Pontifícia Universidade Católica
do Rio de Janeiro

Gustavo Nunes Wagner

Visualização Interativa de Modelos Massivos de Engenharia na Indústria de Petróleo com o Algoritmo de Voxels Distantes

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do título de Mestre pelo Programa de Pós-Graduação em Informática da PUC-Rio.

Orientador: Prof. Marcelo Gattass
Co-Orientador: Prof. Alberto Raposo

Rio de Janeiro,
Abril de 2007

Gustavo Nunes Wagner

Visualização Interativa de Modelos Massivos de Engenharia na Indústria de Petróleo com o Algoritmo de Voxels Distantes

Dissertação apresentada como requisito parcial para obtenção do título de Mestre pelo Programa de Pós-Graduação em Informática da PUC-Rio. Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Marcelo Gattass
Orientador
Departamento de Informática - PUC-Rio

Prof. Alberto Raposo
Co-Orientador
Departamento de Informática - PUC-Rio

Prof. Marcelo Dreux
Departamento de Informática - PUC-Rio

Dr. Luciano Reis
CENPES - Petrobras

Prof. Waldemar Celes
Departamento de Informática - PUC-Rio

Prof. José Eugenio Leal
Coordenador Setorial do Centro
Técnico Científico - PUC-Rio

Rio de Janeiro, 9 de abril de 2007

Gustavo Nunes Wagner

Engenheiro de Computação graduado pela Pontifícia Universidade Católica do Rio de Janeiro em dezembro de 2004. No semestre seguinte à graduação entrou para o Programa de Pós-graduação em Informática na mesma universidade.

Ficha Catalográfica

Wagner, Gustavo Nunes

Visualização interativa de modelos massivos de engenharia na indústria de petróleo com o algoritmo de voxels distantes / Gustavo Nunes Wagner ; orientador: Marcelo Gattass ; co-orientador: Alberto Raposo. – 2007.

88f. : il. ; 30 cm

Dissertação (Mestrado em Informática)–Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2006.

Inclui referências bibliográficas.

1. Informática – Teses. 2. Computação Gráfica. 3. Visualização. 4. Modelos Massivos. 5. Oclusão. 6. LOD Hierárquico. 7. Impostores. I. Gattass, Marcelo. II. Raposo, Alberto. III. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. IV. Título.

CDD: 004

Agradecimentos

À minha família.

Aos meus orientadores Marcelo Gattass e Alberto Raposo, por todas as idéias e conselhos que foram indispensáveis na elaboração dessa dissertação.

Ao Tecgraf, por ser uma ótima fonte de problemas difíceis e interessantes e um lugar único para trabalhar.

À CAPES, pelo auxílio dado ao longo do curso.

À PUC-Rio e aos seus professores que ensinam com o mesmo empenho tanto as matérias mais simples quanto os assuntos mais complexos.

Ao Instituto de Tecnologia ORT, por ter me ensinado sobre linguagens de programação.

Resumo

Wagner, Gustavo Nunes; Gattass, Marcelo. **Visualização Interativa de Modelos Massivos de Engenharia na Indústria de Petróleo com o Algoritmo de Voxels Distantes.** Rio de Janeiro, 2007. 88p. Dissertação de Mestrado - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Projetos recentes de Estruturas *Offshore* criaram a necessidade de prototipação virtual de modelos de CAD massivos. Esses modelos tipicamente têm centenas de milhões de triângulos e, por essa razão, não podem ser enviados diretamente para as placas gráficas atuais que podem renderizar interativamente apenas alguns milhões de triângulos. Existem várias abordagens para lidar com esse problema incluindo uma nova estratégia de uso de impostores baseada na visualização de Voxels. Essa estratégia é promissora, já que lida bem com níveis de detalhe, oclusão e armazenamento em memória secundária. Esta dissertação apresenta uma variação do algoritmo de Voxels Distantes (*Far Voxels)*, que é implementada e testada sobre modelos de CAD típicos. Finalmente, a partir desses testes, a dissertação apresenta algumas conclusões e sugestões para trabalhos futuros.

Palavras-chave

Computação Gráfica; Visualização; Modelos Massivos; Oclusão; LOD Hierarquico; Impostores

Abstract

Wagner, Gustavo Nunes; Gattass, Marcelo. **Interactive Visualization of Massive Engineering Models in the Oil & Gas Industry using the Far Voxels Algorithm**. Rio de Janeiro, 2007. 88p. MSc. Dissertation - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Current projects of Offshore Structures require virtual prototyping of huge CAD models. These models usually have hundreds of millions of triangles and for this reason they cannot be sent directly to current graphical boards that can render interactively only a few millions of triangles. There are many different approaches to deal with this problem including a new impostor strategy based on Voxel visualization. This strategy is promising because it deals well with level of detail, occlusion and out of core model storage. This dissertation presents a variant of the Far Voxels algorithm. This variant is implemented and tested against typical CAD models. Finally, from these tests, the dissertation presents some conclusions and suggestions for future work.

Keywords

Computer Graphics; Visualization; Massive Models; Occlusion; Hierarchical LODs; Impostors

Sumário

Lista de Figuras

Lista de Tabelas

1
Introdução

Indústrias de larga escala como a automobilística, aeroespacial e do petróleo estão investindo na construção de sistemas de informação integrados para controlar seus empreendimentos. Nestes sistemas, o acesso à informação necessária é geralmente feito com auxilio da visualização 3D dos modelos dos objetos envolvidos. Essa visualização pode auxiliar em processos de projeto, prototipação virtual, análise, controle de processos e treinamento, entre outras atividades. O objetivo desse trabalho é estudar uma forma eficiente de visualizar esses modelos, de forma a permitir que um eventual usuário interaja e realize atividades sobre ele.

Visualizar iterativamente modelos completos de engenharia é ainda um problema em aberto. Atualmente, modelos CAD raramente são criados levando-se em consideração que também serão usados em uma visualização 3D. Como conseqüência, muitos objetos acabam sendo representados de forma esquemática, sem nenhum compromisso com a sua forma real. Por outro lado, muitas vezes esses objetos são criados com detalhes que são importantes para a montagem ou operação do equipamento representado, mas que não são necessários para a visualização, e acabam se tornando um grande gargalo para a visualização, especialmente a visualização em tempo-real. Incluir no modelo de Engenharia as informações de textura e iluminação pode ser um processo caro e lento. Processos automáticos de geração de cores, textura e iluminação são ainda um problema em aberto.

Atualmente os projetos de engenharia trabalham com modelos da ordem de 20 a 350 milhões de triângulos, enquanto as placas gráficas conseguem renderizar, com performance aceitável, apenas alguns poucos milhões de triângulos. Assim, é necessário o uso de técnicas que reduzam a quantidade total de primitivas gráficas usadas para representar uma cena composta por estes modelos. Essas técnicas não podem simplesmente remover detalhes dos

modelos já que muitas vezes, para a Engenharia, eles são o objetivo da visualização. Daí resulta o requisito de que o modelo completo deva estar sempre disponível para visualização.

Modelos massivos de engenharia apresentam uma característica a mais que os torna difíceis de tratar: além de serem muito complexos no que diz respeito à quantidade total de polígonos, ainda são formados por uma grande quantidade de objetos, tipicamente milhões deles. Enviá-los individualmente para a placa gráfica gera sérios problemas de desempenho, já que há um *overhead* associado a cada chamada diferente de desenho na API Gráfica (OpenGL, DirectX, etc). É necessário agrupar partes do modelo em lotes e enviá-los em grupos para a placa gráfica, reduzindo o efeito desse overhead no tempo de renderização da cena.

Finalmente, temos ainda que modelos com toda essa complexidade não podem ser alocados de uma só vez na memória. Não existem nos computadores atuais memória RAM e muito menos memória de vídeo para representá-los. Por isto, a cada quadro da visualização é necessário manter em memória apenas a geometria da parte visível na resolução adequada. Determinar a parte visível consiste em incluir apenas os objetos que interceptam a pirâmide de visão e que não estão oclusos por outros. Definir a resolução adequada trata do fato de que objetos muito distantes da câmera afetam poucos pixels da imagem do quadro e por isto sua representação geométrica e de cor pode ser substituída por outra mais simples praticamente sem perda de informação visual. Apesar destes problemas de descarte, oclusão e multi-resolução terem sido tratado por muitos artigos ainda não temos uma boa solução para o tipo de modelo que estamos tratando. Um dos artigos mais promissores por Gobbetti & Marton (2005) apresenta uma técnica denominada por eles de Voxels Distantes (*Far Voxels*).

Esta dissertação implementa e avalia a técnica de Voxels Distantes aplicada à indústria de petróleo, e em especial às instalações marítimas que consistem principalmente de plataformas de produção e explotação. Nosso objetivo central consiste em visualizar interativamente um modelo CAD completo destas plataformas marítimas em microcomputadores com placas gráficas convencionais.

Além da implementação da técnica de Voxels Distantes, este trabalho também apresenta como contribuição a adaptação e avaliação dessa técnica para modelos CAD de estruturas marítimas. A adaptação teve como foco a melhoria visual, de forma a reduzir artefatos causados por características comuns no tipo de modelo utilizado. A fase de preparação do modelo para a visualização também foi simplificada em relação à proposta original de Gobbetti & Marton (2005), usando uma biblioteca para o traçado de raios que gera a informação de visibilidade necessária para a geração dos vários níveis de detalhe do modelo. A avaliação se deu por meio de testes quantitativos (avaliação de desempenho) e qualitativos (avaliação da qualidade visual) de modelos de projeto de plataformas de petróleo da Petrobras.

