Pontifícia Universidade Católica
do Rio de Janeiro

Bernardo Bianchi Franceschin

Visualização de seções de corte arbitrárias de malhas não estruturadas

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós–graduação em Informática do Departamento de Informática da PUC–Rio

Orientador: Prof. Waldemar Celes Filho

Rio de Janeiro
Abril de 2013

Pontifícia Universidade Católica
do Rio de Janeiro

Bernardo Bianchi Franceschin

Visualização de seções de corte arbitrárias de malhas não estruturadas

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós–graduação em Informática do Departamento de Informática do Centro Técnico Científico da PUC–Rio. Aprovada pela comissão examinadora abaixo assinada.

Prof. Waldemar Celes Filho
Orientador
Departamento de Informática — PUC–Rio

Prof. Marcelo Gattass
Departamento de Informática – PUC–Rio

Prof. Luiz Henrique Figueiredo
Instituto Nacional de Matemática Pura e Aplicada (IMPA)

Prof. Hélio Côrtes Vieira Lopes
Departamento de Informática – PUC–Rio

Prof. José Eugênio Leal
Coordenador Setorial do Centro Técnico Científico — PUC–Rio

Rio de Janeiro, 9 de Abril de 2013

Bernardo Bianchi Franceschin

Ficha Catalográfica

Franceschin, Bernardo

Visualização de seções de corte arbitrárias de malhas não estruturadas / Bernardo Bianchi Franceschin; orientador: Waldemar Celes Filho. — Rio de Janeiro : PUC–Rio, Departamento de Informática, 2013.

v., 53 f: il. ; 29,7 cm

1. Dissertação (Mestrado em Informática) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui referências bibliográficas.

1. Informática – Tese. 2. Seção de corte. 3. Malhas não estruturadas. 4. Programação em GPU. I. Celes Filho, Waldemar. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Agradecimentos

Aos meus pais e minha família.

Ao meu Orientador Waldemar, por ter me dado a oportunidade de realizar este trabalho e confiar no meu potencial.

Aos meus amigos César Palomo, Chrystiano Araújo, Eduardo Ceretta, Fábio Miranda, Frederico Abraham e Rodrigo Espinha.

A CAPES, PUC-Rio e Tecgraf pelos auxílios concedidos que viabilizaram este trabalho.

Resumo

Franceschin, Bernardo; Celes Filho, Waldemar. **Visualização de seções de corte arbitrárias de malhas não estruturadas**. Rio de Janeiro, 2013. 53p. Dissertação de Mestrado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Na visualização de campos escalares de dados volumétricos, o uso de seções de corte é uma técnica eficaz para se inspecionar a variação do campo no interior do domínio. A técnica de visualização consiste em mapear sobre a superfície da seção de corte um mapa de cores, o qual representa a variação do campo escalar na interseção da superfície com o volume. Este trabalho propõe um método eficiente para o mapeamento de campos escalares de malhas não estruturadas em seções de corte arbitrárias. Trata-se de um método de renderização direta (a interseção da superfície com o modelo não é extraída) que usa a GPU para garantir bom desempenho. A idéia básica do método proposto é utilizar o rasterizador da placa gráfica para gerar os fragmentos da superfície de corte e calcular a interseção de cada fragmento com o modelo em GPU. Para isso, é necessário testar a localização de cada fragmento na malha não estruturada de maneira eficiente. Como estrutura de aceleração, foram testadas três variações de grades regulares para armazenar os elementos (células) da malha, e cada elemento é representado pela lista de planos de suas faces, facilitando o teste de pertinência fragmento-elemento. Uma vez determinado o elemento que contém o fragmento, são aplicados procedimentos para interpolar o campo escalar e para identificar se o fragmento está próximo à fronteira do elemento, a fim de representar o aramado (wireframe) da malha na superfície de corte. Resultados obtidos demonstram a eficácia e a eficiência do método proposto.

Palavras–chave

Seção de corte ; Malhas não estruturadas ; Programação em GPU.

Abstract

Franceschin, Bernardo; Celes Filho, Waldemar (advisor). **Visualization of arbitrary cross section of unstructured meshes**. Rio de Janeiro, 2013. 53p. MSc. Dissertation — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

For the visualization of scalar fields in volume data, the use of cross sections is an effective technique to inspect the field variation inside the domain. The technique consists in mapping, on the cross section surfaces, a colormap that represents the scalar field on the surface-volume intersection. In this work, we propose an efficient method for mapping scalar fields of unstructured meshes on arbitrary cross sections. It is a direct-rendering method (the intersection of the surface and the model is not extracted) that uses GPU to ensure efficiency. The basic idea is to use the graphics rasterizer to generate the fragments of the cross-section surface and to compute the intersection of each fragment with the model. For this, it is necessary to test the location of each fragment with respect to the unstructured mesh in an efficient way. As acceleration data structure, we tested three variations of regular grids to store the elements (cells) of the mesh, and each element is represented by the list of face planes, easing the in-out test between fragments and elements. Once the element that contains the fragment is determined, it is applied procedures to interpolate the scalar field and to check if the fragment is close to the element boundary, to reveal the mesh wireframe on the surface. Achieved results demonstrate the effectiveness and the efficiency of the proposed method.

Keywords

Cross sections; Unstructured meshes; GPU programming.

Sumário

1
Introdução

Na visualização de campos escalares de dados volumétricos, como dados médicos e resultados de simulações de elementos finitos, o uso de superfícies de corte é uma técnica eficaz para se inspecionar a distribuição do campo no interior do domínio.

A técnica consiste em mapear sobre a superfície renderizada o mapa de cores que representa o campo escalar que incide na interseção da superfície com o volume. Aplicar esta técnica a dados regulares é simples e eficiente, pois devido a natureza do dado, este pode ser diretamente armazenado em uma textura 3D e o cálculo de interseção entre a superfície de corte e o volume do campo escalar é reduzido a atribuir coordenadas de textura aos vértices da superfície renderizada que estejam dentro do domínio da textura 3D (3).

Já uma aplicação voltada para dados representados por malhas não estruturadas, tais como modelos de elementos finitos, se mostra mais desafiadora. Uma implementação que realiza o cálculo da interseção em CPU envolve a construção da geometria da interseção que deverá ser reconstruída toda vez que a posição relativa entre a superfície de corte e o volume do campo escalar mudar. Essa abordagem pode se mostrar ineficiente para cenas mais complexas que apresentem muitas superfícies de corte ou em que superfícies se movam em relação ao volume de maneira frequente.

O objetivo deste trabalho é apresentar um método eficiente para o mapeamento de campos escalares de malhas não estruturadas em superfícies de corte arbitrárias. Trata-se de um método de renderização direta (a interseção da superfície com o modelo não é extraída) que usa a GPU para garantir bom desempenho.

A idéia básica do método proposto é utilizar o rasterizador da placa gráfica para gerar os fragmentos da superfície de corte e calcular a interseção de cada fragmento com o modelo em GPU, utilizando o estágio programável de fragmentos do pipeline gráfico (*fragment shader*). Dessa maneira, o cálculo de interseção consiste em determinar o elemento da malha não estruturada que contém cada fragmento da superfície de corte. Para que esse processo ocorra de maneira eficiente, foi necessário descobrir uma maneira rápida de testar a interseção entre ponto (fragmento) e um elemento da malha, fazendo uso de uma estrutura de aceleração que diminua o número de testes de interseção por fragmento e que ainda possa ser armazenada na GPU. A solução proposta para o teste de interseção consiste em mudar a maneira como os elementos da malha

são representados. Em vez de serem representados por uma lista de vértices, os elementos passaram a ser representados por uma lista de planos. Logo para verificar a colisão entre o fragmento e um determinado elemento, basta testar o fragmento contra os planos do elemento. Como estrutura de aceleração foram desenvolvidas três variações de grades regulares. Assim os elementos da malha são posicionados nas células da grade de maneira que o teste de interseção seja realizado apenas com os elementos presentes dentro de uma mesma célula da grade. Além disso, foi desenvolvido um método para interpolar o valor do campo escalar no fragmento e um método para mapear o aramado (*wireframe*) da malha na superfície de corte.

Este trabalho é voltado para visualização de seções de corte em modelos de simulação numérica. Métodos numéricos são largamente utilizados em simulações para diversos tipos de aplicação. Os detalhes desses métodos podem variar bastante mas todos essencialmente são caracterizados pela discretização do domínio do problema em uma coleção de elementos e pela construção de uma solução global aproximada que é especificada em termos de uma série de aproximações locais. Esta coleção de elementos é o que chamamos de modelo de simulação numérica. Os tipos de modelos utilizados neste trabalho foram modelos de elementos finitos (elementos tetraédricos e hexaédricos) e modelos de diferenças finitas (elementos hexaédricos). Os elementos de um modelo consistem em listas ordenadas de nós. Os nós e, por consequência, as faces são compartilhadas por elementos adjacentes. O método proposto abrange modelos de elementos lineares, isso quer dizer que os elementos não apresentam nós intermediários e o cálculo do campo escalar no interior de um elemento é efetuado através da interpolação linear do valor do campo nos nós do elemento. Assim os elementos considerados pelo método são tetraedros compostos por 4 nós e hexaedros compostos por 8 nós.

Resultados obtidos em diferentes superfícies de corte aplicadas em diferentes modelos demostram a eficácia e eficiência da técnica proposta.

Esta dissertação está organizada da seguinte forma: O Capítulo 2 discute brevemete trabalhos que se relacionam com esta dissertação. O método proposto é explicado no Capítulo 3. No Capítulo 4 é feita uma análise de desempenho do método utilizando três diferentes estruturas de aceleração. O Capítulo 5 apresenta brevemente alguns exemplos de aplicações do método proposto. O Capítulo 6 conclui o trabalho e apresenta as considerações finais.

2
Trabalhos Relacionados

O uso de superfícies de corte, geralmente na forma de planos de corte, está presente em diversos sistemas de visualização. A implementação mais comum consiste na construção em CPU da geometria que representa a interseção entre a superfície e o domínio do dado visualizado. Em (3) é apresentada uma forma simples de se utilizar programação em *shaders* para realizar a renderização direta de superfícies arbitrárias de corte em dados regulares. O dado regular é armazenado em uma textura 3D e para que o corte seja visualizado é necessário apenas atribuir aos fragmentos coordenadas de textura que estejam dentro do domínio da textura 3D. Os trabalhos de modelos de malhas não estruturadas que se propõem a realizar em GPU o cálculo de interseção entre a superfície de corte e o modelo são voltados para a visualização de campos escalares não lineares (*high order*) em modelos de tetraedros. A principal motivação destes trabalhos em utilizar programação em placa gráfica é avaliar o valor do campo escalar do modelo a nível de pixel, evitando assim o uso das primitiva lineares do pipeline convencional e gerando imagens que representem melhor campos escalares não lineares. Em (4), a visualização é restrita a cortes planares, uma aproximação da interseção entre o plano de corte e o modelo é calculada na CPU e é feito no *shader* de fragmentos o cálculo do valor do campo escalar na posição de cada fragmento. Em (16, 17), a renderização de superfícies arbitrárias de corte (superfícies implícitas e paramétricas) é feita através de um algoritmo de traçado de raio em GPU. A implmentação utiliza o *framework* de traçado de raio OptiX da Nvidia e o modelo é estruturado em GPU na forma de um grafo. O método proposto neste trabalho é voltado para a visualização de superfícies de corte arbitrárias em modelos de tetraedros lineares e de hexaedros lineares. A visualização é feita utilizando o pipeline gráfico convencional para rasterizar a superfície de corte e calcular a interseção entre a superfície de corte e o modelo inteiramente no *shader* de fragmentos. Dessa maneira, o teste de interseção é reduzido a identificar qual elemento do modelo contém cada fragmento gerado pela rasterização. Para garantir eficiência, utilizamos como estrutura de aceleração três variações de grades uniformes armazenadas como de texturas.

O método mais eficiente de renderização de aramado (*wireframe*) encontrado na literatura (6, 7) se baseia no mapeamento de textura para o desenho das bordas das primitivas. Cada vértice recebe coordenadas de textura que representam a sua distância para cada borda da primitiva e, devido a in-

terpolação dessas coordenadas, o aramado é mapeado corretamente sobre a superfíce renderizada. No entanto, essa abordagem não se mostrou adequada para o mapeamento do aramado na renderização direta de superfícies de corte por estas serem completamente independentes do modelo e pelo fato das linhas desejadas serem a interseção da superfície com as faces dos elementos do modelo. Por isso, adotamos aqui uma abordagem similar a apresentada em (2, 10), que também desenha o aramado durante a rasterização da geometria mas o desenha através de uma combinação linear que é função da distância do fragmento ao aramado e é efetuada no *fragment shader*. Nesses artigos, essa distância é calculada por vértice no *geometry shader* e então interpolada para cada fragmento. No caso do método proposto neste trabalho, como o aramado é desacoplado da superfície renderizada, o cálculo da distância é efetuado inteiramente no *fragment shader*. O cálculo desta distância envolve calcular a interseção da superfície de corte e as faces dos elementos do modelo.

Grande parte dos trabalhos que tratam de estruturas de aceleração armazenadas em GPU visam a aceleração de algoritmos de traçado de raio. As estruturas de aceleração mais comuns são *kd-tree* (9), hierarquia de volumes envolventes (*BVH*) (14) e grade uniforme (13). Os trabalhos mais recentes dessa área buscam um balanceamento entre a aceleração da travessia do raio pela cena renderizada e uma rápida reconstrução da estrutura. Neste trabalho, a velocidade de construção da estrutura não é relevante, pois a estrutura deve ser construída apenas uma vez, o que é feito em fase de pré-processamento. Além disso, o teste de interseção de fragmentos (pontos) é mais simples do que o teste de interseção de raios. O ponto de interseção de um raio pode ser qualquer ponto ao longo da direção do raio, enquanto o ponto de interseção de um fragmento é a própria posição do fragmento. Isso faz com que o acesso direto possibilitado por uma grade uniforme pareça mais adequado a essa situação do que estruturas hierárquicas como *kd-tree* e *BVH*, que introduziriam mais camadas de indireção. Por essa razão, optamos por usar uma grade uniforme. Nossa implementação de grade uniforme foi baseada na estrutura apresentada em (13, 11, 12), onde cada célula da grade possui uma lista das primitivas gráficas que se sobrepõem a célula e toda estrutura é armazenada de maneira compacta na forma de dois vetores de dados. Este trabalho implementa ainda duas variações desta estrutura, uma grade hierárquica de dois níveis e uma grade de centróides, ambas armazenadas da mesma maneira compacta. Equanto a grade hierárquica pode ser encontrada em (5), não encontramos estrutura similar à grade de centróide na literatura.

Em (19, 20, 8) são apresentadas algumas técnicas de renderização de operações em volumes, convexos e côncavos. Para ilustrar uma aplicação do

método proposto, implementamos uma técnica de recorte de volumes convexos baseada em profundidade apresentada em (8). A técnica consiste em armazenar em duas texturas separadas as profundidades da *back face* e a *front face* da geometria do volume de recorte e durante a renderização do modelo visualizado, utilizar o *fragment shader* para comparar a profundidade dos fragmentos com os valores armazenados nas texturas e decidir se eles devem ser descartados.

As principais contribuições deste trabalho são a apresentação de um algoritmo de visualização de seções de corte arbitrárias de malhas não estruturadas que tira proveito da rasterização do hardware gráfico e de programação em *shaders* para garantir eficiência, um método de mapeamento do aramado de modelos de malhas não estruturadas sobre superfícies arbitrárias e uma análise de desempenho de três estruturas de aceleração armazenadas em GPU que são variações de grades uniformes.

