o *Ray Casting* é um algoritmo de renderização direta de volumes baseado na técnica de *image-order*, a qual pode gerar um bom resultado visual e ser facilmente acelerada, como por exemplo, terminando antecipadamente o raio quando a opacidade máxima do fragmento for atingida, ou deixando de lançar raios não irão atingir o volume de dados.

No *Ray Casting* a cor de um *pixel* da imagem final é determinada pelo lançamento de um raio deste *pixel* ao longo de uma direção de visualização, conforme exemplo da Figura 3. Ao longo de cada raio, os valores dos *voxels* são colhidos e processados, em intervalos de espaçamento constante. O processamento de um *voxel* compreende o cálculo necessário para determinar um valor de cor e opacidade que podem ser obtidos pela aplicação de funções de transferência no *voxel* em questão. Dessa forma, a cor e opacidade das amostras colhidas ao longo do raio são acumuladas para obter a cor final do *pixel*. Os parâmetros de iluminação, tais como direção e tipo da fonte de luz, também infuenciam o processamento das amostras [1].

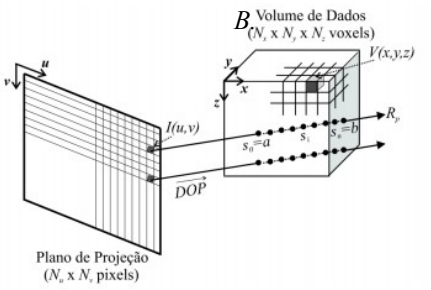

Fig. 3. Exemplo do algoritmo ray casting. Onde (I) - pixel no plano da imagem; (Rp) - raio de projeção; (DOP) - direção de visualização; (V) – voxel; (S) - amostras coletadas ao longo do raio [1].

IV. Recursos de Desenvolvimento

Com o advento do hardware de gráfico programável, várias soluções foram propostas para facilitar o processo de acessar as unidades programáveis da GPU. OpenGL e DirectX são representantes de APIs estabelecidas na programação de hardware gráfico. O OpenGL é uma API aberta compatível com a maioria dos sistemas operacionais modernos. DirectX é uma API proprietária projetada pela Microsoft e oficialmente implementada apenas para sistemas operacionais da família Microsoft. Ambas as APIs fornecem um conjunto de funções para renderização tradicional de gráficos 2D e 3D e aceleração de hardware de gráficos [14].

O recurso computacional presente nas GPUs é acessado via APIs de comunicação, e os programas *shaders* são escritos em linguagens específicas de programação. A seguir serão apresentados os recursos escolhidos para este trabalho.

OpenGL

O OpenGL ou *Open Graphics Library* é uma API para acessar recursos disponíveis no processador gráfico. Essa biblioteca possui comandos para especificar objetos, imagens e operações necessárias para produzir aplicações de Computação Gráfica 3D interativas [15]. Uma das características de design do OpenGL é ser uma interface independente de hardware, a qual pode se comunicar com diferentes tipos de processos gráficos. Não faz parte de seu escopo incluir funcionalidades para abrir ou fechar uma janela de visualização, ou processar entradas de dados, como teclado ou mouse, ao invés disso, sua utilização depende de facilidades providas por outros sistemas para implementar essas funções. Da mesma maneira, o OpenGL não provê funcionalidades para a criação de modelos de objetos 3D, ou métodos para carregamento de imagens. Ao invés disso, o usuário deve implementar funcionalidades para construir seus próprios objetos, ou buscar outras bibliotecas que realizam essas finalidades [15]. Como uma alternativa ao OpenGL existe a biblioteca DirectX. Ambas as bibliotecas possuem a mesma capacidade de se comunicar com o hardware gráfico, e fazem uso do *pipeline* de renderização gráfica tradicional. Entretanto, neste trabalho foi escolhida a biblioteca OpenGL por ser uma API de código aberto, podendo ser executado em múltiplos sistemas operacionais.

GLSL

A OpenGL *Shading Language* (GLSL) é uma linguagem de alto nível, baseada em grande parte na linguagem de programação ANSI C. No âmbito da Computação Gráfica, a linguagem GLSL é empregada para escrever programas que serão executados no hardware gráfico, chamados *shaders*. Como são parte da biblioteca OpenGL, os *shaders* desenvolvidos em GLSL não são aplicativos autônomos, e por isso requerem que a aplicação gráfica em questão, utilize a API do OpenGL para estabelecer a comunicação com a GPU. Portanto, a GLSL é uma linguagem de programação, pelo qual os desenvolvedores que utilizam a biblioteca OpenGL, escrevem programas *shaders* para hardware gráfico [14].