Esta dissertação está organizada da seguinte maneira. O capítulo 2 apresenta o estado da arte em visualização de modelos massivos. O capítulo 3 detalha a proposta do algoritmo de Voxels Distantes. O capítulo 4 mostra a implementação e o capítulo 5, os resultados obtidos. Finalmente o capítulo 6 descreve nossas conclusões e sugestões para trabalhos futuros.

2
Técnicas e Trabalhos Relacionados

Um bom renderizador de modelos massivos tem que ser capaz de resolver três pontos: reduzir a complexidade da geometria onde ela não for necessária, não renderizar geometrias que estejam ocultas ou fora da pirâmide de visualização e usar pouco processamento durante a renderização da cena. Usar processamento de CPU demais fará o desempenho do renderizador ficar limitado pela capacidade de processamento da máquina, ao invés de se limitado pela capacidade da placa gráfica. Esta seção explica como as idéias de simplificação de modelos e descarte de objetos ocultos têm evoluído para permitir seu uso com modelos massivos. A explicação parte dos algoritmos clássicos de níveis de detalhe voltados para um único objeto e procura mostrar como eles foram estendidos para permitir o tratamento de grandes conjuntos de objetos. Nesta seção também são discutidas formas alternativas de criar representações simplificadas dos objetos do modelo como impostores ou grades de voxels.

Técnicas de simplificação de malhas já são estudadas há muitos anos. Em especial, algoritmos voltados para reduzir a complexidade de um único objeto extremamente tesselado, como os apresentados na figura 2.1, já estão num estágio bem maduro de desenvolvimento. A primeira publicação a tratar do assunto foi Clark (1976), que explica como um mesmo objeto pode ser representado por diferentes níveis de resolução, criados manualmente, quando estiver distante o suficiente da câmera. Schroeder et al. (1992) explica como vértices de um modelo podem ser eliminados, seguindo um critério de erro geométrico, para criar modelos com menor resolução automaticamente. Hoppe (1996) introduz uma estrutura de Progressive Meshes que armazena os diferentes níveis de resolução da malha como uma seqüência de refinamentos a partir de uma malha inicial simplificada. Essa estrutura também pode ser percorrida no sentido contrário, permitindo que a malha seja ajustada

continuamente para ficar com a resolução adequada. Como essa técnica não permite que a resolução do modelo varie de acordo com a direção de onde ele é visualizado, ela é denominada de *view-independent LOD*.

Figura 2.1 - Exemplos de modelos estudados em algoritmos clássicos de níveis de detalhe

Hoppe (1997) propõe uma nova estrutura hierárquica de *Progressive Meshes* que permite que diferentes regiões de um mesmo objeto sejam refinadas independentemente. Assim, regiões do objeto mais próximas à câmera podem ser mais refinadas, enquanto regiões mais distantes podem ser mantidas em resoluções menores. Uma idéia semelhante foi apresentada independentemente por Xia et al, (1997). Essa técnica é denominada de *view-dependent LOD*.

O primeiro problema a impedir que essas técnicas sejam usadas em modelos massivos é que elas são criadas especificamente para trabalhar em cima de objetos individuais. Usá-las em modelos formados por vários objetos, como ocorre em modelos de engenharia, não traz bons resultados. Alguns objetos podem ter características que possam ser eliminadas sem serem notadas quando o objeto for simplificado isoladamente, mas que criam grandes falhas na visualização quando simplificadas em conjunto com outros objetos. Simplificar objetos individualmente também pode dificultar a simplificação do objeto, fazendo com que ele seja representado com mais faces do que seria necessário caso os outros objetos fossem levados em consideração.

Essa característica é contornada por Luebke & Erikson (1997) que sugere que o modelo como um todo seja processado como se fosse um único objeto. Assim, uma única hierarquia de subdivisões/refinamentos é criada para todo o modelo. Essa estrutura é, segundo o autor, semelhante às *Progressive Meshes*, mas permitindo a operação envolvendo mais de dois vértices a cada passo (o que

permite a obtenção de uma estrutura mais compacta) e, principalmente, permitindo a união de vértices em objetos diferentes, para que as geometrias de múltiplos objetos possam ser combinadas em uma única, que represente o conjunto.

Outra solução encontrada foi permitir que os objetos fossem agrupados apenas quando fossem simplificados. O conceito de hierarquia de níveis de detalhe (HLOD) também foi introduzido por Clark (1979), mas nessa publicação tanto a simplificação quanto os agrupamentos de objetos são gerados manualmente. A idéia de agrupar objetos de forma automática, visando preservar as características visuais do conjunto foi introduzida por Erikson & Manocha (1998), que defende o uso níveis de detalhe estáticos para representar objetos ou grupos de objetos do modelo, já que estes podem ser armazenados em *display lists*, muito mais eficientes para serem usadas com as placas 3D atuais do que as geometrias dinâmicas como as *Progressive Meshes*. Outra grande vantagem do agrupamento de objetos, além de permitir a geração de uma simplificação mais fiel ao modelo original, é reduzir a quantidade total de objetos a serem enviados para a placa gráfica.

Erikson et al. (2001) aplica a técnica de HLOD em um visualizador de modelos massivos para cenas estáticas ou cenas dinâmicas com movimentação pouco freqüente de objetos. O modelo usado na visualização é gerado durante uma etapa de pré-processamento que, após gerar diversos níveis de detalhe para cada objeto existente no modelo, agrupa os níveis menos detalhados destes e continua a simplificação em cima dos grupos criados. Ao final do pré-processamento, teremos uma hierarquia em que a raiz contém um único objeto criado de forma a representar o modelo como um todo quando visto de longe e que se subdivide em grupos de objetos que representam o mesmo modelo de forma mais detalhada. As folhas da árvore montada nessa hierarquia irão conter os objetos originais do modelo.

Um dos defeitos dessa forma de geração de HLOD, como se pode ver nas figuras 2.2 e 2.3, é que, apesar de preservar bem a forma do conjunto de objetos, a cor final obtida na simplificação nem sempre representa bem o conjunto de objetos. Uma boa representação deve ter como cor a média das cores dos objetos

visíveis existentes. Apesar desse estudo levar em consideração a área das faces existentes durante o cálculo da geometria simplificada, ele falha ao não conseguir determinar quais faces estão ocultas para deixar de considerá-las no cálculo da cor final da geometria simplificada.

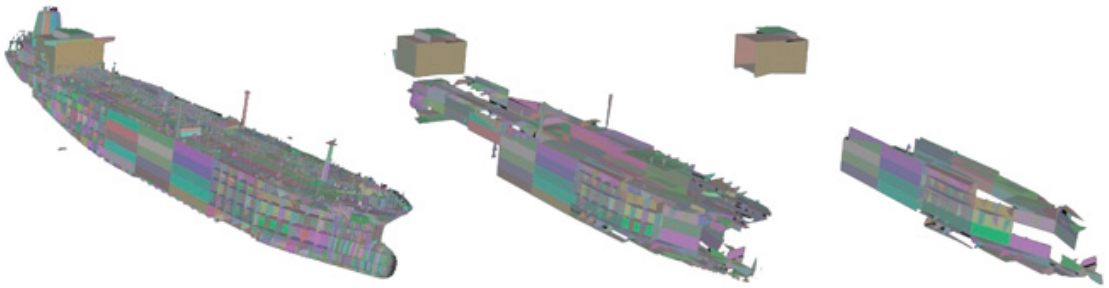

Figura 2.2 - Exemplo de uma técnica tradicional de LOD usada em modelos massivos. Figura extraída da publicação de Erikson et al. (2001)

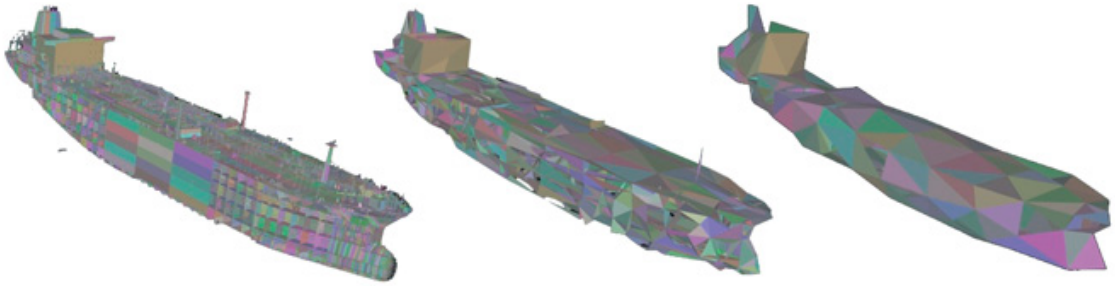

Figura 2.3 - O mesmo modelo simplificado usando HLODs. Figura extraída da publicação de Erikson et al. (2001)

Maciel & Shirley (1995) introduziu o conceito de impostores para substituir objetos ou clusters de objetos. Os impostores são imagens geradas para substituir parte da geometria do modelo. As imagens são geradas para ângulos pré-determinados de cada objeto ou cluster de objetos. Uma vantagem é que, como os impostores são gerados usando o próprio renderizador, superfícies ocultas são escondidas automaticamente, de forma que não afetam a aparência final da representação simplificada.