3
Método proposto

O método proposto utiliza a rasterização da placa gráfica para gerar os fragmentos da superfície de corte e realizar o cálculo da interseção no estágio programável de fragmentos do pipeline gráfico. Dessa maneira, o cálculo de interseção consiste em verificar individualmente para cada fragmento qual elemento do modelo ele intercepta. Para isso, é necessário armazenar os elementos da malha em GPU de um jeito que possibilite a realização de um teste eficiente de interseção entre fragmento e o volume de um elemento, e que faça com que o número de testes necessários para cada fragmento seja pequeno, ou seja, utilize algum tipo de estrutura de aceleração. O teste de interseção será discutido na Seção 3.1 e as estruturas de aceleração testadas serão explicadas na Seção 3.7.

Em termos de implementação, além de uma etapa de pré-processamento, o método faz uso de três estágios programáveis do pipeline gráfico:

- Pré-processamento: envolve a construção da estrutura de aceleração que será utilizada na busca de elementos.
- Vertex shader: primeiro estágio programável do pipeline gráfico, é utilizado para calcular a posição dos vértices da superfície de corte no espaço do modelo.
- Geometry shader: estágio de geometria, calcula informação necessária para a renderização do *wireframe* na superfície de corte.
- Fragment shader: estágio de fragmento, calcula a interseção entre os fragmentos e os elementos do modelo.

O cálculo da interseção no estágio de fragmentos transcorre em linhas gerais da seguinte maneira:

1. Busca os elementos candidatos na estrutura de aceleração.
2. Para todos os elementos candidatos, faz o teste de interseção entre o fragmento e o elemento.
3. Caso o fragmento intercepte algum elemento, calcula a cor do fragmento.

O cálculo da cor leva em conta o valor do campo escalar no fragmento, a renderização da malha e a iluminação. O cálculo do campo escalar no fragmento será explicado na Seção 3.4 e a renderização da malha será abordada na Seção 3.5. O cálculo de iluminação segue o modelo de Phong (1).

3.1 Teste de interseção

Em uma malha não estruturada, em geral, cada elemento é representado por uma lista ordenada de nós. No caso das malhas contempladas por este trabalho, cada elemento pode ter 4 ou 8 nós, elementos tetraédricos e hexaédricos, respectivamente. Tetraedros possuem 4 faces, cada uma contendo 3 nós, e hexaedros possuem 6 faces, cada uma contendo 4 nós. Os nós, e por consequência as faces, são compartilhados por elementos adjacentes. O teste individual entre um fragmento e um elemento da malha deve responder se o ponto que representa o fragmento está contido dentro do volume do elemento. Isso deve ocorrer de maneira eficiente, já que cada fragmento pode ter que realizar o teste com diversos elementos.

A solução proposta é mudar a representação usual do elemento de uma lista de nós para uma lista de planos que representam os planos das faces do elemento. Elementos tetraédricos e hexaédricos passam a ser representados por uma lista de 4 e 6 planos, respectivamente. Assim o teste de interseção entre o fragmento e um elemento passa a ser verificar em que lado de cada plano o fragmento está. Caso o fragmento esteja no lado interno ao volume do elemento para todos os planos, então ele está presente no interior do elemento. Isso pode ser verificado analisando-se o sinal do produto:

$$\begin{aligned} d &= \vec{f}^T \vec{p} \\ \vec{f} &= [frag_x, frag_y, frag_z, 1]^T \\ \vec{p} &= [p_a, p_b, p_c, p_d]^T \end{aligned} \tag{3-1}$$

onde $frag_x, frag_y, frag_z$ são as coordenadas da posição do fragmento e p_a, p_b, p_c, p_d são os coeficientes da equação do plano. O módulo de d representa o valor da distância do fragmento ao plano desde que a equação do plano esteja normalizada. Caso as normais dos planos apontem para fora do elemento o fragmento estará do lado de dentro do plano se o sinal de d for negativo. Como o produto escalar é uma operação desempenhada de maneira muito eficiente pelo *hardware* gráfico, o teste de interseção tem baixo custo computacional.

No caso de tetraedros lineares, a lista de planos representa de forma perfeita o elemento. Já no caso dos hexaedros os planos são uma aproximação das faces do elemento, pois cada uma possui 4 nós que não necessariamente são coplanares. Esse tipo de aproximação também é usado em outros algoritmos de visualização, como por exemplo algoritmos de visualização volumétrica (15), que utilizam dois planos para representar cada face do hexaedro. No nosso

caso, a face do elemento hexaédrico foi aproximada pelo plano médio entre todos os planos derivados das combinações de 3 a 3 dos 4 nós da face. O plano médio é ancorado no centróide dos nós da face e sua normal é calculada como a média das normais dos planos derivados.

Outra questão específica de malhas hexaédricas é o aparecimento de elementos degenerados. Isso acontece quando nós de um mesmo elemento possuem coordenadas iguais ou muito próximas. Como resultado disso, aparecem elementos com formas de "cunha" e "pirâmide" mas que ainda assim formam volumes que não podem ser descartados pelo algoritmo de visualização. Esses elementos degenerados podem possuir faces que formam uma reta ou até mesmo um único ponto. Durante a construção dos planos na fase de pré-processamento, a gereção de planos inválidos inviabilizaria o teste de interseção. A solução proposta para o problema foi substituir o plano inválido por um plano paralelo ao plano formado pela face oposta à face disforme. Para preservar o volume do elemento, esse plano paralelo deve ser ancorado no nó da face disforme que seja mais distante da face oposta, como pode ser visto na Figura 3.3(b).

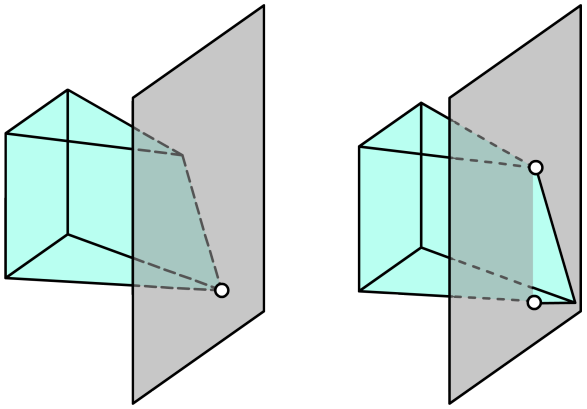
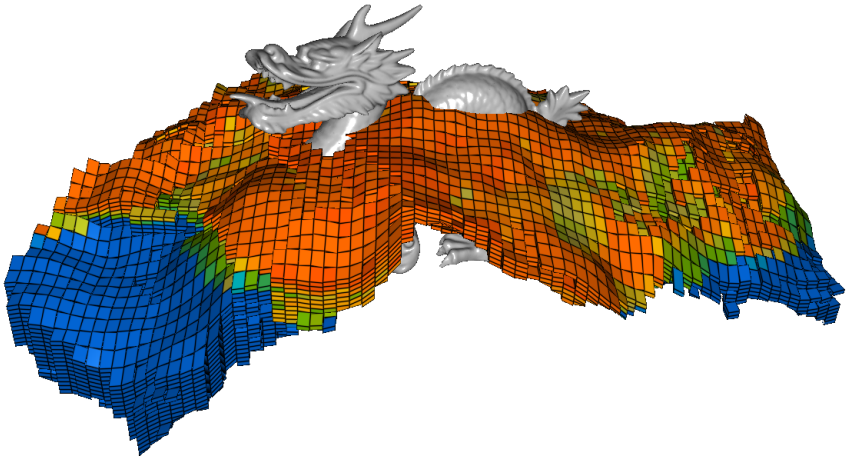


Figura 3.1: A figura à esquerda exemplifica o posicionamento correto do plano no nó mais distante e a figura à direita exemplifica o posicionamento incorreto, no nó mais próximo, o que exclui parte do volume do elemento.

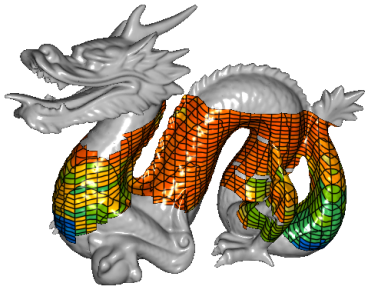
3.2 Posicionamento do objeto de corte

Antes de se calcular a interseção entre a superfície de corte e o modelo, é necessário que posicionemos a superfície no espaço de coordenadas do volume do modelo. Esse processo é similar ao que ocorre no pipeline gráfico convencional, em que utilizamos uma transformação de instanciação para posicionar um objeto no espaço de coordenadas do mundo. Analogamente, no caso das superfícies de corte, é realizada uma transformação que posiciona a superfície no espaço de coordenadas do modelo. A Figura 3.2 ilustra essa

transformação. Isso é efetuado especificando-se uma matriz que representa essa transformação de instanciação e realizando o cálculo do posicionamento dos vértices da superfície de corte no espaço de coordenadas do modelo na GPU, no estágio programável de vértices (*vertex shader*) do pipeline gráfico. Durante a rasterização, esse valor é interpolado para cada fragmento.



(3.2(a)) Posicionamento da superfície em relação ao modelo.



(3.2(b)) Resultado do cálculo de interseção.

Figura 3.2: Exemplo do posicionamento de uma superfície em relação ao modelo.

3.3 Superfície de corte e superfície renderizada

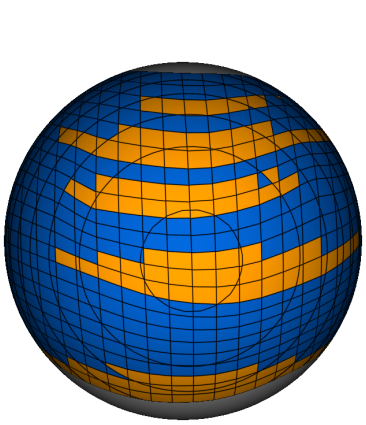
O método proposto desacopla a superfície de corte (usada no cálculo de interseção com o modelo) da superfície renderizada. Exemplos da aplicação deste recurso podem ser vistos na Seção 5.2 e na Figura 3.3.

Conceitualmente é como se dividíssemos o algoritmo em dois processos que ocorrem de maneira paralela. De um lado são calculados as coordenadas de tela e a iluminação do fragmento e do outro a cor pré-iluminação. No primeiro processo estão envolvidos os cálculos relacionados ao pipeline gráfico convencional como as transformações de instanciação, de câmera, de projeção e de viewport além do cálculo de iluminação. No segundo estariam envolvidos a transformação de instanciação para o espaço do modelo, o cálculo de interseção e o cálculo do valor do campo escalar no fragmento. A entrada do primeiro processo seria a superfície renderizada, que é a superfície rasterizada e que efetivamente aparece na imagem final. Já a entrada do segundo processo seria a superfície de corte, que é a superfície utilizada no cálculo de interseção com o modelo a ser visualizado. É natural que na maior parte dos casos deseja-se que a mesma superfície seja usada tanto para a renderização como para o cálculo do corte, porém em algumas aplicações pode ser interessante mapear a interseção entre uma determinada superfície de corte e o modelo sobre uma outra superfície, como é ilustrado pela Figura 3.3.

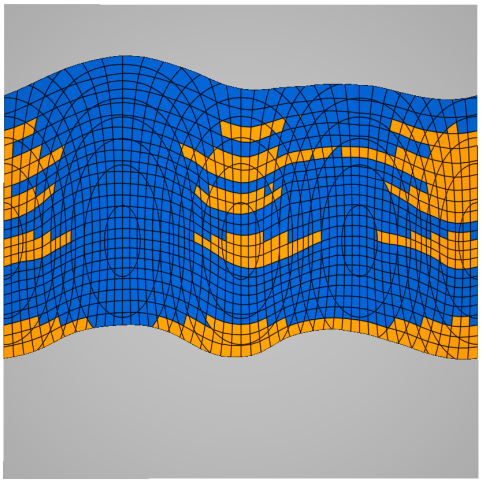
Para se conseguir esse mapeamento é necessário que haja uma relação entre as duas superfícies. Essa relação é expressa utilizando-se coordenadas de textura tridimensionais. Sendo assim, cada vértice enviado pelo pipeline deve possuir coordenadas que descrevam sua posição em relação à superfície renderizada e coordenadas que descrevam sua posição em relação à superfície de corte. No entanto essa relação entre o conjunto de vértices da superfície renderizada e a superfície de corte não necessariamente precisa ser uma relação bijetora. Como ilustrado na Figura 3.4, a mesma coordenada de corte pode ser atribuída a mais de um vértice da superfície renderizada. Isso abre caminho para se utilizar linhas de corte em vez de superfícies, e visualiza-las sobre a superfície renderizada. A Seção 5.2 apresenta exemplos do uso de linhas de corte.

3.4 Cálculo do campo escalar

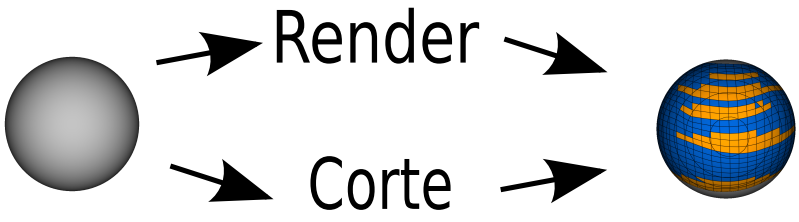
Após realizar o cálculo de interseção e identificar em qual elemento do modelo o fragmento está contido, é necessário avaliar o valor do campo escalar na posição do fragmento para então fazer o mapeamento de cores correto.



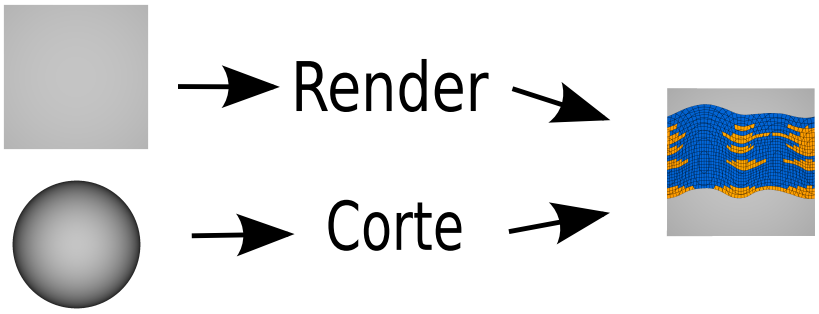
(3.3(a))



(3.3(b))



(3.3(c))



(3.3(d))

Figura 3.3: A Figura 3.3(a) mostra um corte que utiliza como superfície renderizada e superfície de corte uma esfera. A Figura 3.3(b) mostra o corte presente na Figura 3.3(a) mapeado sobre um plano. Neste caso a superfície renderizada é o plano e a superfície de corte é a esfera. As Figuras 3.3(c) e 3.3(d) ilustram respectivamente como foram geradas as imagens 3.3(a) e 3.3(b).

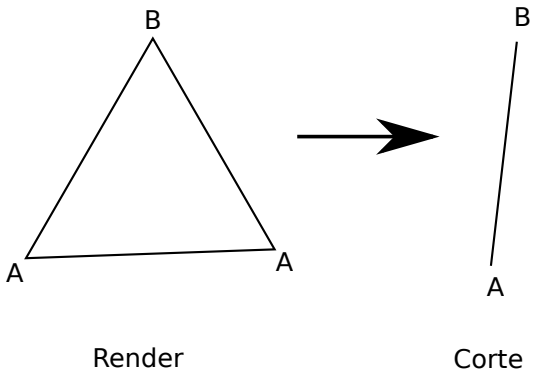


Figura 3.4: Ilustração de como é feito o mapeamento de linhas de corte na superfície renderizada. A e B são as coordenadas de corte (textura) dos vértices do triângulo, que formam o seguimento de reta que será usado no cálculo do corte.

O campo escalar em um modelo pode possuir um valor por nó ou um valor por elemento. Caso o campo possua um valor por elemento, todos os pontos interiores a um elemento possuem o mesmo valor escalar e, por isso, não há necessidade de cálculos adicionais durante a visualização do corte. Já no caso do campo escalar apresentar um valor por nó, o campo em um ponto no interior de um elemento deve ser calculado através da interpolação dos valores do campo nos nós do elemento. A seguir é explicado como é feito a interpolação para modelos de tetraedros e para modelos de hexaedros.