Durante o tempo de execução, os programas GLSL têm acesso automático a uma parte do estado do OpenGL, o qual será afetado por um *shader*. Isso permite que a aplicação gráfica, executada na CPU, use comandos do OpenGL para fazer o gerenciamento de estado dessa biblioteca, e tenha os valores atuais desssas mudanças de estado disponíveis automaticamente para uso de um *shader* na GPU. A GLSL suporta a implementação de ambos os *vertex* e *fragment shaders* [14]. Assim como a GLSL é a linguagem de *shading* para a biblioteca OpenGL, a biblioteca DirectX também utiliza uma linguagem *shading*, chamada *High Level Shading Language*, ou HLSL. Neste trabalho foi utilizada OpenGL, e GLSL foi utilizada para a escrita dos programas *shaders*.

V. Detalhamento do Desenvolvimento

Esta seção apresenta como a pesquisa foi desenvolvida e o desenvolvimento da ferramenta de visualização volumétrica. Primeiramente será apresentada a implementação do algoritmo de *Ray Casting*. Em seguida, cada etapa da implementação do algoritmo será apresentada em detalhe, explicando como o *Ray Casting* foi utilizado em um processo de renderização em etapas e finalmente são apresentados os resultados obtidos, esclarecendo se a técnica foi adequada ou não para alcançar uma performance satisfatória.

Implementação do Algoritmo de Ray Casting

A primeira fase da construção da ferramenta de renderização compreendeu implementar o *Ray Casting* em GPU. Para facilitar a execução deste objetivo, a fase inicial da implementação foi decomposta em atividades: leitura do volume de dados, geração do ambiente tridimensional, lançamento de raios, amostragem e composição. A Figura 4 apresenta estas atividades, desde a leitura do volume de dados até a exibição da imagem final na tela.

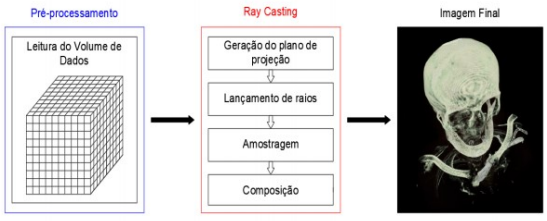

Fig. 4. Fluxo das atividades na implementação do *Ray Casting* em GPU.

A seguir cada uma das atividades será descrita.

a) Leitura do arquivo de Volume de Dados: para efetuar a leitura do arquivo de volume de dados em disco, transferir os dados lidos para memória RAM e interpretar as informações em um formato de volume, foi criado um método para carregar volume de dados. Como parâmetros de entrada, este método recebe o diretório e a dimensão do volume. Como resultado, tem-se um objeto do tipo **Volume** que armazena informações lidas em uma *array* multidimensional (*array* 3D) de *voxels*.

b) Geração do Ambiente Tridimensional: o passo seguinte do desenvolvimento foi criar um ambiente 3D de interação entre o usuário e o mundo virtual. Para que o processo de visualização das estruturas contidas no volume se tornasse mais interativo, a ferramenta foi preparada para permitir que objetos e outras propriedades de visualização pudessem ser modificadas no mundo virtual. Um exemplo disso é a possibilidade de rotacionar o volume de dados ou de movimentar a posição da câmera durante o uso da ferramenta. Diferente de um processo de renderização convencional onde a imagem é construída por polígonos, no *Ray Casting* em GPU o conteúdo exibido no plano de projeção é determinado pelo volume de dados posicionado no centro do mundo com uma câmera em posição arbitrária voltada para ele. O raio é disparado é disparado a partir de cada *pixel* que forma o plano de projeção coletando informações contidas no volume para determinar a cor do *pixel* do qual ele partiu.

Lançamento de Raios: uma vez que o ambiente de interação com o usuário foi construído, a próxima etapa foi implementar o lançamento de raios. No contexto de *Ray Casting*, o termo lançar raio é figurado e representa uma equação para calcular a cor de um *pixel* da imagem final. Dentro do seu intervalo, o raio segue um ponto inicial em um avanço fixo (passo) a cada iteração até alcançar seu ponto final. Neste trajeto ele pode atingir *voxels* posicionados em sua direção e processá-los como uma nova amostra. O intervalo de um raio é calculado pela distância compreendida entre seu ponto de partida e de parada. Para determiná-los, foi implementado um procedimento que utilizou o FBO (*Frame Buffer Object*) para armazenar duas texturas temporárias, chamadas de A e B. Essas texturas foram criadas a partir das faces de um cubo com coordenadas espaciais (0,0,0) e (1,1,1) dentro do mundo 3D.