Rusinkiewicz & Levoy (2000) introduz um algoritmo para visualizar grandes modelos escaneados usando uma hierarquia de nuvens de pontos. Esses modelos são formados por 100 milhões a 1 bilhão de pontos amostrados. Técnicas tradicionais de visualizar esse tipo de modelo envolvem criar uma

triangulação em cima desses pontos amostrados e simplificá-la usando uma técnica de *view-dependend LOD*. Esse novo algoritmo de hierarquias de nuvens de pontos propõe que os pontos amostrados sejam renderizados usando pequenos pontos de pouco mais de um pixel, renderizandos usando uma primitiva de ponto do OpenGL. Quando visualizados a partir de uma certa distância, esses pontos começam a ser agrupados em um único ponto um pouco maior que represente as suas características combinadas. Esses pontos gerados por agrupamento são novamente agrupados para formar pontos maiores, que represente aquela parte do modelo quando for visualizado de longe (Figura 2.4).

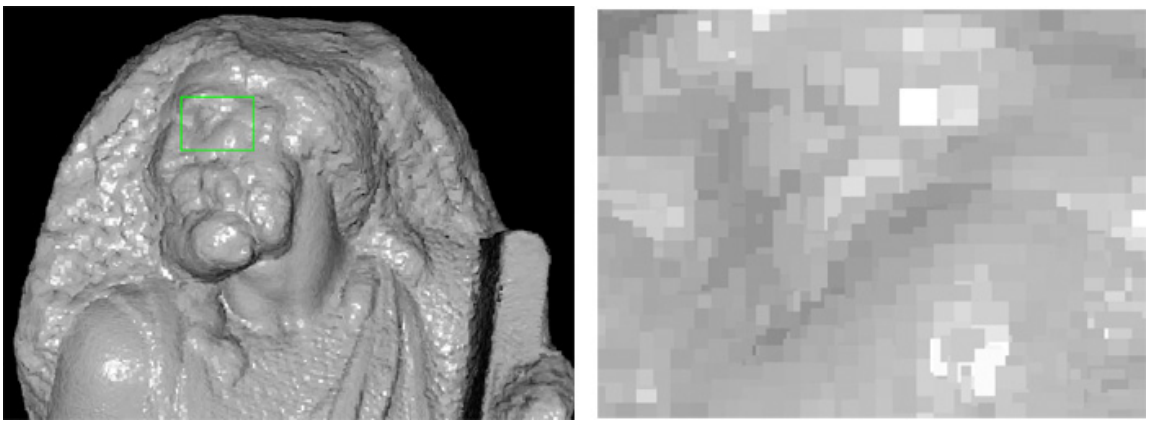

Figura 2.4 - Rusinkiewicz & Levoy (2000) renderizando um modelo escaneado usando uma hierarquia de pontos (esquerda). Quando vistos de perto (direita), podemos ver que os pontos são na verdade primitivas de pontos do OpenGL, que são desenhados como pequenos quadrados. Figura extraída da publicação de Rusinkiewicz & Levoy (2000).

Gobbetti & Marton (2005) apresentam o algoritmo de Voxels Distantes (*far voxels*) que também usa hierarquias de nuvens de pontos, mas voltado para a visualização de modelos CAD, conseguindo um desempenho interativo para modelos com 350 milhões de triângulos (Figura 2.5). Nesse trabalho as nuvens de pontos são chamadas de voxels, e são usadas para representar as versões simplificadas dos objetos da cena numa estrutura de HLODs. Os níveis mais detalhados, correspondentes às folhas da árvore de HLODs, ainda são representados da forma tradicional, usando a geometria original do modelo.

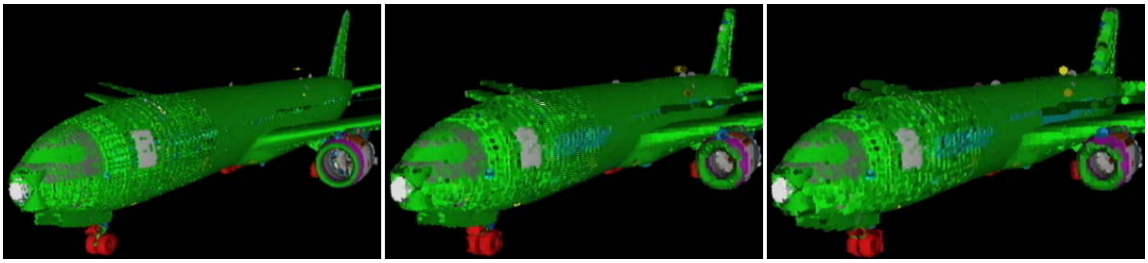

Figura 2.5 – Diferentes níveis de resolução usados para representar um modelo CAD de um Boeing 777 (350 milhões de triângulos) usando a técnica de Voxels Distantes. Figura extraída da publicação de Gobbetti & Marton (2005)

Simplificações baseadas em polígonos permitem um maior desempenho, ao custo de se sacrificar a aparência exata do modelo e qualidade final de imagem. Simplificações baseadas em pontos, por outro lado, permitem um controle maior da aparência de cada região do modelo simplificado, o que melhora a qualidade da imagem ao custo de exigir mais primitivas na representação.

Uma alternativa que recentemente começou a ganhar força para resolver o problema de visualização de modelos massivos é o uso de um algoritmo inteiramente baseado em traçado de raios, sem nenhuma aceleração em placa gráfica. Dietrich et al. (2003) propõe o OpenRT, uma API padronizada para aplicações de traçado de raios interativas. Essa API é orientada para ser simples, fácil de usar e difundida como o OpenGL, mas é estruturada de forma diferente. (Wald et al., 2005) propõe um visualizador que usa essa API para renderizar modelos massivos usando paginação, estratégias de carregamento sob demanda e um esquema de aproximações hierárquicas para representar geometrias que ainda não tenha sido carregadas.

Segundo Yoon et al. (2006), a velocidade atual dos processadores está avançando a ponto de conseguir renderizar modelos a taxas interativas usando traçado de raios e o atual problema a ser resolvido para permitir que essa técnica seja usada para modelos massivos é reduzir o tamanho do *working set* necessário. O *working set* é o conjunto de dados que precisam estar alocados na memória principal para permitir a renderização de um quadro. Nessa publicação, é proposto o uso de R-LODs para substituir grupos de triângulos inteiros nos nós internos da KD-Tree usada. Segundo a publicação, R-LODs são planos que contêm atributos de material, como as cores da geometria substituída. O ganho

de performance obtido, quando comparado a um sistema de traçado de raios sem R-LODs, varia de 2 a 20 vezes, com pouca perda de qualidade visual (Figura 2.6).

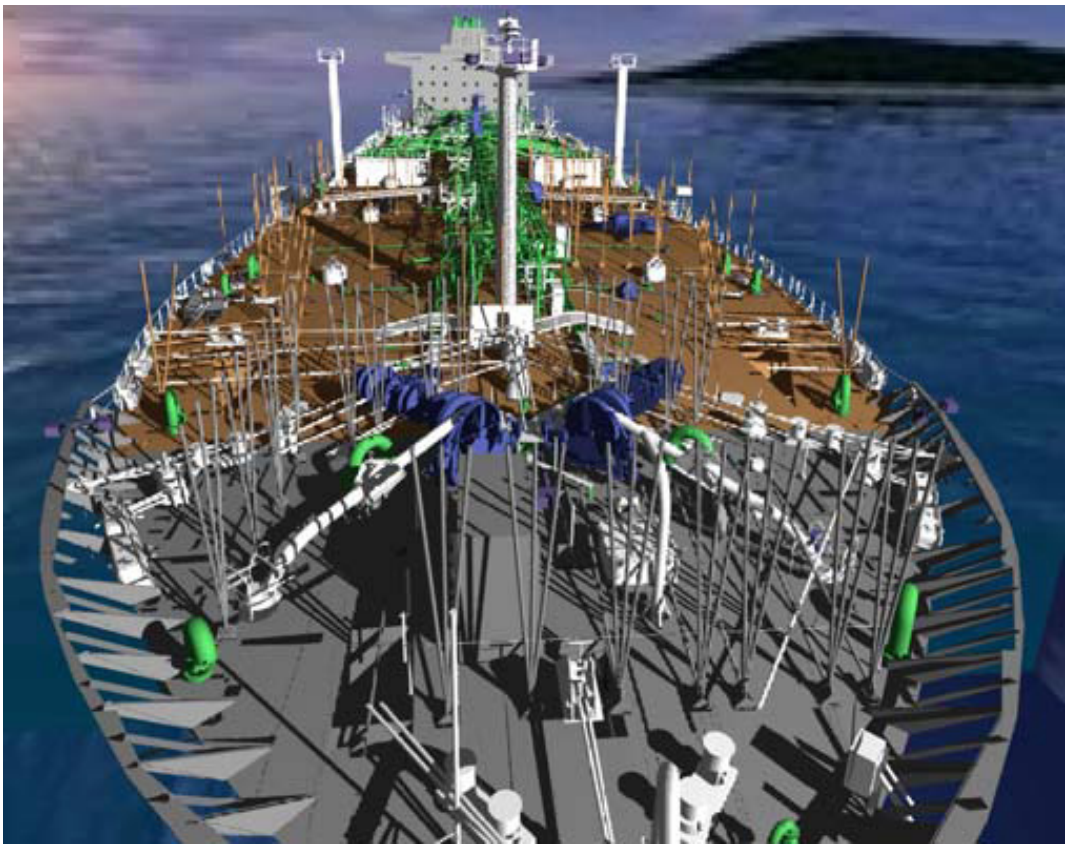

Figura 2.6 – Modelo de um navio com 77 milhões de triângulos renderizado com a técnica de R-LOD apresentada por Yoon et al. (2006)

Soluções baseadas em traçado de raios têm três grandes vantagens sobre outros métodos de renderização. A primeira vantagem é que a visibilidade de superfícies pode ser determinada em tempo logarítmico sobre o tamanho da cena, desde que o modelo seja estruturado de forma adequada. Outra vantagem é que o traçado de raios é altamente paralelizável, bastando que cada pixel a ser renderizado seja atribuído a um processador diferente. E por último, o cálculo de sombras e reflexões pode ser feitos com um custo não muito maior que o da própria renderização usando traçado de raios.