3.4.1 Tetraedros

Nos modelos de tetraedros lineares cada elemento é composto por 4 nós que possuem coordenadas espaciais e valores que representam o campo escalar em suas posições. O campo escalar em um ponto no interior de um elemento é calculado através da interpolação linear dos valores do campo nos 4 nós do elemento. Essa interpolação é realizada utilizando-se coordenadas baricêntricas. A coordenada baricêntrica de um ponto no interior de um elemento em relação ao nó i do elemento é dado por:

$$b_i = \frac{A_i h_i}{3V} \tag{3-2}$$

onde A_i é a área da face oposta ao nó i, h_i é a distância do ponto em relação ao plano desta face e V é o volume do elemento.

Sendo assim, o valor do campo escalar na posição do fragmento em relação ao modelo é calculado como:

$$s_f = \sum_{i=1}^{4} b_i s_i = \sum_{i=1}^{4} \frac{A_i h_i s_i}{3V} \tag{3-3}$$

onde s_i é o valor do campo escalar no nó i do elemento que o fragmento interceptou.

Observando atentamente a Equação 3-3 é possível perceber que o único termo que depende da posição do fragmento é h_i, a distância do fragmento em relação a face oposta ao nó. Definindo o termo c_i como:

$$c_i = \frac{A_i s_i}{3V} \tag{3-4}$$

é possível reescrever a equação 3-3 como:

$$s_f = \sum_{i=1}^{4} c_i h_i \tag{3-5}$$

Como c_i só depende de valores relacionados aos nós do elemento, pode ser calculado durante a fase de pré-processamento e armazenado em uma textura que será acessada no shader. Já que é necessário apenas um valor por nó, devem ser armazenados apenas 4 *floats* por elemento. Essa textura deve ser atualizada sempre que o campo escalar for alterado. Outra opção seria uma textura contendo os valores $\frac{A_i}{3V}$ e outra textura contendo o campo escalar. Isso facilitaria a atualização do campo escalar, mas o uso de memória de vídeo é crítico para modelos grandes.

3.4.2 Hexaedros

O cálculo de interpolação para hexaedros costuma ser feito no espaço de coordenadas naturais de cada elemento. Esse espaço é tridimensional e suas coordenadas (ξ,η,μ) variam dentro do intervalo de $[-1, 1]$. Os 8 nós do hexaedro são posicionados nos 8 cantos do cubo formado pelo domínio do espaço, como ilustrado na Figura 3.5. Uma vez calculadas as coordenadas naturais, a interpolação do campo escalar é feita utilizando-se as funções de forma do elemento:

$$s_f = \sum_{i=1}^{8} (1 + \xi\xi_i)(1 + \eta\eta_i)(1 + \mu\mu_i) s_i \tag{3-6}$$

onde ξ_i, η_i, μ_i são as coordenadas naturais do nó i e s_i é o valor do campo escalar no nó i.

Normalmente o cálculo das coordenadas naturais é feito através do método numérico de Newton-Raphson. Como esse é um método numérico iterativo tende a ser custoso demais para executá-lo no shader para cada fragmento. Por isso, neste trabalho, as coordenadas naturais são aproximadas de forma menos precisa. A aproximação consiste em calcular para cada par de planos opostos a direção da normal média entre as normais dos planos e

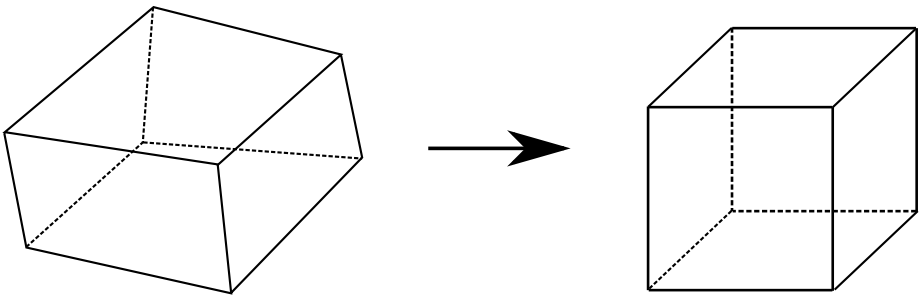


Figura 3.5: Transformação para as coordenadas naturais.

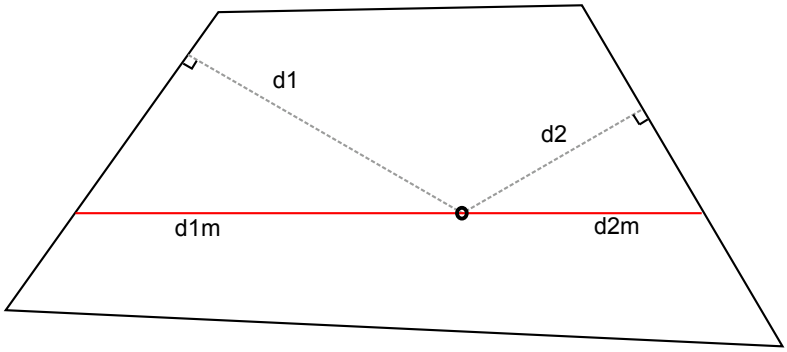


Figura 3.6: Ilustração em 2D da aproximação das coordenadas naturais. As linhas pontilhadas $d1$ e $d2$ representam as distâncias do fragmento para cada uma das faces. A linha em vermelho representa a direção da normal média entre as duas faces opostas e os seguimentos $d1m$ e $d2m$ representam as distâncias nesta direção que são usadas na aproximação.

calcular para cada plano sua distância em relação ao fragmento na direção da normal média. A Figura 3.6 ilustra esse processo. Assim, para cada par de planos opostos, é calculada uma normal média entre as normais dos planos (n_1,n_2):

$$\vec{n}_m = \frac{\vec{n}_1 - \vec{n}_2}{|\vec{n}_1 - \vec{n}_2|} \tag{3-7}$$

As distâncias d_1 e d_2, entre a posição do fragmento e cada um dos planos opostos, são transformadas para a direção da normal média:

$$d_{1m} = \frac{d_1}{\vec{n}_1^T \vec{n}_m} \qquad d_{2m} = \frac{d_2}{\vec{n}_2^T \vec{n}_m} \tag{3-8}$$

Então as coordenadas naturais são calculadas normalizando-se a distância no intervalo $[-1, 1]$:

$$\xi = \frac{2d_{1m}}{d_{1m} + d_{2m}} - 1 \tag{3-9}$$

Após computar as coordenadas naturais, o campo escalar no fragmento é calculado utilizando as funções de forma como visto na Equação 3-6. Esse método necessita apenas que o próprio campo escalar seja armazenado, o que consiste em 8 *floats* por elemento. As Figuras 3.7 mostram resultados da interpolação em modelos de hexaedros.

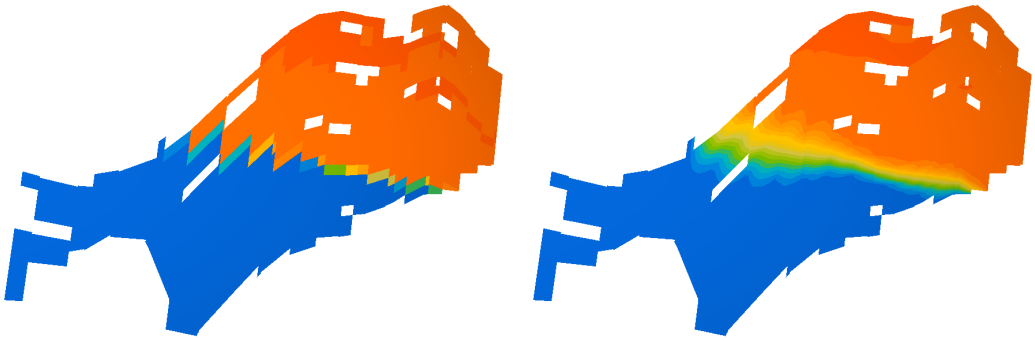


Figura 3.7: Exemplo do resultado da interpolação de valores em um corte planar vertical em um modelo de reservatório de petróleo.

3.5 Renderização do aramado

Outro desafio da aplicação da técnica de superfícies de corte é a renderização de informações associadas à topologia do modelo, como por exemplo a visualização do aramado (*wireframe*) da malha, que pode ser útil para revelar informações estruturais sobre a malha utilizada para representar o domínio do modelo. Renderizar o aramado da malha em um objeto de corte significa delimitar explicitamente os limites dos elementos na superfície renderizada. Para isso é necessário detectar onde o objeto de corte intercepta as faces dos elementos do modelo.

A estratégia usada neste trabalho para renderizar o aramado da malha é baseada no método apresentado em (2). A técnica consiste em computar a cor de cada fragmento através da combinação linear:

$$c = c_w I(d) + (1 - I(d))c_s \tag{3-10}$$

onde c_w é a cor desejada para o aramado e c_s é a cor do mapa de cores associado ao valor do campo escalar no fragmento. I determina a intensidade do aramado no fragmento e é função de d que é a distância, em espaço de tela, do fragmento em relação ao ponto mais próximo da interseção entre a superfície de corte e os planos das faces do elemento interceptado pelo fragmento.

A função utilizada para calcular a intensidade do aramado no fragmento é:

$$I(d) = exp_2(-2d^2) \tag{3-11}$$

Essa função de intensidade produz linhas finas porém com pouco *aliasing*.

Antes de explicar como é feito o cálculo da distância d, é necessário detalhar mais o desacoplamento entre superfície de corte e superfície renderizada. Como foi explicado na Seção 3.3, o método proposto pode ser conceitualmente dividido em dois processos que ocorrem de maneira paralela. De um lado é realizado o cálculo da iluminação e o cálculo das coordenadas de tela do fragmento, e do outro é calculada a posição do fragmento em relação ao modelo e a interseção do fragmento com o modelo. Para cada vértice enviado ao pipeline, o cálculo das coordenadas de tela e de coordenadas do modelo é respectivamente:

$$\vec{v}_s = V_p M_{vp} \vec{v}_r \tag{3-12}$$

$$\vec{v}_m = M_i \vec{v}_c \tag{3-13}$$

onde $\vec{v}_s$ é o vetor de coordenadas de tela do vértice, $\vec{v}_m$ é o vetor de coordenadas do modelo do vértice, $\vec{v}_r$ representa as coordenadas do vértice em relação a superfície renderizada, $\vec{v}_c$ representa as coordenadas do vértice em relação a superfície de corte, que pode ser desacoplada da superfície renderizada, V_p e M_{vp} são respectivamente as matrizes *Viewport* e *ModelViewProjection*, M_i é a matriz de intanciação da superfície de corte. Durante o estágio de rasterização as coordenadas de tela do fragmento f_s e a posição do fragmento em relação ao modelo f_m são obtidas através da interpolação de v_s e v_m respectivamente.

As coordenadas de corte v_c são enviadas ao pipeline como coordenadas de textura dos vértices v_r da superfície renderizada. Isso não só abre a possibilidade de desacoplarmos a superfície de corte da superfície renderizada como abre também a possibilidade de usarmos linhas de corte em vez de superfícies de corte. Assim podemos mapear a interseção de uma linha com o modelo sobre a superfície renderizada. Em termos de primitivas isso é feito atribuindo-se a dois vértices de um triângulo da superfície renderizada a mesma coordenada de textura. Temos assim dois tipos de primitivas de corte, triângulos e segmentos de reta, que são mapeados na primitiva gráfica triângulo. O que desejamos visualizar renderizando o aramado é a interseção dessas primitivas de corte com as faces dos elementos. A interseção da primitiva triângulo com uma das faces de um elemento é um seguimento de reta. A interseção da primitiva seguimento de reta com uma das faces de um elemento é um ponto. As Figuras 3.8 ilustram a interseção entre as primitivas de corte e as faces dos elementos.

Para explicar explicar o cálculo de d devemos definir o ponto p_m. Seja $\mathbf{S}$ o conjunto dos pontos que formam a interseção de uma primitiva com as faces de um elemento. Se a primitiva for um triâgulo, $\mathbf{S}$ será formado pela união dos seguimentos de reta que representam a interseção do triângulo com cada

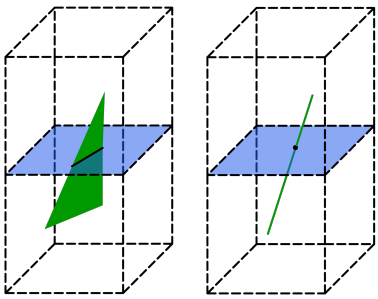


Figura 3.8: Ilustração da interseção das primitivas de corte com a face de um elemento.

uma das faces do elemento. Se a primitiva for um seguimento de reta, **S** será formado pela união dos pontos de interseção entre o seguimento de reta e as faces do elemento. O ponto p_m é definido como:

$$p_m \in \mathbf{S}; \qquad \forall p \in \mathbf{S} : |p_m - f_m| \leq |p - f_m| \tag{3-14}$$

onde f_m representa as coordenadas do fragmento no espaço do modelo. Ou seja, p_m é o ponto de **S** mais próximo do fragmento no espaço do modelo. Sendo assim, a estratégia utilzada neste trabalho para calcular a distância d em espaço de tela é transformar o ponto p_m das coordenadas do modelo para as coordenadas de tela e então calcular sua distância em relação ao fragmento. Para isso, como a superfície de corte e a superfície renderizada são desacopladas, é necessário antes mapear p_m do espaço do modelo para o espaço de coordenadas da superfície superfície renderizada.

Assim, o cálculo da distância d segue as seguintes etapas:

1. Cálculo da interseção entre a primitiva associada ao fragmento e os planos das faces dos elementos.

2. Cálculo do ponto p_m.

3. Mapeamento do ponto p_m no espaço de coordenadas do modelo para o ponto p_r no espaço de coordenadas da superfície renderizada.

4. Cálculo do ponto p_s em coordenadas de tela:

$$\vec{p_s} = V_p M_{vp} \vec{p_r} \tag{3-15}$$

5. Cálculo da distância d em espaço de tela:

$$d = |\vec{p_s} - \vec{f_s}| \tag{3-16}$$

onde f_s é o vetor de coordenadas de tela do fragmento.

As três primeiras etapas do cálculo de d, que são específicas para cada primitiva de corte, são explicadas nas duas Seções seguintes.

3.5.1 Triângulo

Para calcular o ponto mais próximo da interseção da primitiva de corte com as faces do elemento no espaço do modelo, p_m, devemos primeiro calcular a própria interseção entre a primitiva e as faces. Para facilitar o cálculo da interseção entre a primitiva e uma das faces, o triângulo será substituído pelo plano formado pela normal do triângulo e a posição do fragmento no espaço do modelo. Essa normal é calculada no *geometry shader* utilizando as coordenadas de corte no espaço do modelo dos três vértices do triângulo. Dessa maneira, o cálculo da interseção entre a primitiva e uma das faces passa a ser calcular a interseção entre dois planos. A equação paramétrica da reta de interseção entre o plano do triângulo e o plano de uma das faces é calculada como:

$$\begin{aligned} \vec{p} &= (c_1\vec{n}_1 + c_2\vec{n}_2) + \lambda(\vec{n}_1 \times \vec{n}_2) \\ c_1 &= \frac{d_1 - d_2(\vec{n}_1^T\vec{n}_2)}{1-(\vec{n}_1^T\vec{n}_2)^2} \\ c_2 &= \frac{d_2 - d_1(\vec{n}_1^T\vec{n}_2)}{1-(\vec{n}_1^T\vec{n}_2)^2} \end{aligned} \tag{3-17}$$

onde p é um ponto qualquer da reta, n_1 e n_2 são as normais dos planos e d_1 e d_2 são as últimas coordenadas da equação dos planos. Após calcular a equação da reta de interseção para cada uma das faces do elemento, calculamos o ponto mais próximo do fragmento em cada uma das retas. O ponto da reta mais próximo ao fragmento é calculado como:

$$\begin{aligned} \vec{p_l} &= \vec{l_p} + \lambda\vec{l_d} \\ \lambda &= \frac{(\vec{l_d}^T\vec{f_m}) - (\vec{l_p}^T\vec{l_d})}{\vec{l_d}^T\vec{l_d}} \end{aligned} \tag{3-18}$$

onde l_p e l_d são, respectivamente, o ponto e o vetor direção que compõem a reta e f_m é a posição do fragmento no espaço do modelo. Dentre os pontos mais próximos ao fragmento de cada reta, tomamos como p_m o que apresenta menor distância ao fragmento. Em nossos testes, se mostrou suficiente usar apenas as duas faces mais próximas à posição do fragmento em vez de usar todas as faces do elemento.