O FBO é uma memória de vídeo destinada a armazenar imagens de maneira mais leve e rápida, sem que seja necessário exibí-las na tela. Por padrão as cenas renderizadas de uma aplicação gráfica são exibidas diretamente na tela do computador. Entretanto, é possível fazer com que as imagens de uma cena 3D sejam renderizadas para uma textura em memória de vídeo. Este recurso é muito utilizado para gerar efeitos gráficos complexos, onde é necessário renderizar uma cena diretamente em uma textura auxiliar que pode ser usada futuramente em outras operações de renderização. No procedimento para determinar o intervalo dos raios, a aplicação renderizou diretamente para o FBO a face frontal do cubo na textura A e a face traseira do cubo na textura B. As cores desse cubo foram obtidas interpolando os mesmos valores das coordenadas espaciais XYZ para os canais de cores RGB. Cada valor de *pixel* da textura frontal (A) foi usado como uma coordenada de partida de um novo raio, tendo um mesmo *pixel* correspondente em posição na textura traseira (B) como coordenadora de parada. O procedimento para calcular os raios pode ser visualizado na Figura 5. Primeiramente as coordenadas espaciais do cubo foram mapeadas para canais de cores RGB. Após isso, as texturas A e B são renderizadas no FBO, e por último, a direção do raio é determinada pela subtração dos *pixels* da textura A pela textura B.

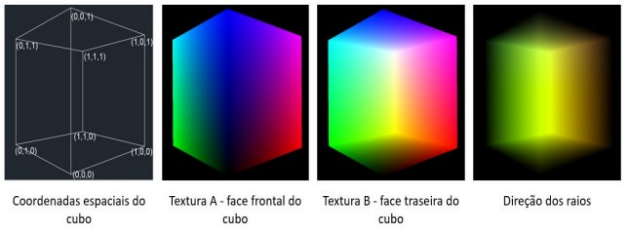

Fig. 5. Procedimento para determinar o intervalo dos raios.

c) Amostragem: durante o curso do raio entre seu ponto inicial até o ponto final, os valores contidos no volume de dados foram coletados em intervalos regulares. A amostragem consistiu em usar a coordenada atual do raio para acessar diretamente o valor do *voxel* no volume de dados. Como visto no referencial teórico, o volume de dados é enviado para a GPU como uma textura 3D podendo ser acessado por meio de coordenadas válidas entre (0,0,0) até (1,1,1). É importante salientar que as coordenadas espaciais do cubo usado na etapa anterior foram definidas como (0,0,0) no ponto mínimo e (1,1,1) no ponto máximo. Isto foi necessário para que os pontos de amostragens também fossem obtidos respeitando os limites de coordenadas válidas para acessar valores na textura 3D. Em cada ciclo de renderização, o raio tem seu início no ponto de partida e segue avançando iterativamente em intervalos definidos em sua direção até que o ponto de parada seja alcançado. Essa iteração pode ser vista na Figura 6, onde a cada repetição, a posição (x, y, z) do raio é somada com um vetor de passo, resultando em nova coordenada para acessar a textura 3D. O vetor passo foi definido pela normal do vetor direção multiplicado por uma constante escalar definida pelo usuário durante a execução da aplicação. Quanto menor for o tamanho do vetor passo, maior será a taxa de amostragem. Uma taxa de amostragem alta permite aumentar a qualidade de visualização dos dados contidos no volume. Entretanto, se essa taxa de amostragem for muito alta, haverá uma queda no desempenho da aplicação, pois a quantidade de informações a serem processadas também será aumentada.

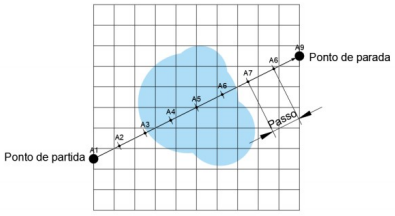

Fig. 6. Amostragem pelo algoritmo de *Ray Casting*.