Frisken et al. (2000) sugere a adoção de *Adaptively Sampled Distance Fields*, ou ADF, como uma estrutura de dados fundamental para a Computação Gráfica. Segundo a publicação, sua estrutura é simples e direta e é extremamente efetiva na reconstrução de formas complexas com boa qualidade.

ADF representam superfícies ou volumes através de um campo de valores de distância amostrados adaptativamente em uma estrutura espacial (no caso, uma *octree*). Nesse campo, cada valor representa a distância do ponto em questão até o ponto mais próximo na superfície ou volume sendo representado na ADF. De posse desse campo, é possível usar um algoritmo de traçado de raios para gerar uma representação para a superfície ou volume em questão de forma eficiente.

Até o momento, existem poucas publicações que abordem o uso de ADF para modelos de engenharia. Duguet et al. (2006) apresenta um relatório de pesquisa não publicado, onde são propostas técnicas para se criar uma estrutura de ADF em *octree* para modelos massivos, técnicas de aceleração para a geração da ADF e formas automáticas de preencher buracos existentes nos modelos massivos usados.

Esta dissertação apresenta uma implementação da técnica de Voxels Distantes introduzida por Gobbetti & Marton (2005). Juntamente com o traçado de raios usando impostores apresentada por Yoon et al. (2006) e ADF, a técnica de Voxels Distantes representa uma das idéias mais atuais para tratar a renderização de modelos complexos com taxas interativas.

A escolha por essa técnica foi tomada porque a técnica de traçado de raios não consegue tirar vantagem das placas 3D existentes atualmente e ainda depende do uso de máquinas com diversos processadores para conseguir uma performance interativa. A técnica de ADF, por outro lado, é voltada para modelos com formas orgânicas, tendo dificuldades para lidar com objetos com arestas, que constituem a maioria dos objetos existentes em modelos de engenharia. Além disso, a técnica de ADF ainda não foi aplicada com eficiência em modelos massivos. Assim, acreditamos que a técnica de Voxels Distantes é atualmente a mais apropriada para a visualização dos modelos massivos de engenharia.

3
O Algoritmo de Voxels Distantes

Nesse capítulo explicamos em detalhes o algoritmo de Voxels Distantes apresentado no artigo de Gobbetti & Marton (2005) e implementado nesta dissertação para avaliação em modelos de plataformas marítimas. A seção 3.1 apresenta o conceito de hierarquia de níveis de detalhes (HLOD ou LOD Hierárquico). A seção 3.2 detalha como voxels são usados para gerar representações mais simples do modelo. A seção 3.3 explica como é realizado o pré-processamento que gera os voxels. Um ponto importante nesta seção é o descarte de voxels que representam partes oclusas no modelo. A seção 3.4 explica o algoritmo de visualização que utiliza o carregamento sob demanda da representação hierárquica do modelo armazenada no disco. A seção 3.5 explica o funcionamento de um tipo especial de voxel criado nessa dissertação para representar características encontradas nos modelos testados. Finalmente, a seção 3.6 expõe as limitações da técnica de Voxels Distantes.

3.1
Níveis de detalhe e hierarquia (HLOD)

Para garantir que cada região do modelo seja representada com a resolução ideal para a visualização, o visualizador desenvolvido neste trabalho emprega a técnica de LOD Hierárquico. Essa forma de LOD, como foi explicado anteriormente, além de reduzir a complexidade de cada objeto da cena, cria grupos de objetos próximos, reduzindo o número efetivo de objetos a serem percorridos durante a renderização e criando representações mais apropriadas para o grupo de objetos como um todo. A resolução ideal a ser escolhida para cada parte do modelo durante a visualização é aquela que representa todos os detalhes visíveis daquela parte do modelo com o menor custo possível de renderização.

Para formar a hierarquia de LODs, o modelo é subdividido de forma que suas partes formem uma árvore. Cada nó da árvore contém uma representação simplificada de todos seus filhos e as folhas armazenam a representação original mais detalhada. A raiz da árvore contém a versão menos detalhada do modelo. Esse nível representa o modelo como uma geometria única de forma bem simplificada e é usada quando o modelo estiver sendo visto de bem longe. A Figura 3.1 ilustra esta árvore.

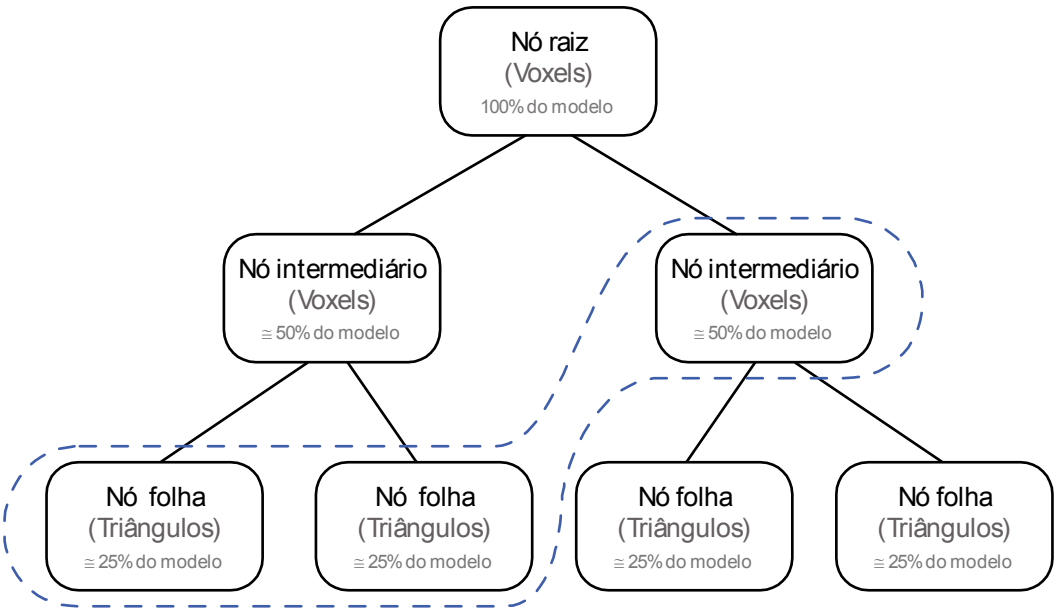

Figura 3.1 - Representação gráfica da hierarquia de níveis de detalhes de um modelo. A linha pontilhada mostra uma possível configuração de nós escolhidos para gerar uma visualização do modelo

Os filhos do nó raiz dividem o modelo em partes menores, cada uma com uma versão um pouco mais detalhada da sua respectiva parte na geometria do pai. Assim, durante a visualização, conforme chegamos perto de determinadas regiões do modelo, devemos selecionar níveis mais baixos (mais detalhados) na hierarquia para representar as partes próximas à câmera.

O uso de uma hierarquia, se comparada a uma simples divisão regular do modelo, permite que tenhamos um maior controle dos níveis de detalhe selecionados nas partes representadas com mais detalhes do modelo. Graças ao uso da hierarquia, estas partes estarão divididas em regiões menores do que as partes menos detalhadas, permitindo um ajuste mais preciso dos níveis de detalhe. Como essas partes mais detalhadas são justamente as que têm um custo maior de renderização, é interessante que elas somente sejam utilizadas quando

um nível de detalhe menos refinado não gerar, com certa tolerância, a mesma informação visual. A deformação da projeção cônica faz com que as regiões do modelo mais distantes da câmera se projetem em áreas menores da tela e assim podem ser representadas por modelos menos detalhados. Na hierarquia, estes são representadas por nós mais próximos da raiz.

Na nossa implementação foi usada uma *octree* modificada para gerar a divisão da cena. Nesta *octree* a geometria de cada nó é subdividida por até 8 filhos. Outra opção, recomendada para esse tipo de divisão, é a KD-Tree, que procura criar uma divisão balanceada do modelo mesmo quando a geometria está distribuída de forma não uniforme. Nos nossos casos de testes, a *octree* modificada gerou resultados suficientemente balanceados, como mostram os histogramas apresentados no Capítulo 5, por isso foi usada no lugar da *KD-Tree*.

Resumindo, neste trabalho, a árvore que compõe a hierarquia do HLOD contém em suas folhas a geometria original do modelo, particionada em conjuntos definidos pelas folhas da *octree*. Os nós intermediários contêm conjuntos de voxels dispostos em uma grade regular. Em cada nó intermediário, cada grade de voxels contém uma representação simplificada da região volumétrica do modelo associada a ela quando esta é vista a partir de uma determinada distância.

3.2. Usando voxels para representar modelos

Esta seção discute a idéia de como voxels podem ser usados para criar representações simplificadas do modelo. A idéia básica é que da mesma forma que uma imagem é usada como um painel (*billboard*) numa cena, uma grade de voxels pode ser usada para melhor representar um modelo geométrico 3D que pode ser visto de diversas direções.