O cálculo de p_r no espaço de coordenadas da superfície renderizada será feito através da interpolação linear das coordenadas dos vértices do triângulo nesse espaço:

$$\vec{p_r} = \sum_{i=1}^{3} b_i \vec{v}_{ri} \tag{3-19}$$

onde b_i é a coordenada baricêntrica de p_m em relação ao vértice i do triângulo e v_{ri} é a coordenada do vértice i no espaço da superfície renderizada.

3.5.2 Segmento de reta

A abordagem para calcular o ponto p_m para a primitiva de corte segmento de reta é a mesma que a adotada para o triângulo. Calculamos a interseção da primitiva com cada uma das faces do elemento interceptado pelo fragmento e extraímos dessas interseções o ponto mais próximo à posição do fragmento. Em vez de utilizar o próprio segmento de reta utilizamos a reta que o define. A reta é formada pela direção do segmento de reta e pela posição do fragmento no espaço do modelo. A direção da reta é calculada no *geometry shader*. O cálculo do ponto de interseção entre esta reta e o plano de uma das faces do elemento e feito por:

$$\begin{aligned} \vec{p} &= \vec{l_p} + \lambda \vec{l_d} \\ \lambda &= \frac{(\vec{p_0} - \vec{l_p})^T \vec{n}_p}{\vec{l_d}^T \vec{n}_p} \end{aligned} \tag{3-20}$$

onde l_p e l_d são respectivamente o ponto e o vetor direção que compõem a reta, p_0 é um ponto qualquer do plano e n_p é a normal do plano. Após calcular os pontos de interseção para cada face do elemento, tomamos como p_m o que apresenta menor distância à posição do fragmento. Assim como no caso dos triângulos, em nossos testes foi suficiente usar apenas as duas faces mais próximas da posição fragmento.

Para computar p_r no espaço de coordenadas da superfície renderizada, primeiro calculamos no *geometry shader* a direção da reta neste espaco. Consideramos a direção da reta no espaço da superfície renderizada como sendo perpendicular à aresta do tríangulo cujos vértices possuem a mesma coordenada de corte. Sendo assim o cálculo de p_r no espaço da superfície renderizada é:

$$\vec{p_r} = \vec{f_r} + \lambda \vec{l_r} \tag{3-21}$$

onde f_r é posição do fragmento no espaço da superfície renderizada, l_r é a direção da reta neste espaço. Como as retas dos dois espaços são equivalentes, o valor do λ desta equação é igual ao calculado na equação 3-20.

3.6 Modelo em GPU

Para realizarmos os procedimentos descritos nas seções anteriores é necessário que o modelo seja armazenado na memória da GPU. O modelo será estruturado na forma de dois vetores de dados armazenados como texturas. O primeiro contendo os planos dos elementos e o segundo contendo a informação necessária para o cálculo do campo escalar. Ainda que elementos adjacentes compartilhem nós e faces, não levaremos isso em conta pois para tirar proveito desse compartilhamento seria necessário introduzir uma camada a mais de indireção e elevaria bastante o número de acessos a textura do método. Sendo assim, será armazenado 1 plano por face e um escalar por nó para cada elemento. Caso o campo escalar seja amostrado por elemento, é necessário armazenar apenas 1 escalar por elemento em vez de 1 por nó. Cada plano é representado por 4 *floats* e cada escalar é representado por 1 *float*. Isso totaliza 80 *bytes* por elemento para modelos de tetraedros com campo escalar por nó, 68 *bytes* para modelos de tetraedros com campo escalar por elemento, 128 *bytes* para modelos de hexaedros com campo escalar por nó e 100 *bytes* para modelos de hexaedros com campo escalar por elemento.

3.7 Estruturas de aceleração

Apenas armazenar o modelo em GPU não é suficiente para que o algoritmo execute em tempo real. É necessário a implementação de uma estrutura de aceleração que faça com que somente os elementos próximos da posição do fragmento sejam usados no teste de interseção. Como estrutura de aceleração foram implementadas três variações de grades regulares que serão descritas a seguir.

3.7.1 Grade regular uniforme

A grade regular uniforme é uma estrutura de subdivisão espacial. A idéia básica consiste em subdividir a caixa envolvente alinhada ao eixo ($AABB$) do modelo em células de tamanho igual ao longo dos três eixos e construir listas de elementos para cada célula da grade, armazenando cada elemento nas listas de todas as células que ele se sobrepõe. O objetivo é possibilitar o acesso direto a qualquer lista de elementos dado a célula. Assim, após computarmos em qual célula da grade o fragmento está, o teste de interseção é efetuado apenas para os elementos presentes na lista desta célula.

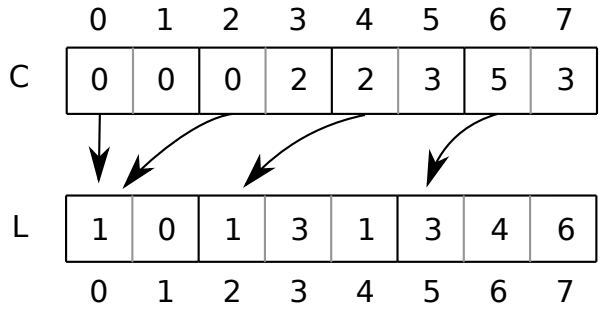


Figura 3.9: Ilustração do armazenamento grade regular. O vetor C representa o vetor de células e L representa o vetor de listas.

Essas listas de elementos na verdade guardam os índices dos elementos. Em posse do índice de um elemento, é possível acessar os seus dados no vetor de planos e no vetor de valores do campo escalar. As listas são armazenadas de forma contígua em um vetor que é acessado em GPU na forma de textura. Para possibilitar o acesso direto é necessário uma maneira de se mapear o identificador da célula da grade para o início da lista de elementos da célula no vetor de listas. Isso é feito construindo-se um segundo vetor, que chameremos de vetor de células, em que para cada célula da grade são armazenados o índice do início da sua lista de elementos no vetor de listas e o número de elementos presentes na lista. Dessa maneira, com o identificador de uma célula da grade, é possivel acessar sua lista de elementos e percorre-la corretamente. A Figura 3.9 ilustra esta estrutura de dados.

Um fator decisivo tanto para o desempenho quanto para o gasto de memória da grade regular é a escolha de uma resolução adequada. Alguns tipos de modelos, como por exemplo modelos de reservatório de petróleo, tem seus elementos organizados topologicamente em uma grade tridimensional. Para esses modelos, tiramos proveito das dimensões da grade topológica do modelo para o cálculo da resolução. As dimensões da grade n_x, n_y, n_z são calculadas como múltiplos das dimensões topológicas n_i, n_j, n_k:

$$n_x = kn_i \qquad n_y = kn_j \qquad n_z = kn_k \tag{3-22}$$

onde k é o fator que determina o quão denso é a grade. Para modelos que não apresentam esse tipo de organização topológica calculamos a resolução de maneira similar à heurística utilizada em implementações de grades regulares para algoritmos de traçado de raio (18):

$$n_x = kd_x\sqrt[3]{\frac{E}{V}} \qquad n_y = kd_y\sqrt[3]{\frac{E}{V}} \qquad n_z = kd_z\sqrt[3]{\frac{E}{V}} \tag{3-23}$$

onde E é o número total de elementos do modelo, V é o volume da grade e d_x, d_y, d_z formam a diagonal da grade. O efeito da variação dos fatores k será

abordado no capítulo 4.

Outro fator importante é o cálculo de interseção dos elementos com as células da grade, para construção das listas. O cálculo para elementos hexaédricos foi implementada de maneira bastante simples. Para cada elemento calculamos a sua $AABB$ e o incluímos nas células da grade que se sobrepõem à $AABB$. No caso dos tetraedros, se mostrou necessário algo um pouco mais sofisticado pois a $AABB$ é uma aproximação muito ruim do volume do tetraedro. Por isso adicionalmente à identificação das células sobrepostas à $AABB$ do elemento, testamos para cada célula sobreposta se esta intercepta o tetraedro. Esse teste é realizado testando-se individualmente a interseção de cada face triangular do tetraedro com a célula em questão (1) e verificando se a célula não se encontra completamente no interior do tetraedro. Dessa maneira apenas as células da grade que de fato interceptam o tetraedro terão o elemento incluído em sua lista.

3.7.2 Grade hierárquica de dois níveis

Alguns modelos apresentam uma variação muito grande no tamanho dos seus elementos. Nestes modelos existem determinadas áreas de interesse onde a densidade de elementos é bastante alta e o tamanho destes é reduzido, e áreas de menor interesse onde poucos elementos ocupam grandes porções do volume do modelo. Nestes casos, grades uniformes com uma resolução mais baixa tendem a ser ineficientes pois as listas de elementos das células da grade que contemplam essas regiões de interesse se tornam grandes demais, pois como o tamanho das células é relativamente grande elas acabam englobando muitos elementos. Isso deixa o algoritmo lento, principalmente quando o foco da visualização está nessas regiões. Por outro lado o problema com resoluções mais altas é o desperdício de memória, já que em regiões do modelo menos densas um número grande de células da grade contempla poucos elementos do modelo.

A proposta para solucionar esse problema foi o desenvolvimento de uma grade hierárquica de dois níveis, baseada em uma das estruturas apresentadas em (5). A idéia é basicamente construir uma grade como a descrita na seção anterior com uma resolução relativamente baixa e refinar localmente as células da grade cujas listas de elementos apresentem um número maior que um determinado limite. Esse refinamento consiste em construir uma grade uniforme como a da seção anterior que abrange apenas o volume da célula refinada. A resoluçao dessa nova grade é calculada como função do número de elementos que caíram na célula a ser refinada:

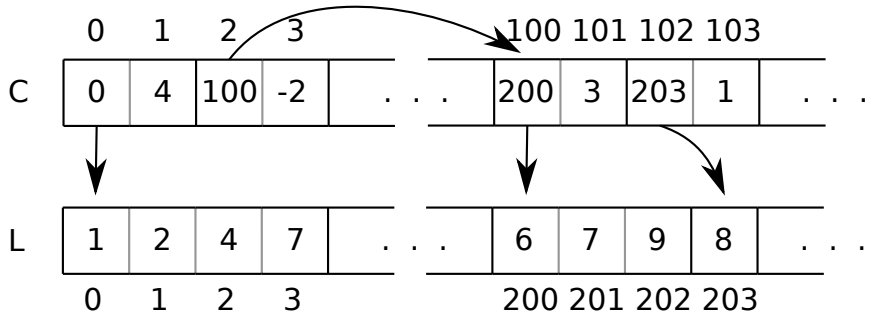


Figura 3.10: Ilustração do armazenamento da grade hierárquica. O vetor C representa o vetor de células e L representa o vetor de listas.

$$dim = floor(\sqrt[3]{n}) \tag{3-24}$$

onde dim é a dimensão usada para os três eixos da grade refinada e n é o número de elmentos presentes na célula da grade original.

Representaremos essa grade hierárquica em GPU de maneira similar à representação da grade uniforme apresentada anteriormente. Para grade uniforme utilizamos dois vetores: o vetor de listas guarda as listas de elementos de todas as células da grade, e o vetor de células que guarda para cada célula um índice para o início de sua lista no vetor de listas e o número de elementos presentes na lista. Para a grade hierárquica manteremos essa mesma estrutura de dois vetores. O vetor de listas será exatamente igual ao vetor utilizado na grade regular, acrescido das listas de células refinadas. Já o vetor de células será dividio em duas partes: a primeira armazenará a informação das células da grade principal e a segunda armazenará a informação relativa às células das grades resultantes do refinamento de células da grade original. Para as células da grade original que não foram refinadas continuam sendo armazenados uma indireção para o início da lista de elementos e o número de elementos que a lista possui. O mesmo acontece para as células das grades refinadas. Já para as células refinadas da grade original serão armazenados o índice do início da grade refinada no próprio vetor de células e a dimensão da grade refinada, que será a mesma para os três eixos. Para que seja possível diferenciarmos uma célula refinada de uma célula não refinada, a dimensão do refinamento da célula será armazenado com o sinal negativo. A Figura 3.10 ilustra essa estrutura hierárquica.

3.7.3 Grade regular de centróides

As duas estruturas de aceleração apresentasdas até aqui trabalham com o conceito de listas de elementos. Como as listas armazenam índices para elementos, na prática elas funcionam como uma camada de indireção para

se acessar os dados de um determinado elemento, pois durante o cálculo de interseção antes de se realizar qualquer teste entre fragmento e elemento é necessário acessar a textura do vetor de listas para se extrair o índice do elemento. Essa camada de indireção se mostra necessária devido a possibilidade de um mesmo elemento estar presente em mais de uma lista. Como o gargalo do algoritmo se mostrou ser acessos à textura, nos pareceu interessante investigar uma forma de se eliminar essa camada de indireção.

A forma investigada para se eliminar o vetor de listas, o que além de se eliminar uma camada de indireção deve economizar memória, foi associar cada elemento do modelo a somente uma célula da grade. A célula escolhida para armazenar o elemento é aquela que contém o centróide da $AABB$ do elemento. Dessa maneira é possível ordenar os elementos no vetor de planos e no vetor do campo escalar de acordo com a incidência destes nas células da grade. Assim, em vez de armazenar no vetor de células um índice para uma lista de índices de elementos, armazenamos diretamente o índice do primeiro elemento da célula.

A idéia básica do algoritmo continua a mesma, testar a interseção do fragmento com os elementos presentes na célula da grade em que o fragmento caiu, porém existe a possibilidade que o fragmento intercepte um elemento cujo centróide não esteja presente nessa célula. Por isso, caso o fragmento não intercepte nenhum elemento da célula que o contém será ncessário realizar o teste também com elementos de outras células. Para restringirmos a nossa procura somente às células vizinhas devemos fixar o tamanho da célula da grade em cada dimensão como a maior variação de um elemento do modelo naquela dimensão. Dessa forma cada elemento pode se sobrepor a no máximo duas células em cada direção. Cada célula da grade tem até 26 células vizinhas, porém não é necessário que realizemos o teste de interseção para os elementos de todas essas células. Como cada elemento se estende no máximo por duas células em cada direção, podemos procurar somente nas vizinhas que tocam o octante da célula que contém o fragmento. O que dá um total de 7 células.