d) Composição: aqui as cores dos *pixels* foram calculadas usando a contribuição dos *voxels* coletados durante a amostragem. No lançamento de um raio, o valor de cada amostra foi aplicado nas funções de transferência de cor e opacidade, resultando em uma cor RGBA. Esse processamento é realizado iterativamente, e os valores RGBA vão sendo combinados até que o raio chegue ao fim ou até que a opacidade máxima da cor seja atingida. Ambas as funções de transferência de cor e opacidade são texturas unidimensionais (1D). *Pixels* em texturas podem ser acessados por um único índice, e dessa forma, o valor numérico contido na amostra foi utilizado como índice para acessar uma cor RGB na função de transferência de cor e um valor de alfa na função de transferência da opacidade. Essas duas funções de transferência foram enviadas para a GPU como texturas durante o *Ray Casting*. É importante mencionar que o algoritmo de *RayCasting* foi na ordem *front-to-back* (frente para trás), isto é, as amostras foram processadas a partir do primeiro ponto de intersecção do raio com o volume até os pontos seguintes. Entretanto é possível realizar este processamento na ordem *back-to-front* (trás para frente), caso em que as amostras são processadas na ordem inversa A vantagem deste modelo é o ganho de desempenho, pois o raio pode ser terminado quando o valor máximo de opacidade foi atingido.

Implementação do Algoritmo de Ray Casting

Devido à limitação imposta pela capacidade de memória de vídeo, foi necessário tratar adequadamente os dados antes de enviá-los para o *Ray Casting* em GPU. Portanto, a segunda fase do desenvolvimento da ferramenta de visualização correspondeu a implementação de uma técnica de renderização em etapas que possibilitasse renderizar volumes que não podem ser completamente carregados na memória da GPU. A técnica de renderização em etapas é baseada em decompor o grande volume de dados em 8 blocos de volumes menores e enviá-los para o *Ray Casting* em GPU em uma passagem de renderização separada. Todos os blocos foram criados com o mesmo tamanho (exemplo 256^3 *voxels* por bloco) e o resultado do processamento individual de cada bloco é a única textura que representa somente os dados contidos no bloco. Todo esse processo de divisão do volume de dados em sub-volumes pode ser melhor visualizado na Figura 7.

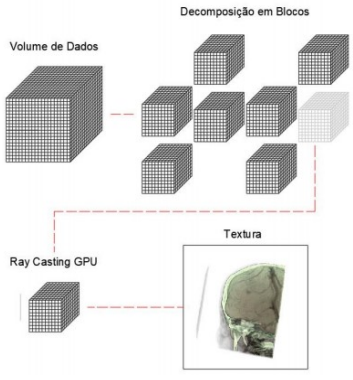

Fig. 7. Divisão do volume de dados na técnica de renderização em etapas.

Com o particionamento dos dados e do processamento, as restrições de memória da GPU podem ser amenizadas, resultando em um ganho de desempenho quando comparado a uma tentativa de enviar para a GPU um volume de dados maior que sua capacidade de memória.

No final de um ciclo de renderização, todas as texturas resultantes são combinadas em um processo de mesclagem, levando em consideração a cor, opacidade e profundidade de cada *pixel* desse conjunto de texturas. O propósito da mesclagem é gerar uma única imagem a partir do conjunto de texturas obtidas pela renderização em etapas.

a) Posicionamento dos Blocos: cada bloco compreende uma região distinta das estruturas contidas no volume de dados. Quando um bloco é enviado separadamente para o Ray Casting *em GPU*, é importante que a textura resultante já seja obtida considerando a posição atual deste bloco no mundo 3D para dar ao usuário a sensação de imersão no ambiente virtual. Se isto não ocorresse, todas as estruturas contidas nos blocos seriam representadas na mesma posição do centro da tela, sobrepondo umas às outras na imagem final. Para tratar essa sobreposição inoportuna, o cubo foi transladado várias vezes para posições diferentes e só depois que suas faces (frontal e traseira) foram renderizadas para o FBO. Dessa forma, foi assegurado que o intervalo de lançamento dos raios correspondessem somente à região do bloco atual, sem interferir nos demais. A Figura 8 apresenta como o cubo de lançamento de raios foi posicionado na cena 3D a fim de corresponder às regiões dos blocos. O conjunto de texturas resultantes não foi exibido na tela porque foi renderizado diretamente no FBO. Somente depois que o processo de mesclagem foi finalizado, uma imagem foi exibida na tela.

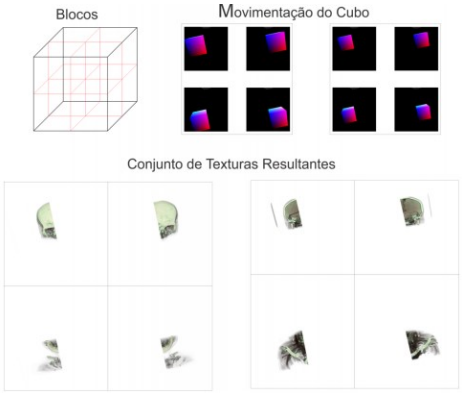

Fig. 8. Posicionamento dos blocos na renderização em etapas.