Dada uma caixa envolvente do objeto a ser representado, os voxels são organizados em uma grade regular alinhada com a caixa como ilustra a Figura 3.2. Dentro de cada voxel podem existir zero, um ou mais triângulos representando um ou vários materiais. O que a representação por voxel busca é,

quando este voxel se projeta em um ou poucos pixels o valor da radiância capturada pela câmera sintética do OpenGL seja a mesma deste conjunto de triângulos. Para isto as propriedades ópticas do voxel precisam ser escolhidas de forma correta. Para facilitar esta escolha, idealmente um voxel deve se projetar em um ou poucos pixels.

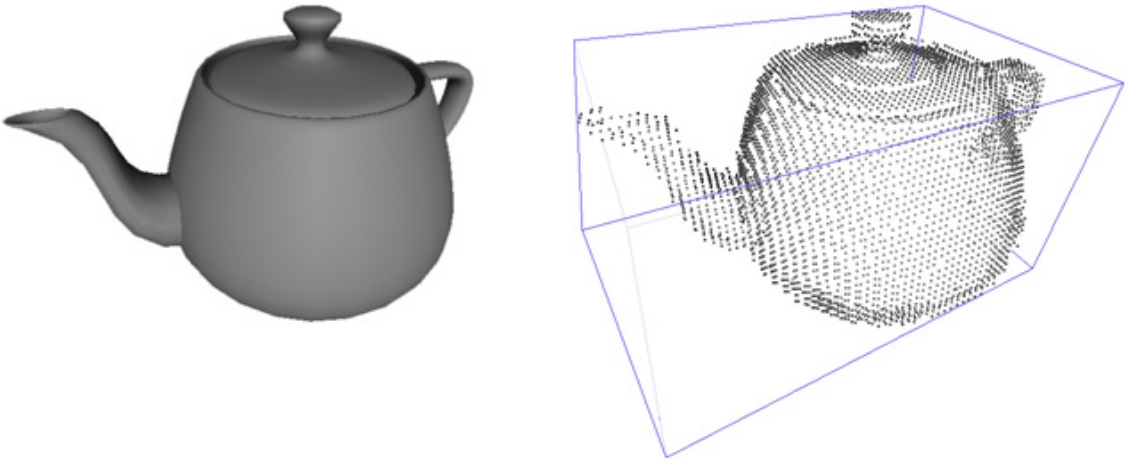

Figura 3.2 - Caixa envolvente e representação de voxel. Os voxels foram representados como pequenos pontos para notarmos sua organização.

O tamanho escolhido para um voxel determina a distância a partir da qual este se projeta em poucos pixels, de forma a conseguir aproximar as características da região associada. Quanto menor o voxel maior a resolução necessária e o custo de renderização da representação volumétrica. Por outro lado, quanto maior o voxel, maior é a distância a partir da qual os voxels formam uma representação adequada. Como em todos os problemas deste tipo, existe sempre um compromisso entre qualidade e desempenho. As Figuras 3.3 e 3.4 mostram dois modelos representados em diferentes níveis de detalhe. Nessa figura, vemos que as representações que usam menos voxels só são suficientes para substituir o modelo quando forem projetadas em áreas muito pequenas da tela.

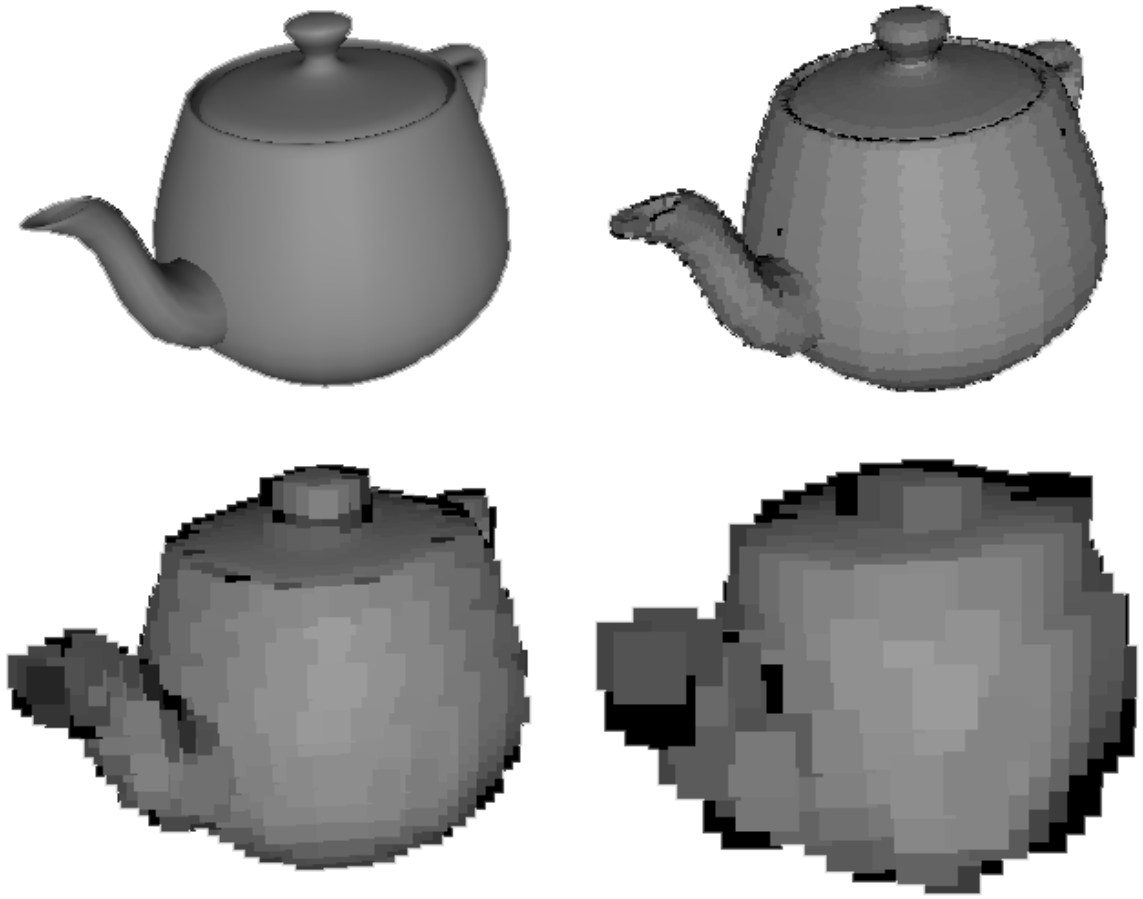

Figura 3.3 - Diferentes níveis de detalhe para representar o modelo do Teapot

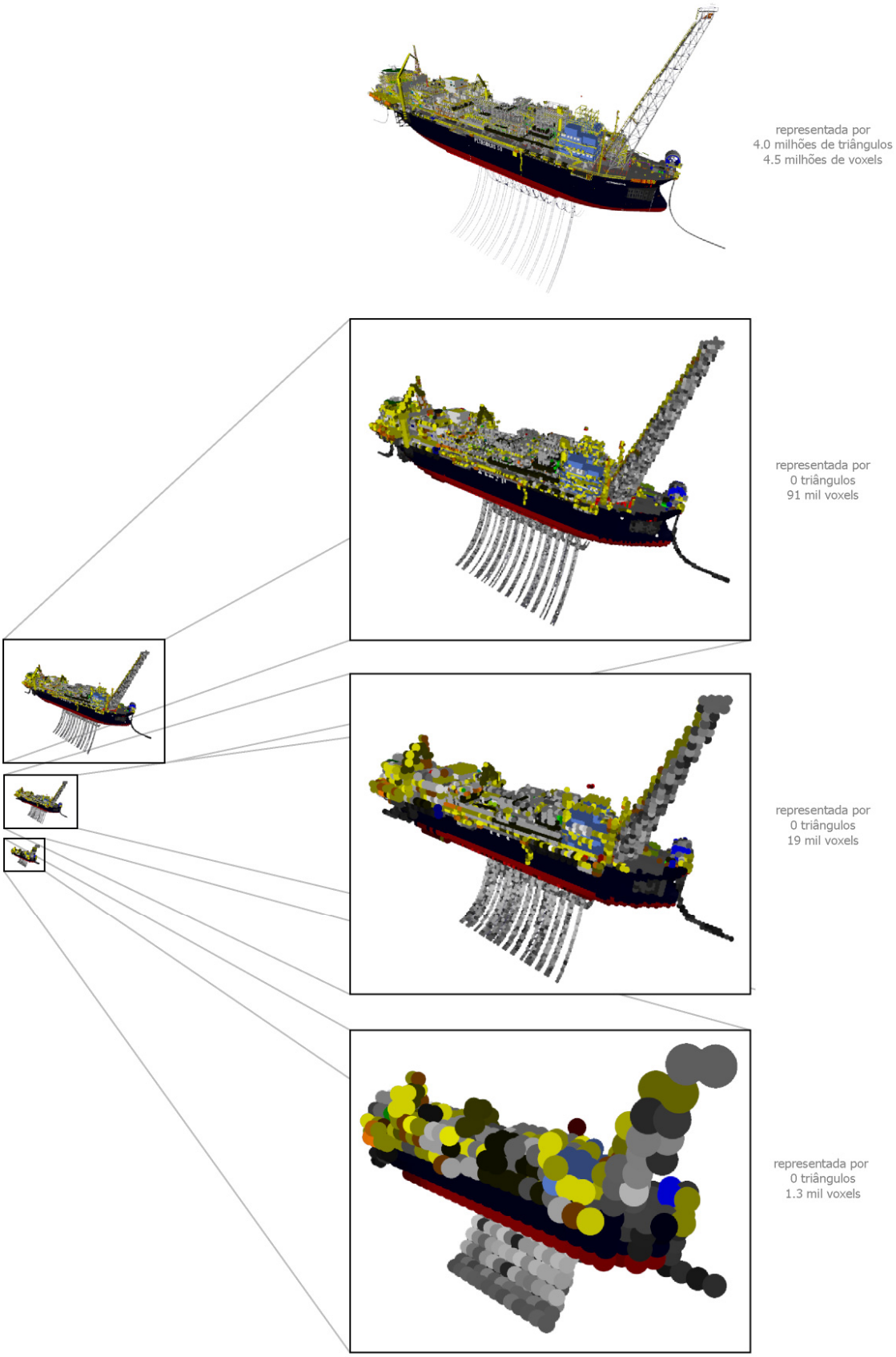

Figura 3.4 - Diferentes níveis de detalhe para representar a plataforma P-50. As imagens da esquerda mostram o tamanho que a representação usando voxels deve ter para não causar perda de qualidade visual. À direita, temos os voxels ampliados.