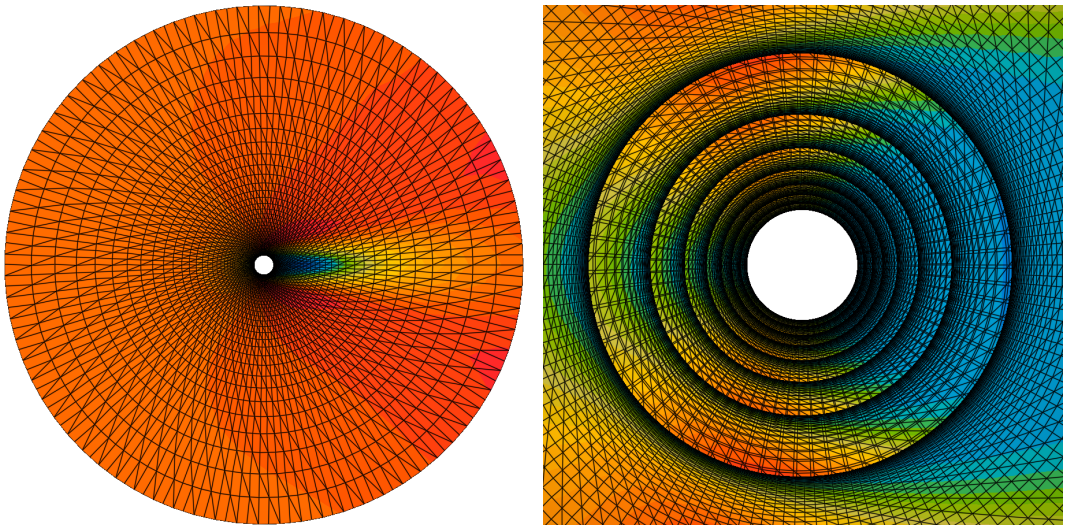
Os resultados de desempenho e gasto de memória dessa estrutura serão analisados no próximo capítulo, no entanto, é possível se prever que para que o algoritmo execute de maneira eficiente com essa estrutura de aceleração o número de elementos em cada célula da grade deve ser relativamente baixo e o modelo visualizado deve apresentar pouca variação de tamanho entre os seus elementos. Caso contrário, o custo de se ter que testar elementos de células vizinhas superará o ganho causado pela eliminação da indireção do vetor de listas.

4
Análise de desempenho

Este capítulo apresenta os resultados de desempenho e gasto de memória do método proposto nos testes realizados. A implementação do método proposto foi feita em *C++* utilizando *OpenGL* e a linguagem de *shader GLSL*. Os testes foram efetuados em um processador *Intel i7* de *3.3 GHz* com *24 GB* de memória *RAM*, utilizando como placa de vídeo uma *Nvidia GeForce GTX 680*. A Seção 4.1 apresenta os resultados dos testes para modelos de tetraedros e a Seção 4.2 para modelos de hexaedros.

4.1
Tetraedros

Os resultados expostos a seguir foram obtidos utilizando o modelo de tetraedros *Liquid Oxygen Post* que possui 109.744 nós distribuidos em 616.050 elementos. Este modelo apresenta uma grande variação no tamanho de seus elementos. A parte central, mais densa, possui elementos muito menores do que os elementos que se encontram nas partes afastadas do centro. Por isso foram utilizadas duas cenas nos testes para medir o desempenho do método. A *cena 1* mostra uma visão geral do modelo e é composta por 10 planos de corte renderizados de trás para frente para evitar que fragmentos sejam descartados antes de passarem pelo *fragment shader*. A *cena 2* possui os mesmos 10 planos de corte, mas foca na região mais densa do modelo. Foi utilizada nos testes uma resolução de 800x800 e foram gerados ao todo aproximadamente 4.1 milhões de fragmentos na *cena 1* e 4.9 milhões na *cena 2*. Devido ao número de fragmentos gerados, tanto a *cena 1* quanto *cena 2* demandam do método proposto um esforço computacional maior do que uma cena normal em uma aplicação de visualização científica demandaria. As cenas 1 e 2 podem ser vistas nas Figuras 4.1(a) e 4.1(b) respectivamente.



(4.1(a)) Cena 1. (4.1(b)) Cena 2.

Figura 4.1: Cenas de teste para o modelo de tetraedros.

4.1.1 Grade regular uniforme

Os gráficos das Figuras 4.2 e 4.3 mostram respectivamente o desempenho e o gasto de memória do método proposto nos testes realizados com o modelo de tetraedros utilizando como estrutura de aceleração a grade regular uniforme apresentada na Seção 3.7.1. Nos testes, variamos o fator de resolução k, definido na Equação 3-23, de 0.5 a 4.0.

Como esperado, o desempenho do método melhora com o aumento da resolução da grade, tanto na *cena 1* quanto na *cena 2*. Com o aumento da resolução da grade, suas células diminuem de tamanho e consequentemente o número de elementos em cada célula diminui, o que resulta em um menor número de testes de interseção necessários para cada fragmento. Como podemos observar no gráfico da Figura 4.2, existe uma significativa diferença de desempenho entre a *cena 1* e a *cena 2*. Isso acontece porque, diferentemente da *cena 1*, que apresenta uma visão global do modelo, a *cena 2* abrange apenas a região mais densa do modelo. Os elementos dessa região são menores e por isso, o número de elementos por célula é maior do que no resto do modelo. Logo, o tempo total de renderização da *cena 2* é maior porque esta possui mais fragmentos nessa região.

O gráfico da Figura 4.3 mostra o gasto de memória para as diferentes resoluções testadas. Além de mostrar o gasto total, o gráfico também mostra a distribuição da memória pelos vetores que compõem a estrutura de dados. O vetor de valores de elemento (elemento), o vetor de valores de nós (nós) e o

vetor de planos (planos) formam a estrutura que representa o modelo em GPU e não dependem da resolução da grade. O vetor de células (grade) e o vetor de listas (listas) formam a estrutura da grade regular uniforme. O gráfico mostra o crescimento destes dois vetores com o aumento da resolução.

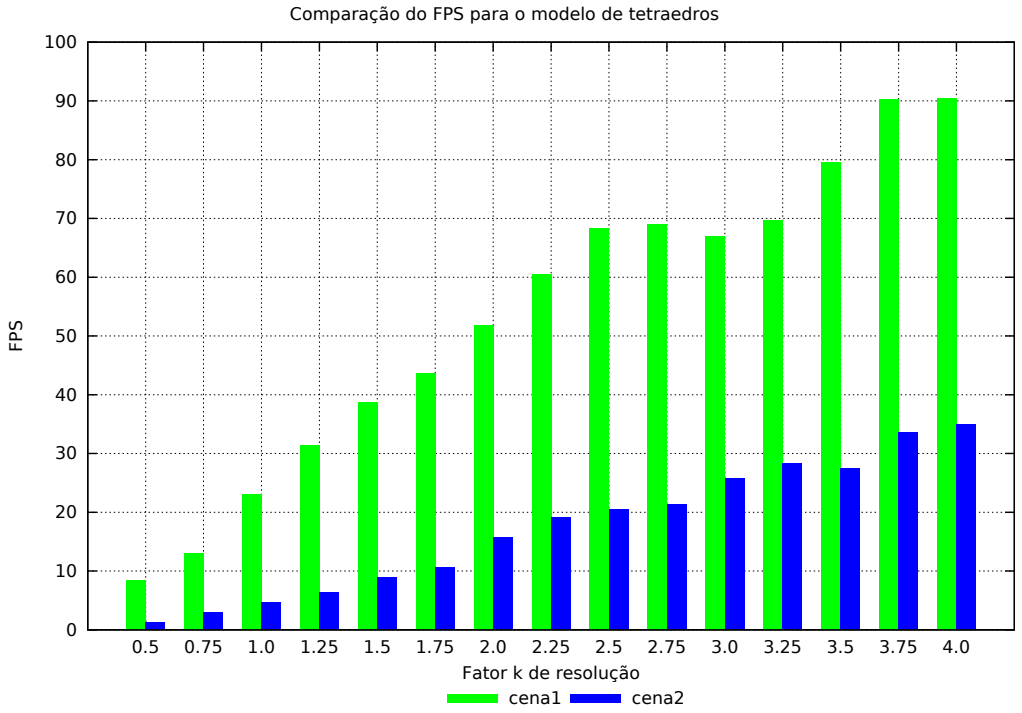


Figura 4.2: Gráfico de desempenho da grade regular uniforme.

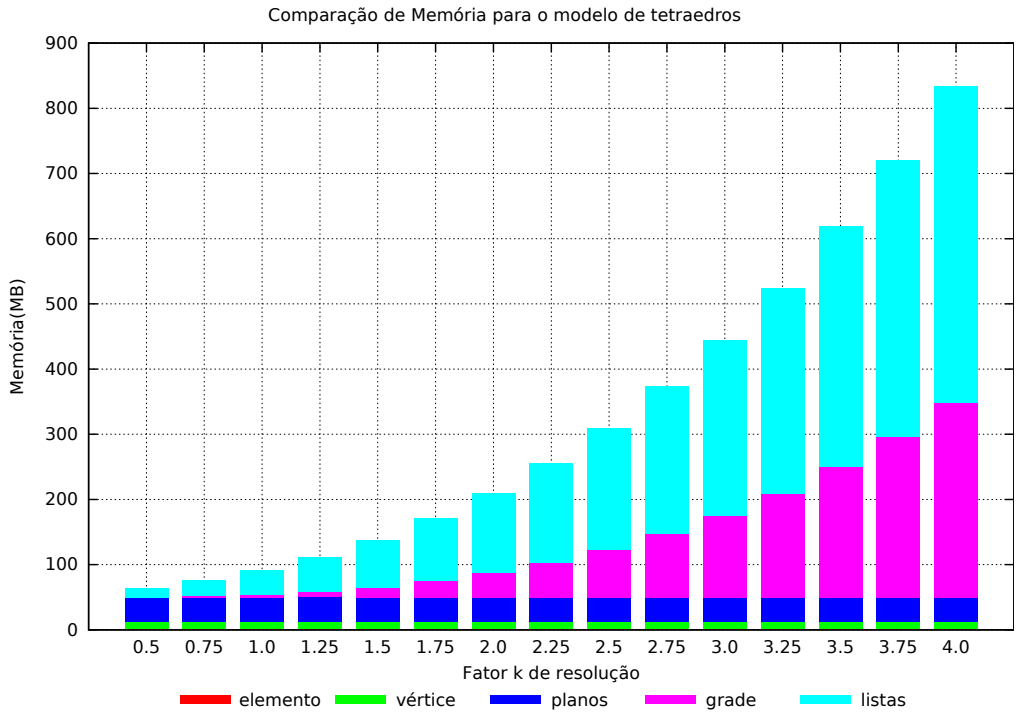


Figura 4.3: Gráfico de gasto de memória da grade regular uniforme.

4.1.2
Grade hierárquica de dois níveis

Os gráficos das Figuras 4.4 e 4.5 mostram respectivamente o desempenho e o gasto de memória do método proposto nos testes realizados com o modelo de tetraedros utilizando como estrutura de aceleração a grade hierárquica de dois níveis apresentada na Seção 3.7.2. Nos testes, variamos de 0.1 a 1.0 o fator k da resolução inicial da grade. A resolução inicial é o primeiro nível da grade hierárquica. Após a construção deste primeiro nível, as células que possuem um número de elementos maior que um determinado limite são refinadas. O limite utilizado nos testes foi de 16 elementos. A dimensão da grade refinada segue a fórmula da Equação 3-24.

O gráfico da Figura 4.4 mostra uma significativa melhora de desempenho da *cena 2* e uma diminuição da diferença de desempenho entre as duas cenas. Isso se deve ao fato da dimensão da grade do refinamento ser função do número de elementos presentes na célula antes desta ser refinada. Assim, quanto mais densa a região em que a célula se encontra, mais refinada ela será. Isso impede que o número de elementos por célula seja muito alto, mesmo em regiões muito densas.

O gráfico da Figura 4.5 mostra o impacto do refinamento no vetor de células (grade) e no vetor de listas (listas) no gasto de memória. Como podemos observar, o vetor de células (grade) permanece pequeno após o refinamento. Isso acontece porque células posicionadas em regiões menos densas do modelo não são refinadas.

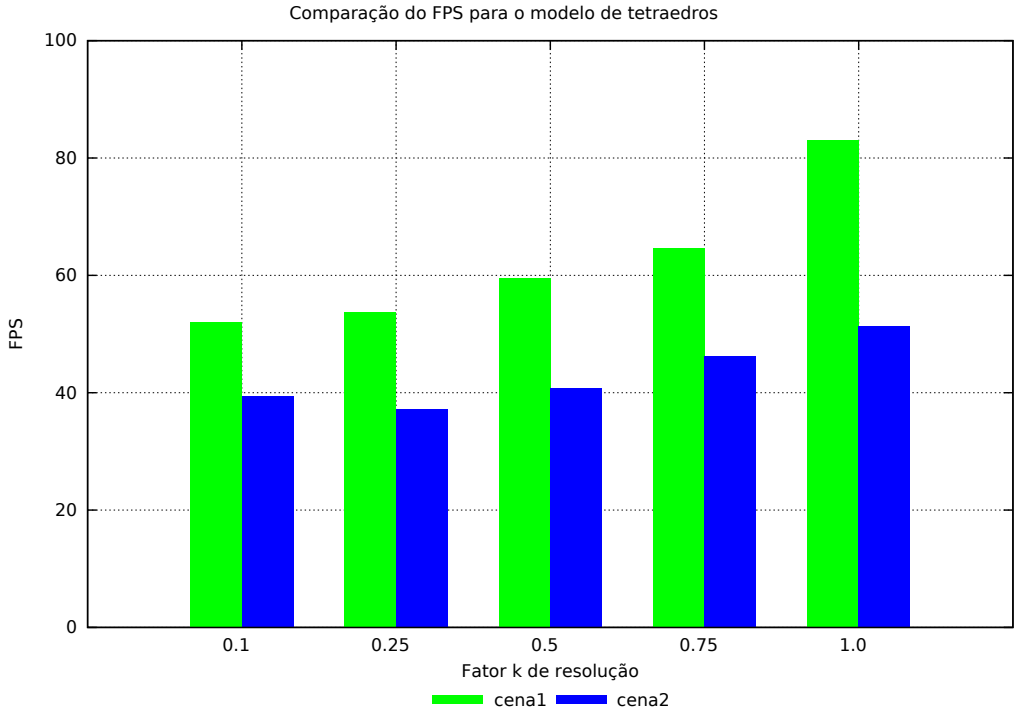


Figura 4.4: Gráfico de desempenho da grade hierárquica de dois níveis.

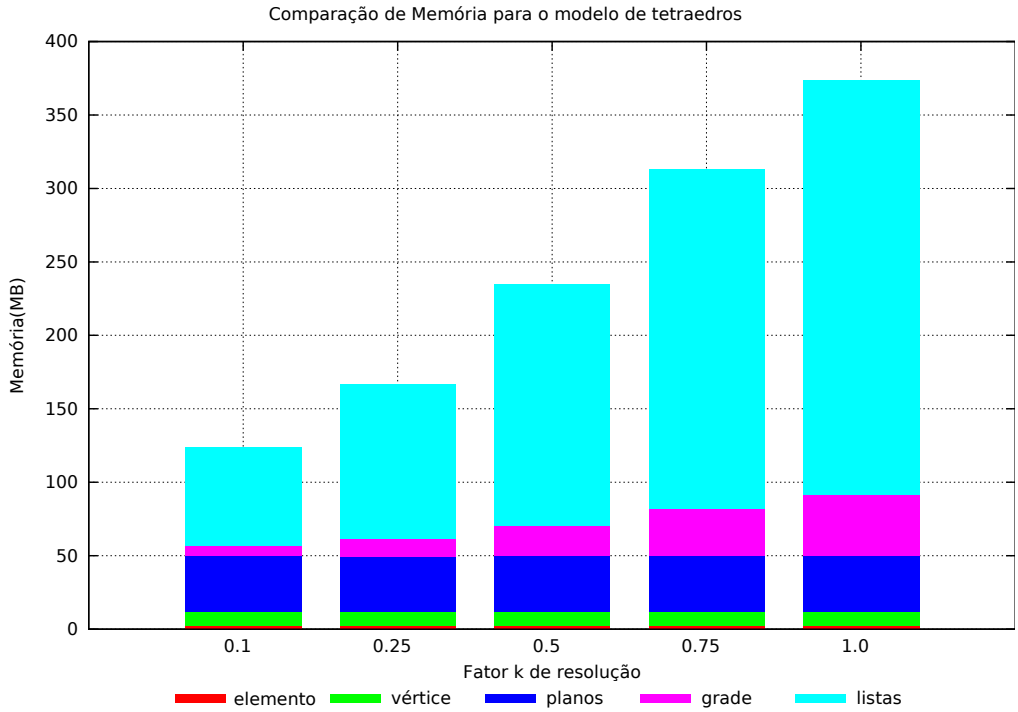


Figura 4.5: Gráfico de gasto de memória da grade hierárquica de dois níveis.