Além da posição, as dimensões x, y e z do cubo também foram escalonadas para 0,5 vezes em relação ao seu tamanho normal. Isso foi necessário porque um bloco corresponde a apenas 1/8 partes do volume total, e sua representação precisa ser reduzida para tratar regiões de sobreposição incorretas nas estruturas renderizadas para a imagem final.

b) Posicionamento dos Blocos: para combinar as texturas dos blocos e compor uma única imagem final, foi criado um método de mesclagem de texturas na GPU. Nesse método, os *pixels* que formam as texturas resultantes são combinados um a um, de acordo com sua posição e no final

uma nova cor é obtida. Para realizar o cálculo dessa nova cor, o método de mesclagem precisa conhecer a profundidade dos *pixels*, que é distância entre o plano da tela e o fragmento. Isso porque a estruturas dos blocos mais próximos da câmera devem receber opacidade maior do que as estruturas dos blocos mais afastados. Portanto, para determinar a ordem de precedência das cores, a profundidade de cada *pixel* também foi enviada para a GPU. A biblioteca OpenGL oferece uma maneira de armazenar os valores de profundidade em um *buffer* extra, chamado de *depth buffer* (*buffer* de profundidade). O *depth buffer* é uma maneira de manter um registro de profundidade de cada *pixel* na tela. Esse recurso foi usado na ferramenta de visualização para mapear a profundidade dos *pixels* em texturas auxiliares no FBO. Para ordenar os *pixels* pela profundidade, o algoritmo de ordenamento *bubble sort* foi implementado na GPU. Em seguida, as cores foram combinadas de forma que as estruturas mais próximas do observador ganhassem maior opacidade. A Figura 9 apresenta como as texturas resultantes dos blocos foram mescladas em uma única imagem final.

Fig. 9. Mesclagem das texturas na renderização em etapas.

VI. Resultados

Para comparar os resultados de desempenho entre a implementação do *Ray Casting* padrão e renderização em etapas, foram realizados testes submetendo essas duas aproximações nas mesmas condições de sobrecarga. Para isso, volumes de tamanhos maiores e menores que a capacidade de memória da GPU foram utilizados. A performance de cada aproximação foi medida em FPS (*frames per second*), ou seja, quantos quadros foram exibidos na tela em um intervalo de 1 segundo.

Nos testes de avaliação de desempenho considerando as duas situações, o ambiente de testes foi assim composto: processador Intel Core I5 2.60 GHz; memória RAM de 8,00 GB DDR3; GPU NVIDIA GTX 950M – 2,00 GB RAM; e sistema operacional Linux Ubuntu Desktop 16.04.3 LTS 64 bits.

Escolheu-se como base de testes o volume de dados chamado MANIX que foi fornecido gratuitamente pela OSIRIX *data sets*. Tal volume é construído a partir do empilhamento de 460 imagens de tamanho 512 x 512 em escala de cinza. Cada *voxel* deste volume teve a intensidade dada em *unsigned short int* ocupando 2 *bytes* de memória. O volume totalmente carregado ocupou 241 MB de espaço na memória principal.

Com o intuito de sobrecarregar a memória da GPU, foi criado um método de aumento do tamanho do volume de dados. Este método é chamado uma única vez no início da execução após a leitura do volume de dados ser feita para a memória principal. Dessa forma, foi possível aumentar o tamanho do volume conforme as proporções descritas na Tabela 1.

TABELA I. Escalonamento do volume de dados

Taxa de Reescalonamento	Largura	Altura	Profundidade	Número Total de Voxels	Memória Alocada
1.25x	640	640	575	235520000	471 MB
1.5x	768	768	690	406978560	813 MB
1.75x	896	896	805	646266880	1292 MB
2x	1024	1024	920	964689920	1929 MB
2.25x	1152	1152	1035	1373552640	2747 MB
2.5x	1280	1280	1150	1884160000	3768 MB

Durante os testes com sobrecarga foi verificado que o *driver* de vídeo faz o gerenciamento do uso de memória abstrata, por vezes tratando a memória da GPU como um cache da memória principal. Isso foi identificado quando um comando para copiar os dados do volume para a GPU foi executado, mas esses dados foram copiados primeiramente para a memória principal. Somente quando os dados eram requeridos para a renderização é que eles eram copiados da RAM principal para a RAM da GPU. Dessa forma, foi constatado que se a quantidade de dados não pudesse ser alocada na RAM da GPU, esses dados ficavam em uma zona de *swap* (troca) entre as duas memórias.