Para obter uma representação fiel ao modelo original quando visto de longe, cada voxel pode assumir uma dentre diversas representações possíveis. Voxels que correspondam a partes da caixa envolvente que não são ocupadas por nenhuma geometria são marcados como transparentes. Por outro lado, quando existir geometria na região do voxel, não é suficiente atribuir a ele uma cor única. Como a direção de observação não é fixa, é necessário que cada voxel possa assumir cores diferentes de acordo com a direção de onde é visualizado e a configuração das luzes da cena. Aqui, novamente, podemos utilizar modelos mais ou menos sofisticados com diferentes impactos na eficiência da renderização. Podemos, por exemplo, utilizar uma função bidirecional de distribuição de reflectância, BRDF (Dutré et al., 2003), ou podemos utilizar representações menos complexas para atender o espírito da simplificação em prol da eficiência.

Nos casos mais simples, essas diferenças de cor são causadas apenas pela iluminação da cena. Assim, como na representação de modelos CAD trabalhamos com materiais simples, podemos optar por armazenar apenas as cores ambiente e difusa do material e a normal média naquela região para cada voxel. A cor final do voxel pode ser calculada usando as equações tradicionais de cálculo de iluminação difusa e especular como as usadas pelo OpenGL.

Em outros casos, em que a superfície do modelo na região do voxel é formada por superfícies com características diferentes, temos que permitir que este possa assumir representações bem diferentes para cada direção de onde é visualizado, como ilustra a Figura 3.5. Nos objetos (a) e (b) da figura, a normal percebida pelo usuário varia de acordo com a direção de onde o voxel (marcado pela linha pontilhada) é visualizado. No objeto (c), a cor do voxel varia de acordo com a direção de visualização: as setas indicam a cor que o voxel deve assumir em cada direção de visualização.

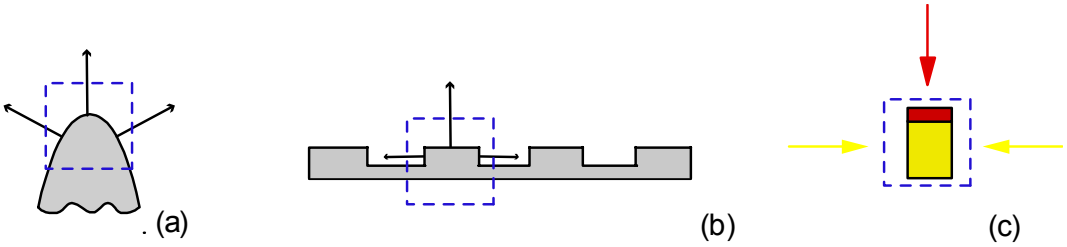

Figura 3.5 - Exemplos de voxels com atributos que variam com a direção de visualização: (a) e (b) ilustram normais variando de acordo com a posição de observação; (c) ilustra a variação de cor.

Para permitir essas diferentes variações, usamos uma representação de voxel que pode apresentar diferentes características de material dependendo da direção de onde é observado. Esse voxel armazena parâmetros independentes de cor, propriedades especulares e normal para cada direção associada às 3 principais direções de visualização (±x, ±y, ±z), totalizando seis propriedades independentes em cada voxel. Quando a câmera está alinhada exatamente com um dos eixos, a cor final resultante será o resultado do cálculo de iluminação usando apenas os parâmetros associados àquela direção. Em direções intermediárias, quando direção da câmera está posicionada entre 2 ou 3 desses eixos, usamos uma interpolação do resultado do cálculo de iluminação de cada uma delas.

Renderizadores clássicos de voxels usam traçados de raios para determinar qual voxel deve ser usado para gerar a representação final de cada pixel da tela. O renderizador proposto nessa dissertação, assim como no artigo sobre Voxels Distantes, trabalha de forma mais simples. Para tirar proveito das placas aceleradoras, cada voxel é representado por um uma primitiva GL_POINT do OpenGL.

Todos os valores necessários para o cálculo da cor final do voxel, como normais e cores de material, são armazenados em vetores de atributos, com uma entrada para cada voxel a ser renderizado. Vetores de atributos funcionam de forma análoga aos vetores de vértices ou normais usados tradicionalmente no OpenGL e foram criados para permitir mais parâmetros associados por vértice sejam enviados para *o vertex shader*.

Em *vertex shaders*, os parâmetros de cada um desses voxels são combinados com a iluminação da cena para gerar a sua cor final. Uma

desvantagem dessa abordagem é que, na maior parte dos casos, vários voxels estarão sendo enviados à placa desnecessariamente, já que não há como determinar que eles estão ocultos. Essa desvantagem é amenizada pelo cálculo da Oclusão Ambiental (*Enviromental Occlusion*), explicado na seção 3.3. Essa forma de descarte permite que voxels que estão sempre ocultos por outras geometrias sejam descartados na visualização.

3.3. Pré-processamento

O objetivo do pré-processamento é transformar o modelo em uma estrutura mais eficiente para ser visualizada. O primeiro passo consiste em dividir as faces do modelo em uma árvore espacialmente hierárquica. Nesta dissertação optamos pela *octree* modificada da forma explicada abaixo. O segundo passo gera representações menos detalhadas que substituam o modelo original nos nós intermediários da hierarquia. Aqui essas representações são formadas por conjuntos de voxels, criados usando um algoritmo de traçado de raios que permite determinar quais faces estarão realmente visíveis durante a visualização e devem contribuir para a cor final de cada voxel. Esse algoritmo também permite que voxels que estejam sempre ocultos sejam descartados. No último passo o resultado do pré-processamento é salvo em um arquivo de forma que o visualizador possa carregá-lo sob demanda do disco. A seção 3.4 detalha o funcionamento desse visualizador.

A geração dos diferentes níveis de detalhe é feita em cima de uma árvore gerada por divisões sucessivas do modelo. Ao final dessa divisão, as faces existentes no modelo estarão separadas em grupos, localizados nas folhas dessa árvore.

A geração da hierarquia de subdivisões começa trabalhando sobre o conjunto total de faces do modelo. A cada passo da subdivisão, devemos decidir em qual dos sub-grupos gerados colocar cada uma das faces existentes. O destino de cada face é decidido de forma independente, sem considerar a subdivisão original em objetos que existia no modelo original. Isso permite que

criemos subdivisões mais eficientes, principalmente em modelos formados por grandes objetos. A desvantagem de ignorar a subdivisão original do modelo é que, durante a visualização, operações que precisem usar a informação de objetos têm que ter uma forma alternativa de reconstruí-los a partir do conjunto de faces gerado.

Em uma subdivisão usando *octree*, as faces existentes são divididas por até 3 planos em até 8 sub-grupos como ilustra a Figura 3.6. Esses planos são sempre alinhados com os eixos ortogonais e passam pelo centro da caixa envolvente da região de interesse. Faces que estejam contidas por inteiro dentro das regiões definidas pelos planos, são atribuídas ao sub-grupo correspondente àquela região. Caso uma face pertença a mais de um sub-grupo, ela tem que ser tratada de forma mais complexa, podendo ser repartida entre os sub-grupos ou replicada em cada um deles. Optamos por adotar nesta dissertação ambas estratégias dependo de uma análise da face.

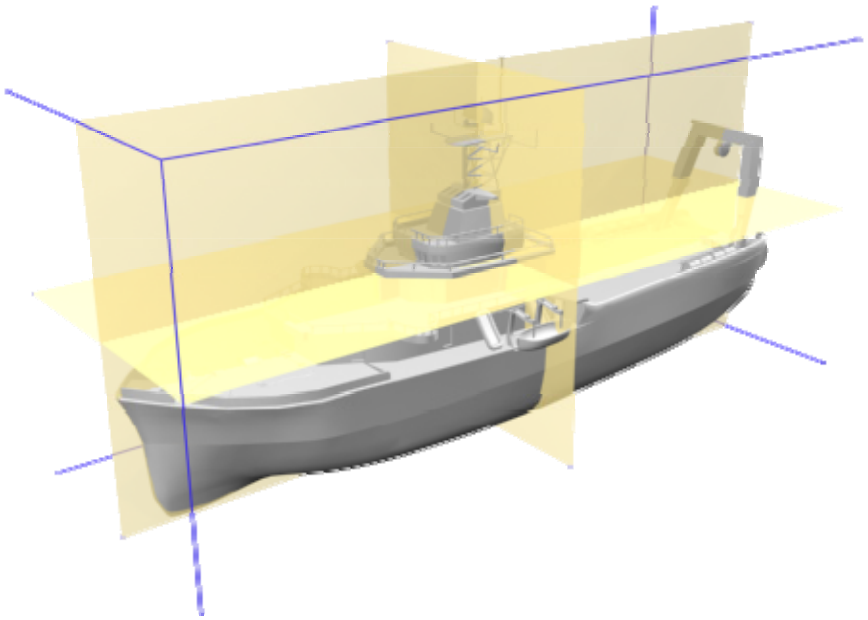

Figura 3.6 - Planos de corte de uma *octree*.