4.1.3 Comparação entre as estruturas

Os gráficos das Figuras 4.6 e 4.7 mostram a comparação de desempenho e gasto de memória entre a grade regular uniforme com o fator de resolução

k igual a 2.75 e a grade hierárquica de dois níveis com o fator de resolução inicial k igual a 1.0. A estrutura grade regular de centróides não foi incluída nesta comparação por apresentar características incompatíveis com este tipo de modelo.

O gráfico de gasto de memória da Figura 4.7 mostra que as duas configurações são equivalentes neste quesito, ambas consomem praticamente a mesma quantidade do recurso. O gráfico de desempenho da Figura 4.6 mostra que a grade hierárquica se saiu melhor nas duas cenas de testes. A diferença de desempenho na *cena 2* foi bastante significativa. Nesta cena, o desempenho da grade hierárquica foi maior que o dobro do desempenho da grade regular. Com base nestes dados podemos afirmar que dentre as estruturas de aceleração apresentadas, a grade hierárquica de dois níveis é mais adequada para este tipo de modelo.

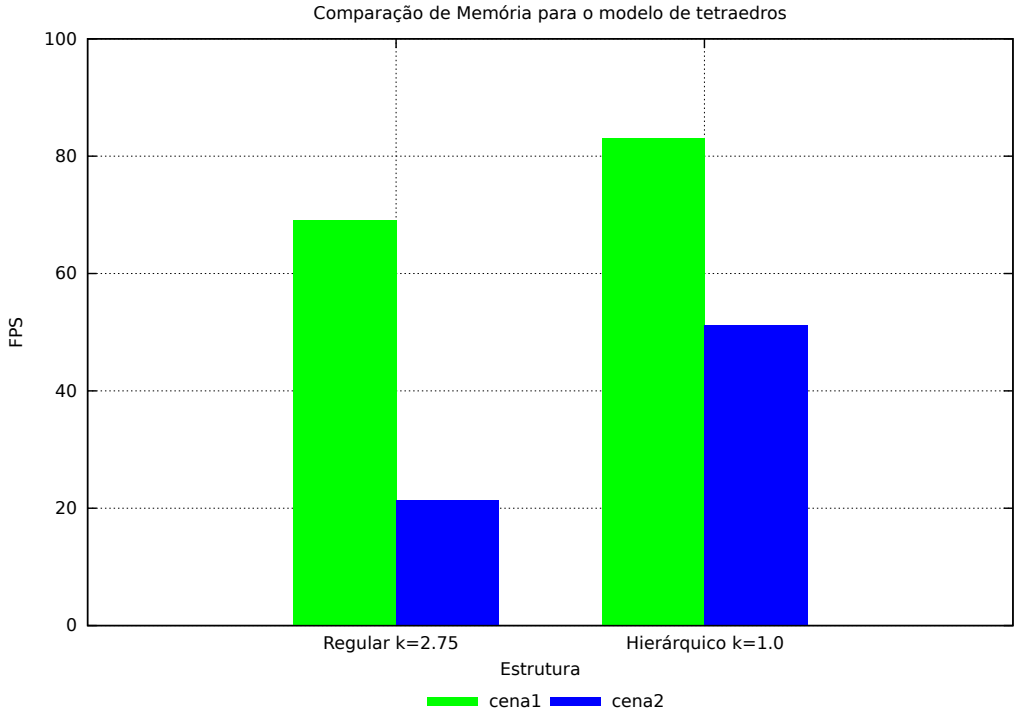


Figura 4.6: Gráfico de desempenho da comparação de estruturas.

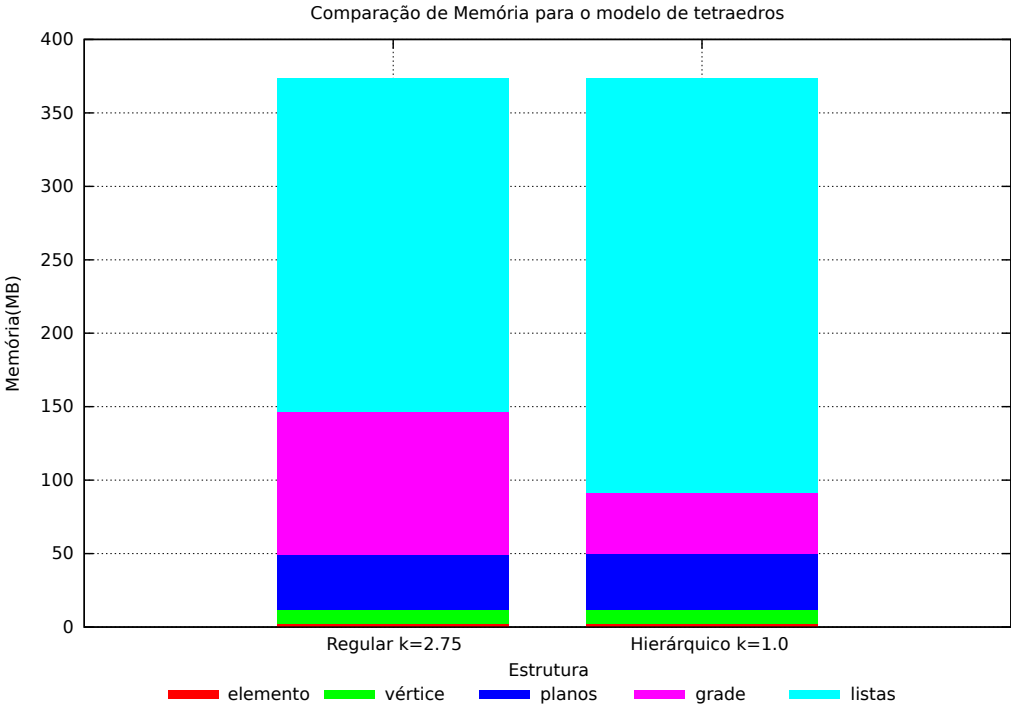


Figura 4.7: Gráfico de gasto de memória da comparação de estruturas.

4.2 Hexaedros

Para ilustrar o desempenho do método e das estruturas de aceleração propostos em modelos de hexaedros, a seguir serão expostos os resultados obtidos em testes utilizando um modelo de hexaedros de um reservatório de petróleo que possui 175.601 elementos ativos. A cena de teste neste caso possui uma reseolução de 1024x1024 e também é composta por 10 planos de corte renderizados de trás para frente. Ao todo são gerados na cena aproximadamente 2.2 milhões de fragmentos, o que constitui um número acima do normal para cenas de aplicações de visualização científica.

4.2.1 Grade regular uniforme

Os gráficos das Figuras 4.8 e 4.9 mostram o desempenho e o gasto de memória nos testes com o modelo de hexaedros utilizando a grade regular uniforme como estrutura de aceleração. Nestes testes, o fator de resolução k foi variado de 0.5 a 4.0.

O gráfico de desempenho da Figura 4.8 mostra que a taxa de quadros por segundo cresce a medida que aumentamos a resolução da grade. Porém, podemos observar que esse crescimento diminui ao longo do gráfico, chegando próximo a zero nas últimas duas resoluções testadas. Já o gráfico de gasto

de memória da Figura 4.9 apresenta um comportamento praticamente oposto. A variação entre o gasto de memória das duas primeiras resoluções é muito pequena. Porém a variação entre duas resoluções consecutivas no gráfico vai aumentando a medida que aumentamos o fator k. Esse comportamento combinado de desempenho e memória se mostra bastante interessante, pois nos possibilita chegar perto do potencial do uso da estrutura de aceleração com relativamente pouco gasto de memória.

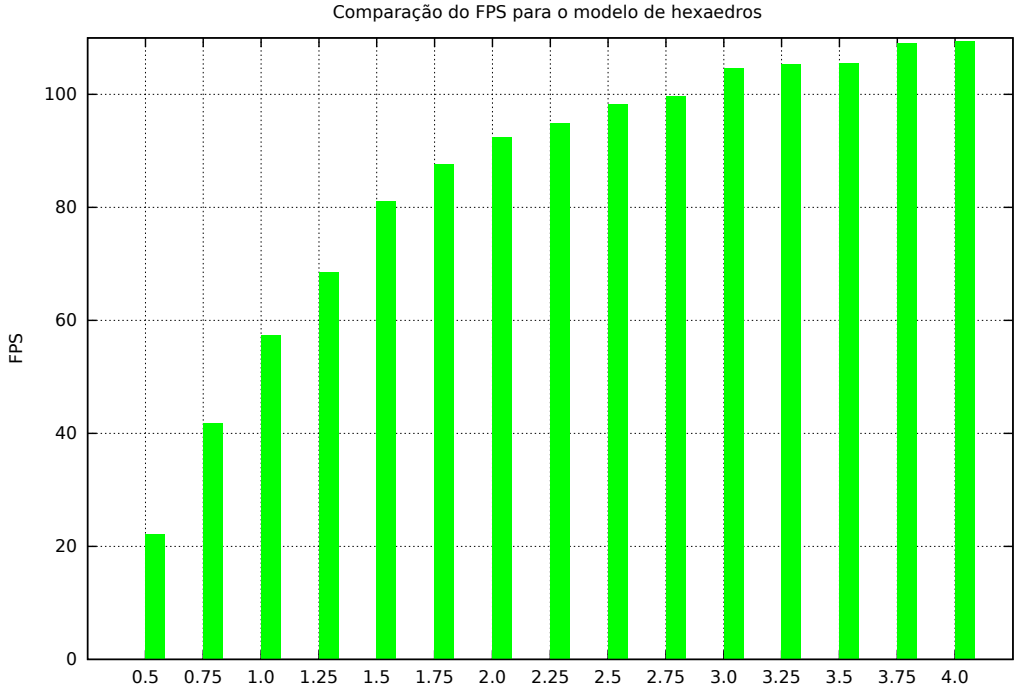


Figura 4.8: Gráfico de desempenho da grade regular uniforme.

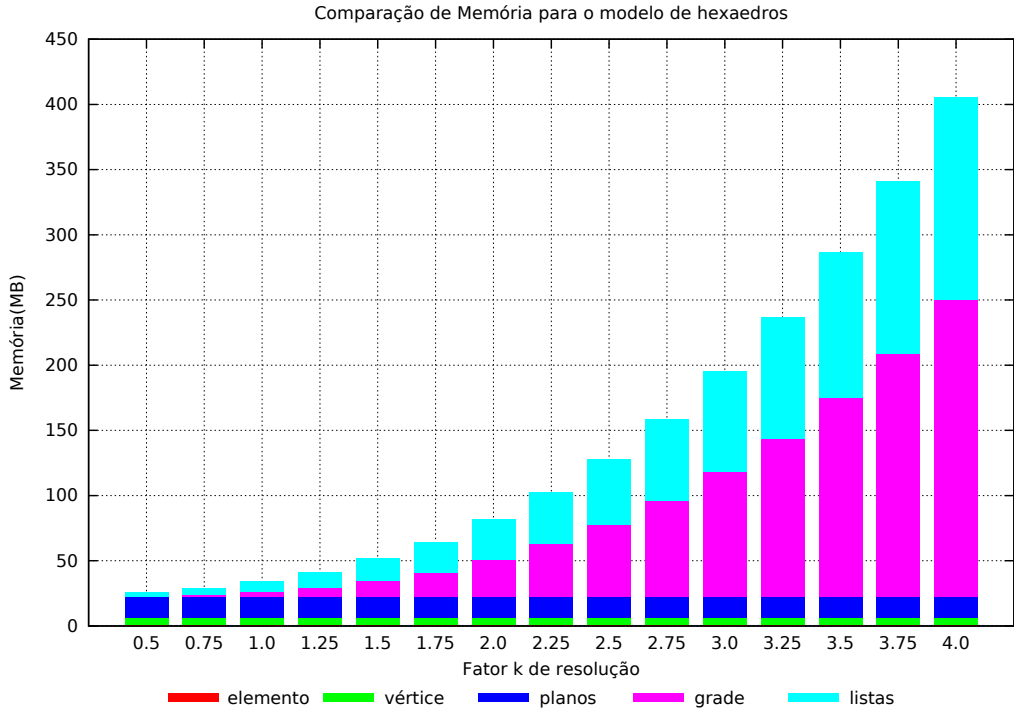


Figura 4.9: Gráfico de gasto de memória da grade regular uniforme.

4.2.2 Grade hierárquica de dois níveis

Os gráficos das Figuras 4.10 e 4.11 mostram respectivamente o desempenho e o gasto de memória nos testes do modelo de hexaedros utilizando como estrutura de aceleração a grade hierárquica de dois níveis. Nos testes variamos de 0.1 a 1.0 o fator k da resolução inicial da grade.

Os gráficos mostram que os testes utilizando a grade hierárquica atingiram boas taxas de quadros por segundo, a maior parte acima de 60 quadros por segundo, sem apresentar um gasto excessivo de memória. Porém não houve nenhum ganho real em relação aos dados conseguidos com a grade regular uniforme. Isso acontece porque, ao contrário do modelo de tetraedros utilizado, este modelo não apresenta uma variação muito grande no tamanho de seus elementos.

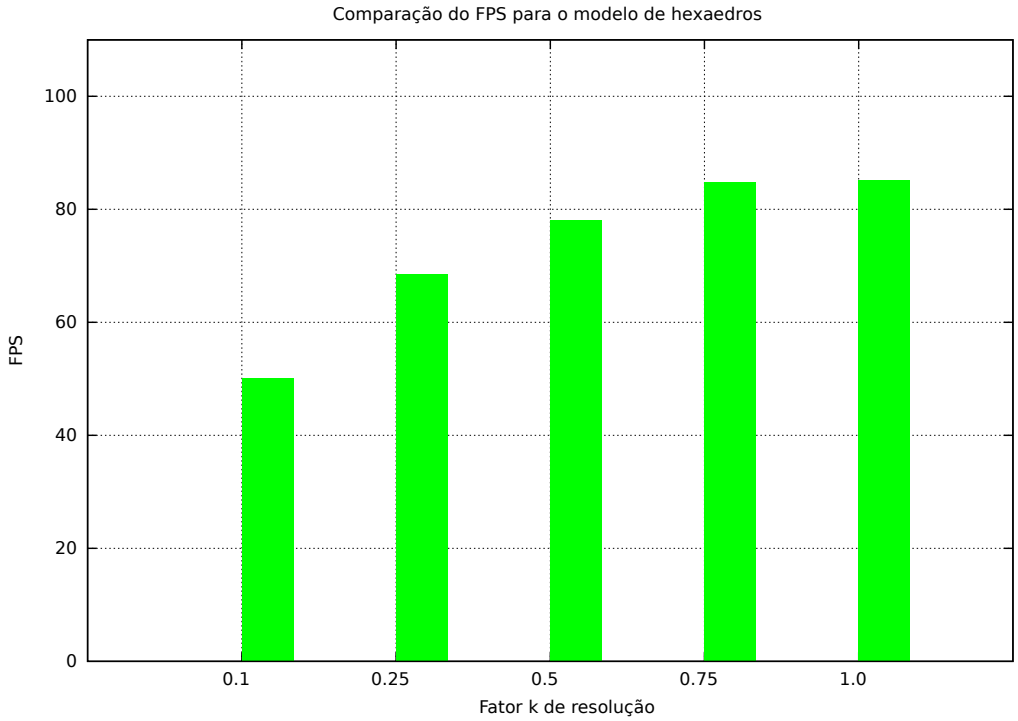


Figura 4.10: Gráfico de desempenho da grade hierárquica de dois níveis.