Os testes de desempenho com o *Ray Casting* padrão foram realizados com diferentes tamanhos de volume. Os resultados mostraram que esta abordagem foi sempre mais veloz que a renderização em etapas. Como desvantagem, o *Ray Casting* padrão não conseguiu renderizar volumes tão grandes quanto a renderização em etapas. Isso foi verificado em uma tentativa de renderizar um volume com tamanho de 3.768 MB. O gerenciamento do *driver* de vídeo fez com que esse volume ficasse na zona de *swap* entre a memória principal e a memória da GPU. Isso gerou um grande consumo de memória principal, pois os dados tiveram que ser constantemente trafegados pelo barramento para serem enviados a GPU. Dessa forma, o próprio sistema operacional parou de realizar suas funções e teve que ser reiniciado. A taxa de desempenho do *Ray Casting* padrão para cada volume de dados pode ser vista na Figura 10. Nas figuras 11, 12 e 13 o resultado do processo de renderização é exibido utilizando três ajustes de função de transferência diferentes.

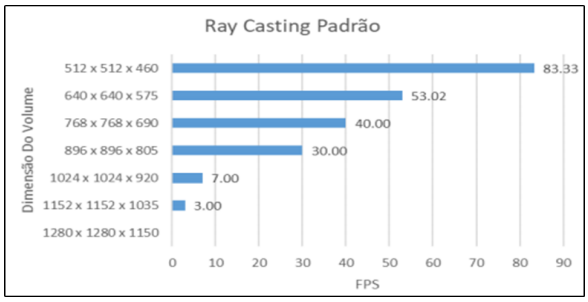

Fig. 10. Desempenho do *Ray Casting* padrão para diferentes volumes.

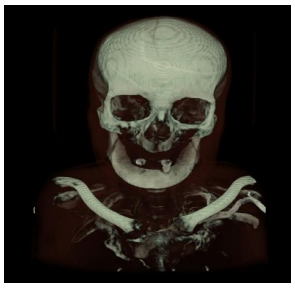

Fig. 11. Resultado obtido pela aplicação do *Ray Casting* padrão usando uma função de transferência para visualizar pele e ossos.

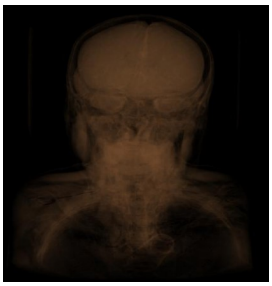

Fig. 12. Resultado obtido pela aplicação do *Ray Casting* padrão usando uma função de transferência para visualizar tecidos.

Fig. 13. Resultado obtido pela aplicação do *Ray Casting* padrão usando uma função de transferência para visualizar ossos.

Os testes de desempenho da renderização em etapas foram realizados com os mesmos volumes e recursos de hardware usados nos testes do *Ray Casting* padrão. Como cada volume foi fragmentado em oito blocos menores, o tamanho do bloco gerado é diretamente proporcional ao tamanho do volume pelo qual ele foi obtido. A técnica de renderização em etapas envolve ter que enviar todos os blocos separadamente para GPU a cada FPS. Quando um bloco é enviado, seus dados são transmitidos por um barramento entre a CPU e a GPU, portanto, quanto maior for o tamanho do bloco mais dados serão trafegados e mais tempo será gasto até que um novo bloco possa ser enviado. O preço para enviar todo o conjunto de blocos foi alto para a renderização em etapas, isso porque muito tempo foi dedicado apenas para transferir e apagar dados da GPU a cada ciclo de renderização.

Além do tempo gasto com o envio de blocos, a renderização em etapas exigiu mais tempo durante a etapa de mesclagem. Na mesclagem os *pixels* nas texturas resultantes são primeiramente classificados de acordo com sua profundidade para depois serem combinados até formar uma cor final. No entanto, o tempo gasto para realizar a mesclagem foi sempre o mesmo para todos os volumes, pois a dimensão e a quantidade de texturas são sempre iguais. A taxa de desempenho da renderização em etapas para cada volume pode ser vista na Figura 14. Nas figuras 15, 16 e 17 o resultado do processo de renderização é exibido usando três ajustes de função de transferência diferentes.