Quando uma face do grupo sendo dividido é cortada por um dos planos de corte, temos que dar um tratamento especial a ela. Quando isso ocorre, isso significa que essa face está ao mesmo tempo em dois ou mais sub-grupos. Colocar a face inteira em apenas um dos sub-grupos não é uma opção viável, já que isso deixaria um buraco na representação do outro. Uma primeira abordagem para resolver esse problema seria quebrar cada face em faces

menores, de forma que as novas faces não fossem mais cortadas pelos planos e pudessem ser divididas entre os sub-grupos sem maiores complicações. O principal problema dessa escolha é que ela gerará, na grande maioria dos casos, três novas faces no lugar da face existente, como mostrado na Figura 3.7. Outra solução possível é simplesmente duplicar a face cortada nos dois grupos. Se pensarmos apenas no número de faces geradas, essa é uma decisão bem mais eficiente, já que geramos apenas duas faces no lugar da face cortada contra 3 do outro método. Essa eficiência só é perdida quando temos faces muito grandes que, quando duplicadas, possam aumentar consideravelmente o custo de rasterização da cena.

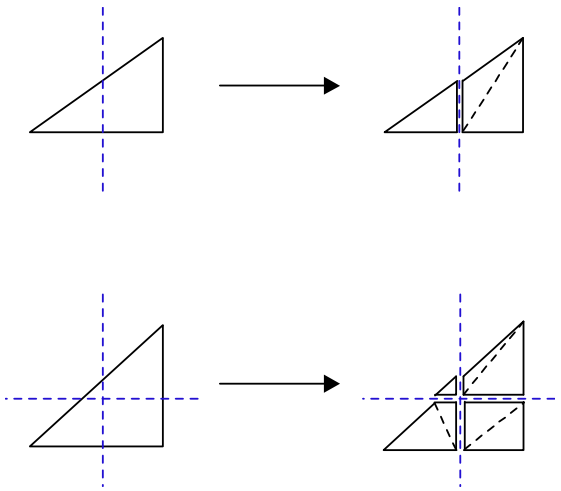

Figura 3.7 - Face cortada pelos planos de corte

Para evitar esses problemas, nosso algoritmo de subdivisão escolhe um desses dois métodos de acordo com as características da face cortada. Definimos uma margem de tolerância, que servirá para decidir quando uma face se estende demais para dentro do subgrupo vizinho. Quando isso ocorre, decidimos cortá-la. Caso contrário, ela é duplicada. Essa margem é definida como sendo uma percentagem fixa do tamanho do subgrupo, como ilustrado na figura 3.8.

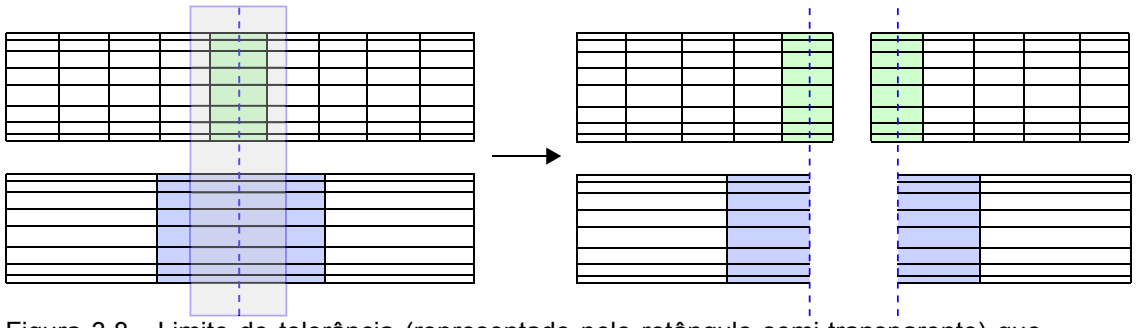

Figura 3.8 - Limite de tolerância (representado pelo retângulo semi-transparente) que determina quando as faces deverão ser duplicadas, caso do objeto de cima ou cortadas, caso do objeto de baixo.

Durante a subdivisão da cena podem surgir nós com caixas envolventes retangulares, em que seus lados tenham dimensões muito diferentes. É possível gerar descendentes com formas mais regulares deixando de dividir a geometria desse nó retangular pelos planos perpendiculares aos lados pequenos do retângulo. A heurística utilizada aqui é que sempre que a medida de um dos lados da caixa envolvente for muito menor que o maior lado, ou seja, quando:

$$L_{menor} < \frac{L_{maior}}{\sqrt{2}}$$

este nó não é subdividido na direção perpendicular a este lado pequeno. Com isto procuramos obter nós com dimensões mais próximas às de um cubo. Essa heurística, criada nessa dissertação, garante que a razão entre os tamanhos da caixa envolvente após a subdivisão será menor do que a razão existente antes.

Os nós são divididos recursivamente até que o nó resultante tenha um número máximo de faces pré-definido. Esse nó será uma folha na representação final da hierarquia de LODs. Para aumentar a eficiência de desenho, toda a geometria da folha é agrupada por materiais para ser toda desenhada em poucas chamadas OpenGL. Tratar a folha como um elemento único também reduz o custo de percurso da nossa estrutura durante o rendering, que seria caro demais se tivéssemos que percorrer e avaliar cada um dos milhões de objetos existentes.

Gerada a subdivisão hierárquica da cena, temos que preencher todos os nós intermediários com representações simplificadas da região que cada um deles ocupa. Como foi explicado anteriormente, essas representações simplificadas usam voxels para recriar as características da geometria existente no modelo. A aparência de cada voxel é gerada a partir de um algoritmo de traçado de raios que coleta amostras da cor dos materiais das superfícies visíveis na geometria do modelo. Nesse algoritmo, raios são traçados a partir de uma grande quantidade de possíveis direções de visualização. Todas as superfícies atingidas por raios são consideradas visíveis e irão contribuir para a cor final do voxel correspondente ao ponto em que houve a interseção. As superfícies que não forem atingidas poderão ser ignoradas. Quando a câmera estiver suficientemente longe, a maior parte das superfícies que formem detalhes

internos no modelo serão ignoradas.

Traçar raios para testar interseções em modelos massivos exige a construção de uma estrutura que permita que esses testes sejam realizados de forma eficiente. Idealmente, queremos que o tempo gasto com cada raio não cresça de forma proporcional ao modelo, o que tornaria lento demais o uso dessa técnica com modelos grandes demais. Nesse trabalho, usamos uma biblioteca de física para acelerar essa etapa, como será explicado no capítulo 4.

O uso do traçado de raios nos permite obter informações sobre a visibilidade das diversas superfícies do modelo. O número de raios traçados, 1 milhão de raios por grupo de voxels, é assumido como sendo suficiente para que as amostras obtidas sejam usadas para determinar a visibilidade dessas superfícies de forma conservativa. Com essa informação é possível evitar que superfícies ocultas contribuam para a renderização da cena causando artefatos visuais. Sem um método eficiente de descarte de superfícies ocultas, partes internas de uma estrutura que tivessem superfícies próximas à superfície externa acabariam influenciando na formação da cor final do voxel, criando uma representação errada. Um exemplo desse tipo de falha é demonstrado na figura 3.9. Na figura, os voxels gerados para o chão da estrutura estão sendo influenciados pela cor amarela da estrutura existente sob o chão, que deveria ter sido ignorada por estar totalmente oculta.

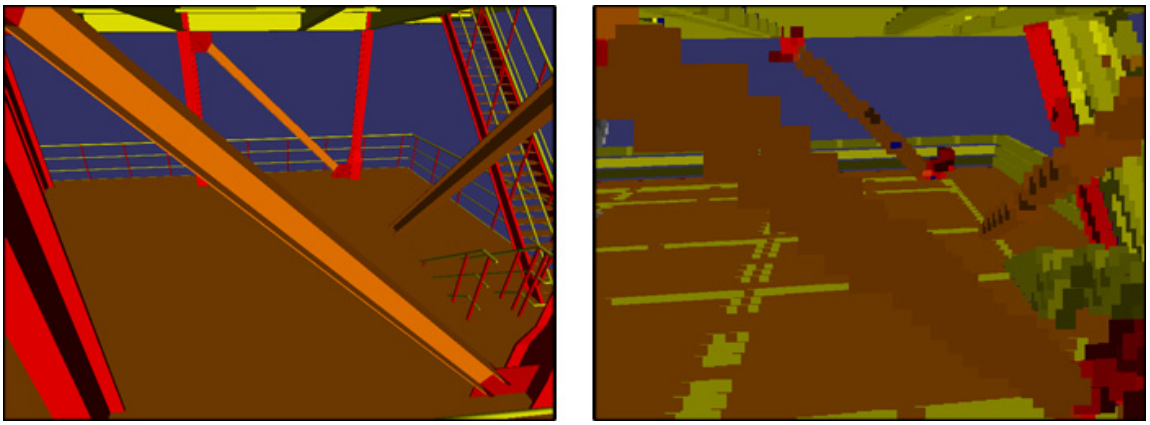

Figura 3.9 – A figura da direita mostra voxels gerados com cores erradas devido a um tratamento incorreto das superfícies ocultas. Na figura, a estrutura em amarelo que estava abaixo sob o chão dessa parte do modelo acabou sendo representada na sua versão simplificada com voxels.