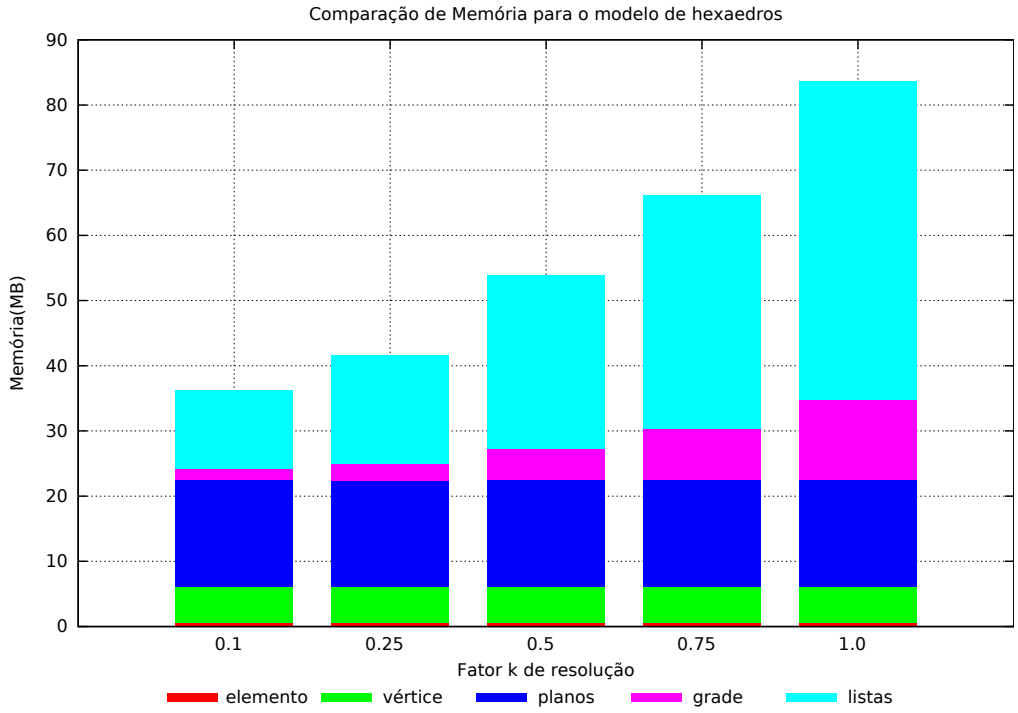


Figura 4.11: Gráfico de gasto de memória da grade hierárquica de dois níveis.

4.2.3 Comparação entre as estruturas

Os gráficos das Figuras 4.12 e 4.13 mostram a comparação de desempenho e gasto de memória entre as estruturas apresentadas neste trabalho. Nesta

seção incluímos na comparação a grade regular de centróides apresentada na Seção 3.7.3. Estão presentes também na comparação a grade regular uniforme com fator de resolução k igual a 0.25, usada como parâmetro de comparação da grade de centróides, a grade regular uniforme com fator de resolução k igual a 1.75, e a grade hierárquica com fator de resolução inicial k igual a 0.75.

Analisando os gráficos podemos perceber que a grade regular de centróides apresenta um desempenho melhor do que a configuração de grade regular equivalente em termos de gasto de memória (k igual a 0.25). Porém, esse desempenho foi muito abaixo do melhor desempenho apresentado pelas demais estruturas.

Na comparação entre a configuração de grade regular com o fator de resolução k igual a 1.75 e a configuração de grade hierárquica com k igual a 0.75 podemos observar que a configuração de grade regular apresenta uma leve vantagem no desempenho utilizando menos memória que a configuração de grade hierárquica. Com base nestes dados podemos concluir que a grade regular uniforme é mais adequada a este tipo de modelo do que as demais estruturas apresentadas neste trabalho.

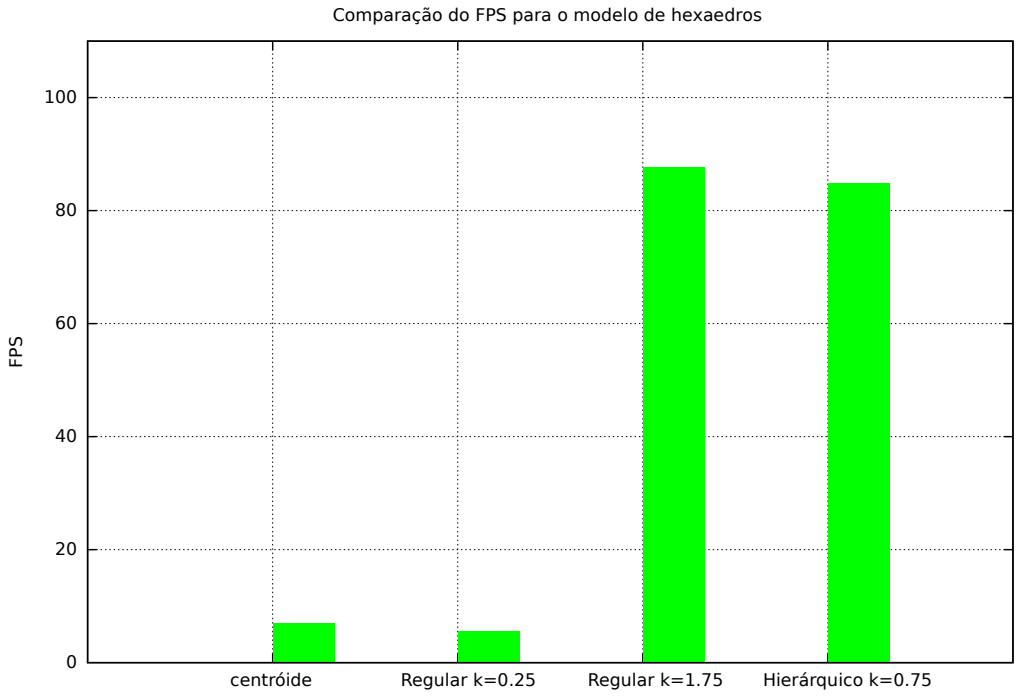


Figura 4.12: Gráfico de desempenho da comparação de estruturas.

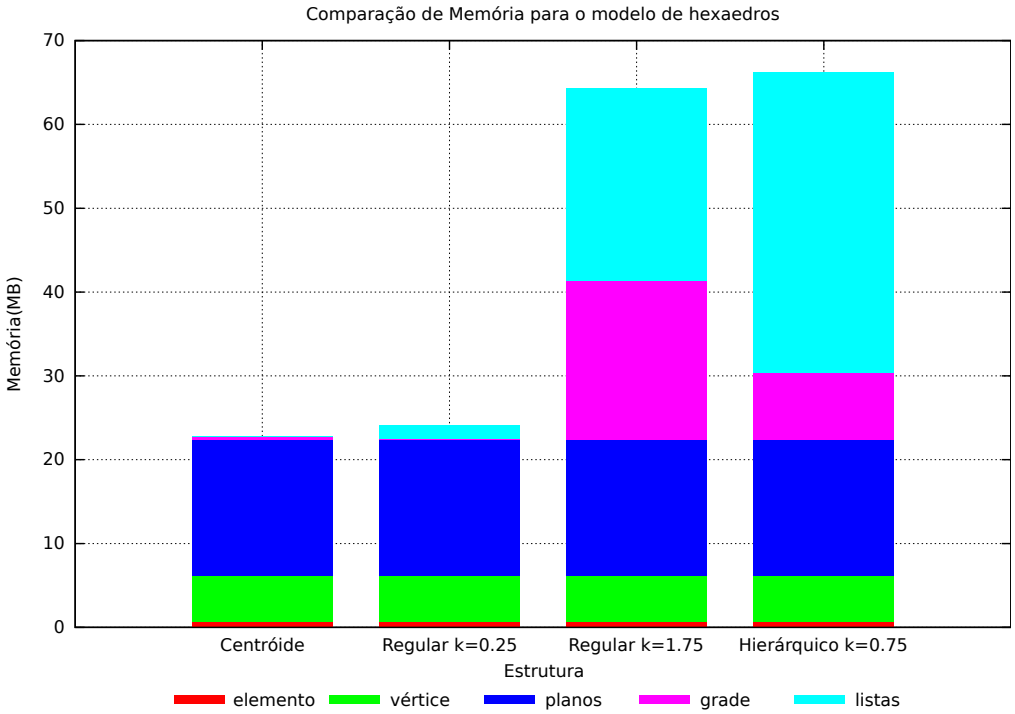
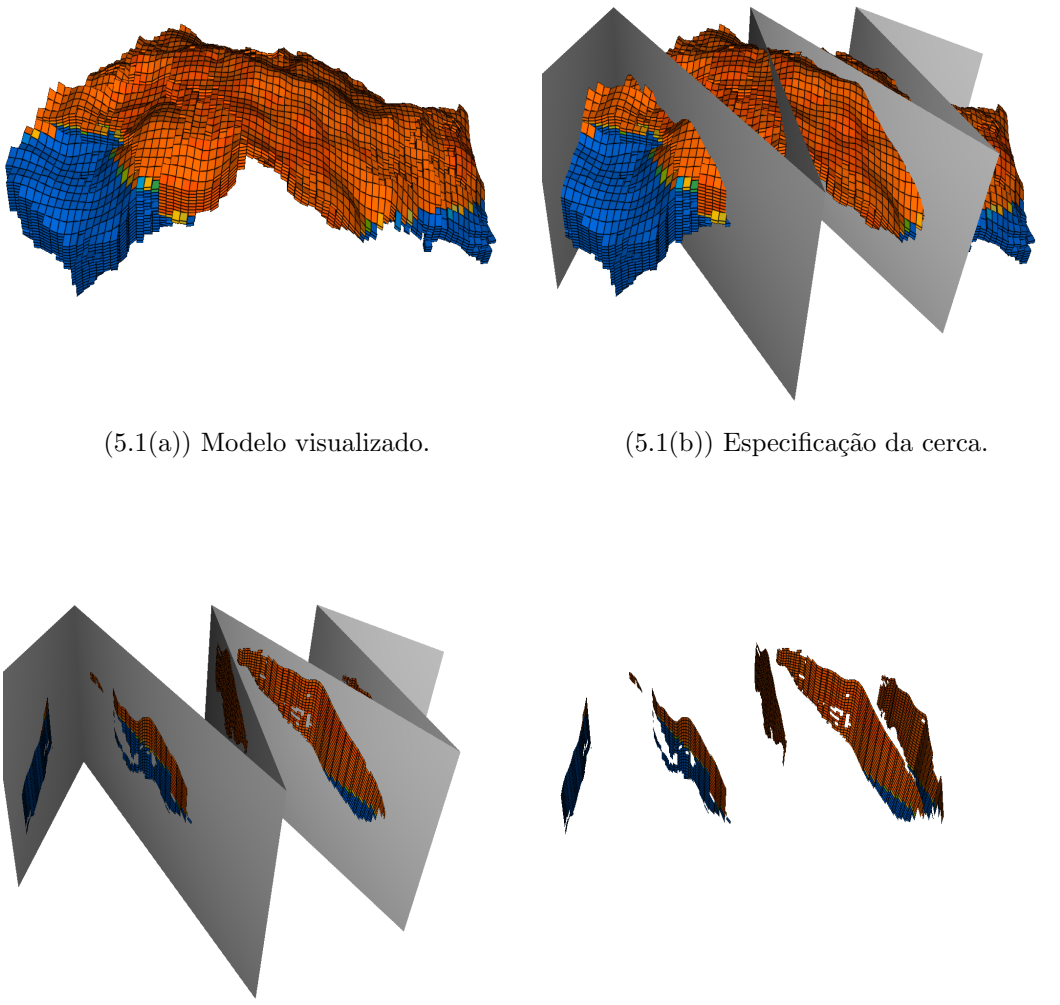


Figura 4.13: Gráfico de gesto de memória da comparação de estruturas.

5
Aplicações

Este capítulo apreseta brevemente alguns modos de visualização que tiram proveito do método proposto.



(5.1(a)) Modelo visualizado.

(5.1(b)) Especificação da cerca.

(5.1(c)) Visualização da cerca.

(5.1(d)) Visualização apenas da interseção.

Figura 5.1: Diagrama de cerca.

5.1
Diagrama de cerca

O diagrama de cerca é formado por cortes planares verticais que possibilitam a inspeção da distribuição do campo e da geometria no interior do modelo. A Figura 5.1 ilustra a ferramenta.

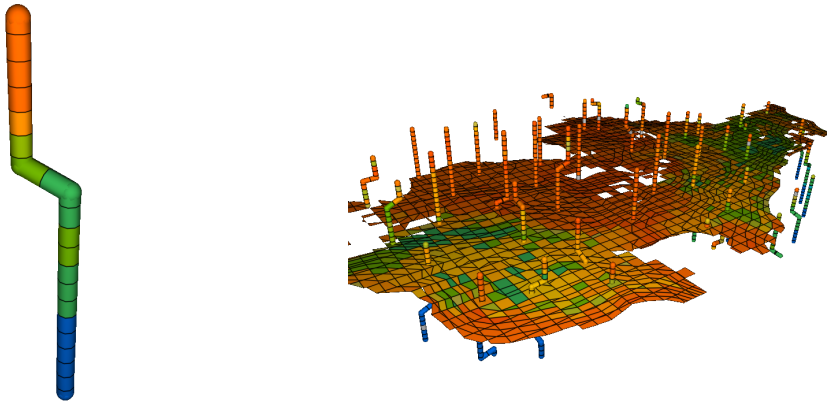
5.2
Visualização de poços em modelos de reservatório de petróleo

Um modelo de reservatório de petróleo é um modelo de simulação definido por uma malha discreta de elementos hexaédricos não estruturada. As propriedades do modelo apresentam um valor por elemento e são baseadas em informações geofísicas e geológicas. Além dos próprios elementos do modelo, a simulação também faz uso de objetos que representam poços de petróleo. Os poços são representados por uma lista ordenada de completações, que são pontos de extração ou injeção de fluidos e são sempre posicionados no meio de algum elemento ativo do modelo. Dada uma configuração de poços, o simulador calcula o fluxo dos fluidos, como óleo, gás e água com base no seu modelo numérico. Dito isso, é possível se constatar que a informação relacionada aos elementos nas proximidades dos poços pode ser bastante relevante para o entendimento dos resultados de uma simulação. Sendo assim, serão apresentados a seguir alguns modos de visualização que utilizam o método proposto e auxiliam no entendimento deste tipo de informação.

5.2.1
Visualização de poços como objetos de corte

Uma forma eficaz de se visualizar o valor de uma determinada propriedade nos elementos interceptados por um poço juntamente com a disposição tridimensional de suas completações é utilizar o próprio poço como objeto de corte. Essa técnica permite não só a análise de poços individuais (Figura 5.2(a)) como também pode proporcionar uma visão global dos poços do modelo, que se combinada com outras técnicas de visualização, como planos de corte (Figura 5.2(b)), facilita a compreensão da influência da disposição dos poços nos resultados da simulação.

Na simulação, um poço é definido por uma lista ordenada de pontos. Isso, na prática, faz com que neste caso o objeto de corte seja uma linha em vez de uma superície. Como explicado nas Seções 3.3 e 3.5, o método proposto suporta o uso de linhas como objetos de corte e mapeia o resultado deste tipo de corte sobre uma superfície arbitrária. O objeto de corte é passado como coordenada de textura dos vértices da superfície renderizada e é utilizado no cálculo de interseção com o modelo. Com o objetivo de simular o formato de um poço real, a superfície utilizada neste modo de visualização é um objeto cilíndrico que segue a trajetória das completações do poço.



(5.2(a)) Análise individual de um poço. (5.2(b)) Visualização de todos os poços.

Figura 5.2: Visualização de poços como objetos de corte.

5.2.2 Análise de perfil de poço

A visualização do perfil de um poço é uma vista unidimensional dos elementos interceptados pelo poço. A estrutura tridimensional do poço é desconsiderada para possibilitar que todos os elementos interceptados sejam visualizados. Esse modo de visualização costuma ser usado junto com gráficos de análise de propriedades de poço para ilustrar a variação de propriedades dos elementos interceptados pelo poço ao longo do tempo. Assim como no procedimento anterior, a renderização consiste em utilizar o próprio poço como objeto de corte, que é enviado ao pipeline como coordenada de textura dos vértices da superfície renderizada, que neste caso é um retângulo. A Figura 5.3 mostra a visualização do perfil de um poço em diferentes momentos da simulação.