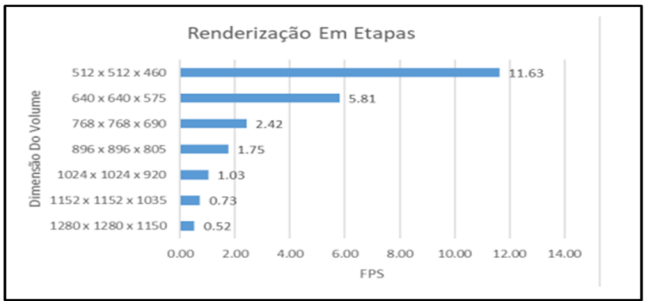

Fig. 14. Desempenho do *Ray Casting* padrão para diferentes volumes.

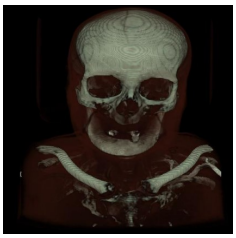

Fig. 15. Resultado obtido pela aplicação da renderização em etapas usando uma função de transferência para visualizar pele e ossos.

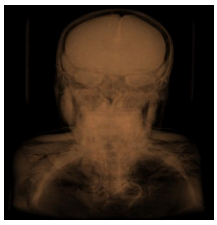

Fig. 16. Resultado obtido pela aplicação da renderização em etapas usando uma função de transferência para visualizar tecidos.

Fig. 17. Resultado obtido pela aplicação da renderização em etapas usando uma função de transferência para visualizar ossos.

Uma vez que os procedimentos *Ray Casting* padrão e renderização em etapas foram implementadas, comparou-se os dois resultados. Os critérios usados para a comparação foram a fidelidade das imagens obtidas, a taxa de FPS e o tamanho máximo do volume suportado por cada uma das abordagens.

Em relação a fidelidade de representação das imagens do volume, notou-se uma pequena diferença entre as duas abordagens. Tal diferença ocorre porque no *Ray Casting* padrão a precisão do cálculo de opacidade é maior, pois o raio sempre é terminado quando o valor máximo de opacidade é atingido. Já na renderização em etapas, o raio também é terminado dessa maneira, mas há uma pequena sobra porque os raios são lançados através de blocos e não pelo volume inteiro. Dessa forma, toda vez que um novo bloco é processado, o valor de opacidade é reiniciado para zero novamente.

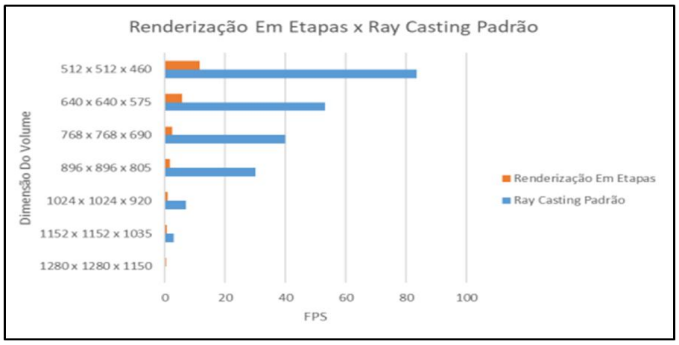

Fig. 18. Comparação dos resultados.

Outro ponto importante verificado foi que o *drive*r de vídeo disponibilizado pelo fabricante da GPU (NVIDIA) otimizou o tráfego de dados pelo barramento e impediu o estouro da memória de vídeo. Isso foi percebido durante os testes do *Ray Casting* padrão quando um comando era executado para forçar o carregamento de um volume de dados maior que a capacidade de memória da GPU e a aplicação continuava a funcionar. Suspeita-se que isso ocorre porque quando os dados são muito grandes, o *driver* de vídeo opera colocando as duas memórias (CPU e GPU) em um estado de *swap* onde os dados de renderização são copiados para a memória principal e a memória da GPU passa a funcionar como uma memória de cache. Em relação à comparação dos tamanhos de volume suportados pelas aproximações, no último teste realizado, um volume de 1280 x 1280 x 1150 *voxels* foi processado e somente a renderização em etapas foi capaz de renderizá-lo. Não foi possível determinar com exatidão a causa deste fato, mas suspeita-se que, embora as duas aproximações façam a leitura e armazenamento do volume para a memória principal de forma igual, durante o processo de renderização os dados do volume são acessados e copiados para serem enviados para a GPU. A vantagem da renderização em etapas neste caso é que essa cópia é feita em cima de um bloco, enquanto que no *Ray Casting* padrão o volume inteiro é copiado novamente gerando o estouro da memória RAM principal.