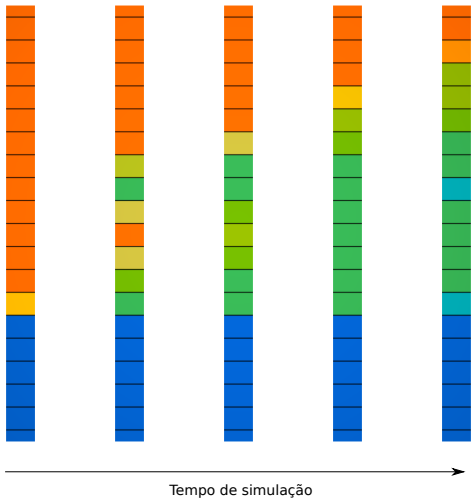
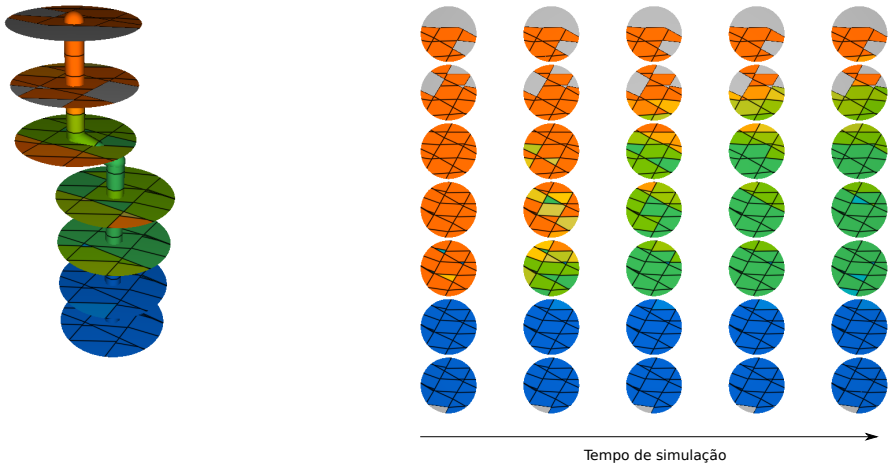


Figura 5.3: Análise de perfil de poço.

5.2.3
Seções horizontais circulares

Para se ter uma visão um pouco mais abrangente da influência de um poço no modelo pode ser interessante se inspecionar não só os elementos interceptados pelo poço, mas também os elementos nas suas proximidades. Isso pode ser feito utilizando-se seções horizontais circulares de corte ao longo da trajetória do poço. Essas seções de corte podem ser visualizadas dentro do contexto global do modelo ou de maneira similar à visualização do perfil de poços. As Figura 5.4 exemplifica ambos os modos de visualização.



(5.4(a)) Seções visualizadas em suas posições de corte.

(5.4(b)) Seções visualizadas na forma de perfil de poço.

Figura 5.4: Seções horizontais circulares.

5.3 Recorte de volumes convexos

Recorte de volumes é uma ferramenta interessante para o entendimento de dados volumétricos tridimensionais. A técnica consiste em separar do modelo pedaços delimitados por um determinado volume convexo. Isto permite a inspeção de elementos que se encontram no interior do domínio sem se perder a noção da geometria global do modelo. A Figura 5.5 ilustra o uso da técnica.

A renderização desta técnica de visualização é dividida em duas partes: a renderização do pedaço recortado pelo volume de corte e a renderização do restante modelo. A renderização de ambas as partes consiste em renderizar parte da superfície externa do modelo juntamente com a interseção da superfície do volume de recorte com o modelo, que é desenhada utilizando-se o método proposto nos capítulos anteriores. A única diferença entre esses dois processos é qual parte da superfície externa do modelo deve ser renderizada. No caso do pedaço recortado, devemos renderizar a parte da superfície externa do modelo que se encontra no interior do volume de recorte e no caso do restante do modelo devemos renderizar a parte da superfície que se encontra fora do volume de recorte. As Figuras 5.5(c) e 5.5(d) exemplificam as duas etapas. Em ambos os casos a superfície externa do modelo é enviada ao pipeline por inteiro e no *fragment shader* é testado para cada fragmento gerado se este se encontra no interior do volume de recorte. Se o pedaço recortado estiver sendo desenhado, os fragmentos exteriores ao volume de corte são descartados; se o restante do modelo estiver sendo desenhado, os fragmentos no interior do volume de corte são descartados. Se traçarmos um raio do observador em direção a um volume convexo, para qualquer ângulo de visão esse raio interceptará a superfície do volume no máximo 2 vezes, ou seja, caso intercepte o volume, o raio entrará e sairá deste apenas uma vez. Assim para verificar se um fragmento está no interior do volume basta conferir se ele está a frente do ponto de entrada e atrás do ponto de saída. Isso é feito comparando a profundidade do fragmento da supefície externa do modelo com a profundidade da *front face* e da *back face* da superfície do volume nas coordenadas de tela do fragmento. Por isso, antes da renderização da superfície externa do modelo, a *front face* e a *back face* da superfície do volume de corte são renderizadas e seus valores de profundidade armazenados em duas texturas que são acessadas durante a renderização da superfície do modelo.

Resumidamente o algoritmo de recorte de volumes convexos transcorre da seguinte maneira:

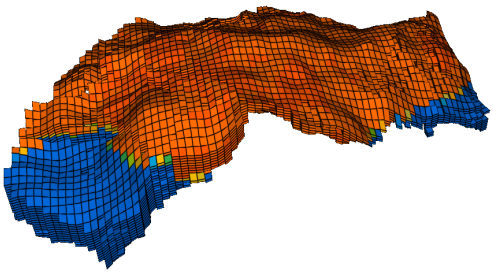
– Renderização do pedaço recortado:

1. Renderiza a *front face* e a *back face* da superfície do volume de corte e armazena os valores de profundidade em duas texturas.
2. Renderiza a superfície externa do modelo descartando os fragmentos que se encontram fora do volume de corte.
3. Renderiza a interseção entre a superfície do volume e o modelo utilizando o método proposto.

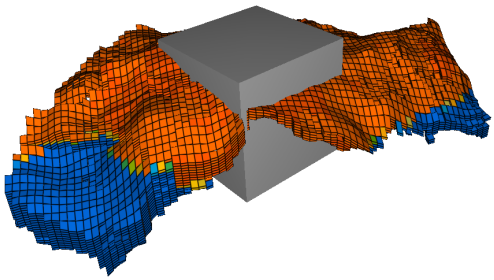
– Renderização do restante do modelo:

1. Renderiza a *front face* e a *back face* da superfície do volume de corte e armazena os valores de profundidade em duas texturas.
2. Renderiza a superfície externa do modelo descartando os fragmentos que se encontram no interior do volume de corte.
3. Renderiza a interseção entre a superfície do volume e o modelo utilizando o método proposto.

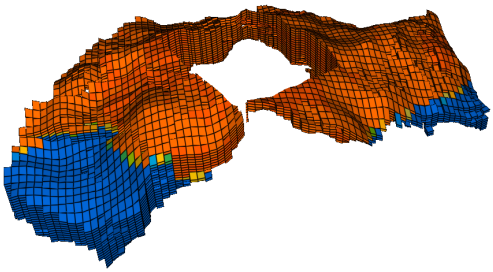
A desvantagem dessa implementação é a necessidade de se alterar o algoritmo de renderização do modelo, pois é preciso que o desenho do modelo dê suporte ao descarte de fragmentos como foi explicado anteriormente. Porém, o uso do método proposto no cálculo de interseção faz com que a ferramenta seja de fácil implementação e apresente um bom desempenho.



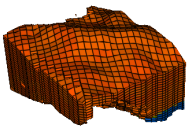
(5.5(a)) Modelos visualizado.



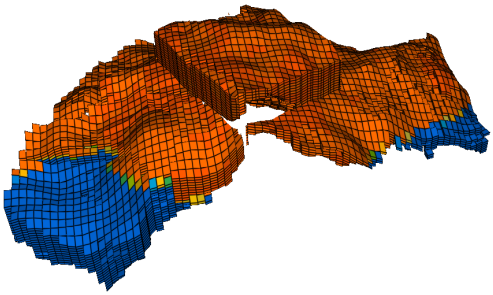
(5.5(b)) Volume de corte posicionado.



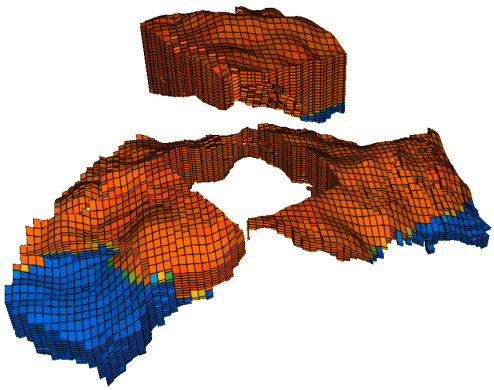
(5.5(c)) Restante do modelo.



(5.5(d)) Pedaço recortado.



(5.5(e)) Exemplo de recorte.



(5.5(f)) Exemplo de recorte.

Figura 5.5: Recorte de volumes.

6
Conclusão

Este trabalho apresentou um método que utiliza programação em GPU para realizar a renderização direta de superfícies de corte em malhas não estruturadas. A idéia básica do método proposto é utilizar o rasterizador da placa gráfica para gerar os fragmentos da superfície de corte e calcular a interseção de cada fragmento com o modelo em GPU. Para garantir eficiência, foram testadas, como estrutura de aceleração, três variações de grades regulares para armazenar os elementos da malha, e cada elemento é representado pela lista de planos de suas faces, facilitando o teste de interseção fragmento-elemento. Uma vez determinado o elemento que contém o fragmento, são aplicados procedimentos para interpolar o campo escalar e para identificar se o fragmento está próximo à fronteira do elemento, a fim de representar o aramado (*wireframe*) da malha na superfície de corte. O método ainda desacopla a superfície de corte (usada na interseção com o modelo) da superfície renderizada e possibilita o uso de linhas como objetos de corte.

Os resultados mostram que o método conseguiu produzir imagens em tempo real para todas as cenas de testes. Com base nos dados gerados podemos concluir que a grade hierárquica de dois níveis é mais adequada a modelos que possuem uma variação grande de densidade de elementos enquanto a grade uniforme se mostrou melhor para modelos com pouca variação de densidade. A grade de centróides, mesmo se mostrando mais eficiente do que a configuração de grade uniforme equivalente em termos de memória, não apresentou desempenho equiparável às configurações mais eficientes das outras duas estruturas.

Foram implementados também alguns modos de visualização que se beneficiam do método proposto para exemplificar o seu uso em aplicações de visualização científica. Entre esses modos estão o diagrama de cerca, recorte de volumes convexos e visualizações de poços em reservatórios de petróleo.

Como trabalhos futuros pretendemos fazer um estudo de escalabilidade e investigar a extensão do método para modelos de elementos não lineares (*high order*). Acreditamos que esta abordagem baseada na interseção de fragmentos com o modelo pode se mostrar mais eficiente do que o uso de traçado de raios apresentado em (16).

7
Referências Bibliográficas

[1] AKENINE-MÖLLER, T.; HAINES, E. ; HOFFMAN, N. **Real-Time Rendering 3rd Edition**. Natick, MA, USA: A. K. Peters, Ltd., 2008, 1045p.

[2] B, J. A.; NIELSEN, S. L.; GJØL, M. ; LARSEN, B. D. **Two methods for antialiased wireframe drawing with hidden line removal**. Em: PROCEEDINGS OF THE 24TH SPRING CONFERENCE ON COMPUTER GRAPHICS, SCCG '08, p. 171–177, New York, NY, USA, 2010. ACM.

[3] BAILEY, M. Using gpu shaders for visualization, **Computer Graphics and Applications, IEEE**, v.29, n.5, p. 96 –100, sept.-oct. 2009.

[4] BRASHER, M.; HAIMES, R. **Rendering planar cuts through quadratic and cubic finite elements**. Em: VISUALIZATION, 2004. IEEE, p. 409 – 416, oct. 2004.

[5] CAZALS, F.; DRETTAKIS, G. ; PUECH, C. **Filtering, clustering and hierarchy construction: a new solution for ray-tracing complex scenes**. Em: COMPUTER GRAPHICS FORUM, volume 14, p. 371–382. Wiley Online Library, 1995.

[6] CELES, W.; ABRAHAM, F. **Texture-based wireframe rendering**. Em: GRAPHICS, PATTERNS AND IMAGES (SIBGRAPI), 2010 23RD SIBGRAPI CONFERENCE ON, p. 149–155. IEEE, 2010.

[7] CELES, W.; ABRAHAM, F. Fast and versatile texture-based wireframe rendering, **The Visual Computer**, v.27, p. 939–948, 2011.

[8] ENGEL, K.; HADWIGER, M.; KNISS, J. M.; LEFOHN, A. E.; SALAMA, C. R. ; WEISKOPF, D. **Real-time volume graphics**. Em: ACM SIGGRAPH 2004 COURSE NOTES, p. 29. ACM, 2004.

[9] FOLEY, T.; SUGERMAN, J. **Kd-tree acceleration structures for a gpu raytracer**. Em: PROCEEDINGS OF THE ACM SIGGRAPH/EUROGRAPHICS CONFERENCE ON GRAPHICS HARDWARE, p. 15–22. ACM, 2005.

[10] GATEAU, S. Solid wireframe, **NVIDIA Whitepaper WP-03014-001 v01, February**, 2007.

[11] IVSON, P.; DUARTE, L. ; CELES, W. Gpu-accelerated uniform grid construction for ray tracing dynamic scenes, **Master's thesis, Departamento de Informatica, Pontificia Universidade Catolica, Rio de Janeiro**, 2009.

[12] KALOJANOV, J.; SLUSALLEK, P. **A parallel algorithm for construction of uniform grids**. Em: PROCEEDINGS OF THE CONFERENCE ON HIGH PERFORMANCE GRAPHICS 2009, p. 23–28. ACM, 2009.

[13] LAGAE, A.; DUTRÉ, P. **Compact, fast and robust grids for ray tracing**. Em: COMPUTER GRAPHICS FORUM, volume 27, p. 1235–1244. Wiley Online Library, 2008.

[14] LAUTERBACH, C.; GARLAND, M.; SENGUPTA, S.; LUEBKE, D. ; MANOCHA, D. **Fast bvh construction on gpus**. Em: COMPUTER GRAPHICS FORUM, volume 28, p. 375–384. Wiley Online Library, 2009.

[15] MIRANDA, F. M.; CELES, W. Volume rendering of unstructured hexahedral meshes, **The Visual Computer**, p. 1–10, 2012.

[16] NELSON, B.; HAIMES, R. ; KIRBY, R. Gpu-based interactive cut-surface extraction from high-order finite element fields, **Visualization and Computer Graphics, IEEE Transactions on**, v.17, n.12, p. 1803 – 1811, dec. 2011.

[17] NELSON, B.; LIU, E.; KIRBY, R. ; HAIMES, R. Elvis: A system for the accurate and interactive visualization of high-order finite element solutions, **Visualization and Computer Graphics, IEEE Transactions on**, v.18, n.12, p. 2325–2334, Dec.

[18] WALD, I.; IZE, T.; KENSLER, A.; KNOLL, A. ; PARKER, S. G. Ray tracing animated scenes using coherent grid traversal, **ACM Trans. Graph.**, v.25, n.3, p. 485–493, Jul 2006.

[19] WEISKOPF, D.; ENGEL, K. ; ERTL, T. **Volume clipping via per-fragment operations in texture-based volume visualization**. Em: PROCEEDINGS OF THE CONFERENCE ON VISUALIZATION'02, p. 93–100. IEEE Computer Society, 2002.

[20] WEISKOPF, D.; ENGEL, K. ; ERTL, T. Interactive clipping techniques for texture-based volume visualization and volume shading, **Visualization and Computer Graphics, IEEE Transactions on**, v.9, n.3, p. 298–312, 2003.