VII. Conclusões

A otimização de técnicas para visualização de conjuntos de dados volumétricos é uma importante área de pesquisa dentro da visualização científica. Uma ferramenta interativa que permita visualizar grandes volumes de dados pode trazer resultados que beneficiam áreas importantes da sociedade.

No trabalho aqui apresentado foi abordado especificamente o problema ocasionado quando o tamanho de um volume de dados excede a capacidade de armazenamento de memória no hardware de vídeo, causando uma redução na performance interativa da aplicação. A estratégia adotada para tratar esse problema foi baseada na hipótese de dividir o grande volume de dados em blocos menores e enviá-los para o processador gráfico individualmente, evitando que a memória de vídeo trabalhe sobrecarregada. A fim de avaliar e validar a metodologia proposta foram realizados testes onde grandes volumes foram carregados e renderizados em uma plataforma de simulação desenvolvida exclusivamente para esse fim. Os resultados dos testes mostraram que a performance da renderização em etapas ficou bastante limitada quando comparada com o procedimento padrão de renderização, até mesmo para os volumes maiores que ocupavam quase o dobro da capacidade da GPU. Na renderização em etapas muito tempo é gasto somente para trafegar dados enviando todos os sub-volumes para a GPU a cada ciclo de renderização. No *Ray Casting* padrão isso não ocorre porque os dados do volume são enviados em uma única vez, logo no início da aplicação. Quando os dados são grandes demais para a GPU, o próprio driver da GPU otimizou o gerenciamento das duas memórias e impediu o estouro da memória de vídeo. A metodologia desenvolvida não se mostrou totalmente adequada para resolver o problema de baixo desempenho na renderização de grandes volumes de dados. Porém, todo o desenvolvimento deste trabalho constitui um estado inicial para implementar novas melhorias a partir do que foi construído.

Referências

[1] I. H. Manssour, "Visualização de Estruturas Internas em Volumes de Dados Multmodais," 135 f. Tese (Doutorado) – Curso de Ciência da Computação, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2002.

[2] S. Bruckner, "Efficient Volume Visualization of Large Medical Datasets", 111f. Dissertação (Mestrado) – Curso de Ciência da Computação, Institute of Computer Graphics and Algorithms, Viena University of Technology, 2004.

[3] T.T. Todd, "A Survey of Algorithms for Volume Visualization," in SIGGRAPH Computer Graphics, New York, US, v. 26 n. 3. P. 194-201, ago. 1992.

[4] H. Ray et al, "Ray Casting Architecture for Volume Visualization" IEEE Transactions on Visualization and Computer Graphics, vol. 5, pp. 210–223, July, 1999.

[5] A. Kaufman, D. Cohen, R. Yagel, "Volume Graphics" Computer, vol. 26, n. 7, pp. 51-64, July 1993.

[6] A. Kaufman, "Volume Visualization: tutorial". California: IEEE Computer Society Press, 1991.

[7] S. Callahan et al, "Direct Volume Rendering: a 3D plotting technique for scientific data". Computing in Science Engeneering, v. 1o. n.1, p. 88-92, jan. 2008.

[8] D. Cohen, A. Shaked. "Photo-realistic Imaging od Digital Terrains" Computer Graphics Forum, v. 12, n.3, p. 363-373, ago. 1993.

[9] T. Akenine-Moller, E. Haines. N. Hoffman "Real-Time Rendering" 3. ed. Natick, Ma, USA: A.K. Peters, 2008.

[10] J. Owens et al, "GPU Computing" Proceedings of IEEE, v. 96, n. 5, p. 879-899, mai. 2008.

[11] B. Neelima, R. Prakash S Raghavendra. "Recent Trends in Software and Hardware for GPGPU Computing: A Comprehensive Survey" 2010 5th International Conference on Industrial and Information Systems, ICIIS 2010, Jul 29 - Aug 01, 2010, India.

[12] Chih-Hao Sun. You-Ming Tsao. Ka-Hang Lok. "Universal Rasterizer with edge equations and tile-scan triangle traversal algorithm for graphics processing units". IEEE International Conference on Multimedia Proceedings: IEEE, jun., 2009.

[13] J. Mesmann; T. Ropinski, K. Hinrichs. "An advanced volume ray casting technique using GPU stream processing". International Conference on Computer Graphics Theory and Applications. Angers INSTICC Press, 2010.

[14] R. Borgo, K. Brodlie. State of Art Report on GPU Visualization. Visualization & Virtual Reallity Research Group: School of Computing. University of Leeds, 2009.

[15] D. Shreiner et al. Open GL Programming Guide: the official guide to learning OpenGL. 8. ed. Michigan: Pearson, 2013.