A informação de visibilidade obtida também nos permite esconder voxels internos que fiquem sempre ocultos durante a visualização. Esses voxels

representam uma boa percentagem do modelo, chegando a 57% no modelo da plataforma P-50. Assim, eliminá-los do modelo aumenta consideravelmente a performance da visualização. O descarte de voxels sempre invisíveis é denominado, no artigo sobre Voxels Distantes, de Oclusão Ambiental (*Environmental Occlusion*). Nesse artigo, os voxels removidos por essa operação chegaram a no máximo 43%, no caso do modelo do Boeing 777.

O objetivo do algoritmo de voxelização é calcular uma grade 3D de voxels **V** que represente com a maior fidelidade possível a geometria da região correspondente no modelo. A quantidade de voxels em cada dimensão dessa grade é escolhida de forma que a quantidade total de voxels na grade seja igual a uma constante pré-determinada e que cada voxel corresponda a uma área com forma cúbica.

Os raios do algoritmo são traçados a partir de uma superfície **S**, afastada da grade de voxels **V** de uma distância definida do seguinte modo: Dado que o tamanho de um voxel em unidades do modelo seja **t**, e assumindo que essa grade de voxel será usada para a visualização apenas quando seus voxels tiverem no máximo um pixel de tamanho quando forem projetados na tela. A Figura 3.10 ilustra os elementos contidos em uma configuração típica que pode ser encontrada durante um traçado de raios

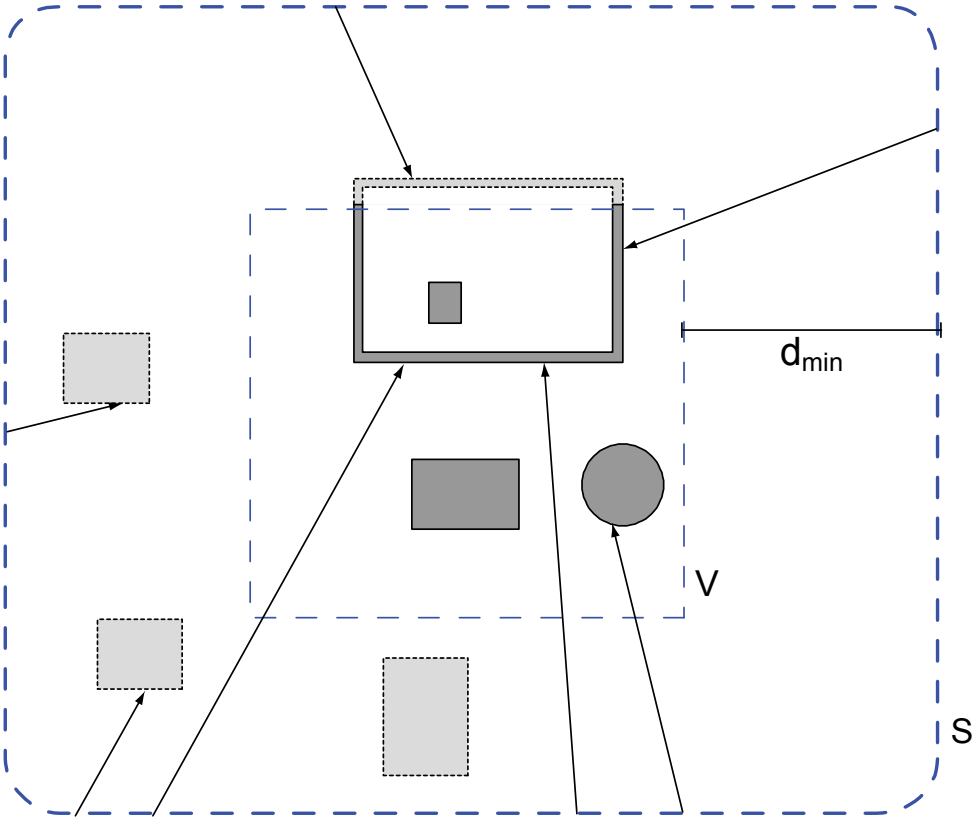

Figura 3.10 – Exemplo de configuração de um traçado de raios típico.

Podemos usar a seguinte fórmula para determinar a distância mínima $\mathbf{d_{min}}$ que a câmera estará dos voxels em **V** durante a visualização:

$$d_{\min} = \frac{t \times res}{2 \times \tan(\frac{fov}{2})}$$

Na equação, $\mathbf{d_{min}}$ representa a distância da superfície **S** à grade de voxels **V**. Essa superfície será a origem dos raios traçados em direção à grade de voxels **V** sendo processada. A variável **t** corresponde ao tamanho de um voxel. As variáveis **res** e **fov** são, respectivamente, a resolução da tela e o ângulo de visão em qualquer direção (horizontal ou vertical). A equação recém apresentada garante que a câmera estará sempre a uma distância maior ou igual a $\mathbf{d_{min}}$ de **V**. Assim, podemos usar todos os objetos localizados na região entre **S** e **V** como oclusores para tentar reduzir a quantidade de superfícies visíveis em **S** e, por conseqüência, reduzir a quantidade de primitivas usadas para desenhar **S** durante a visualização. Apesar de depender diretamente do ângulo de abertura da câmera (**fov**) e da resolução da tela (**res**), a escolha de $\mathbf{d_{min}}$ não obriga esses parâmetros

a serem mantidos fixos. O que deve ser preservado para que não sejam causados defeitos visuais é a relação entre **res** e **fov**.

Cada raio é traçado a partir de posições aleatórias na superfície de **S** em direção a pontos aleatórios localizado dentro do volume de **V**. Caso haja uma interseção fora de **V**, esse resultado é descartado, já que indica que nenhuma superfície dentro de **V** é visível a partir da direção do raio traçado.

Cada interseção que ocorrer dentro de **V** deve ser armazenada. Após o traçado de raios, todas as informações recolhidas em todas as interseções serão agrupadas para formar os voxels de **V**. A posição de cada interseção pode ser convertida em um índice, de forma a enumerar todos os voxels dentro de **V**. Esse índice é usado para indexar a estrutura responsável por armazenar, para cada voxel em **V**, as informações de material e normal no ponto onde houve a interseção.

Terminado o traçado de raios para a grade **V**, temos que reunir toda a informação coletada pelo traçado de raios e gerar os voxels. Cada posição correspondente a um voxel na estrutura poderá ter zero, um ou vários resultados de interseções armazenadas. Caso nenhuma interseção seja atribuída a um determinado voxel, podemos assumir que este não contém nenhuma geometria ou que qualquer geometria contida nele nunca é visível quando a câmera está fora da superfície envolvente **S**. Esse voxel não precisará ser representado durante a visualização.

Caso haja apenas um resultado de interseção naquele voxel, a informação de material e normal desse único resultado será usada para definir um voxel com apenas uma cor difusa e uma normal, iguais às obtidas na interseção.

Quando houver mais de um resultado, temos que combiná-los de forma a criar um resultado coerente. Caso todos os resultados de interseções apresentem normais com direções suficientemente próximas, podemos representá-los com um voxel simples, com apenas uma normal correspondente à média das normais de todas as amostras. Se ocorrer de algumas normais terem direções opostas às outras, ainda podemos usar esse tipo mais simples de voxel, já que ele pode armazenar duas cores independentes, para a direção frontal e traseira do voxel. Essas duas cores, que funcionam de forma análoga à FRONT_FACE e à

BACK_FACE do OpenGL, são usadas respectivamente quando a câmera está posicionada na direção da normal do voxel ou contra ela. Essa situação é encontrada no modelo, por exemplo, em paredes muito finas, que possam ser visualizadas a partir de ambos os lados.

Se em um voxel tivermos materiais diferentes mas com normais iguais, basta fazermos a média das cores e dos outros parâmetros de cada um desses. O resultado será coerente já que, caso essa geometria com múltiplos materiais fosse visualizada ocupando uma área de apenas um pixel, o resultado desejado também seria uma mistura das cores dos materiais existentes.

Caso os resultados das interseções em um voxel apresentem normais muito diferentes, será necessário usar o tipo mais complexo de voxel apresentado na seção 3.2, capaz de representar materiais diferentes para cada direção a partir de onde é visualizado. Para cada uma das seis direções principais de visualização (±x, ±y, ±z), fazemos a média das normais e das cores de todas as amostras em que suas normais apontem para aquela direção. O resultado será um voxel com seis representações independentes, escolhidas de acordo com a direção de onde ele é visualizado.

A cor final do voxel é calculada pela soma das cores das três direções de referência mais próximas à direção **v** a partir de onde o voxel é visualizado pesadas pelos co-senos diretores associados aos respectivos eixos. O co-seno diretor de um determinado eixo corresponde ao comprimento do vetor **v** normalizado projetado naquele eixo.

Esse procedimento de geração de voxels é repetido até que tenha sido gerado uma grade de voxels para cada nó intermediário criado na etapa de subdivisão hierárquica do modelo.

O resultado de todo o processamento (a hierarquia gerada, a geometria particionada e as grades de voxels) é salvo em um arquivo binário de forma que possam ser acessados sob demanda durante a visualização.

3.4. Visualização

O trabalho do visualizador pode ser resumido em selecionar a representação mais simples possível para cada parte do modelo de forma que ela represente todos os detalhes visíveis para cada posição da câmera assumida durante a navegação do usuário. Ele faz isso escolhendo os níveis de resolução apropriados na estrutura de LOD Hierárquico de acordo com a distância que cada um dos nós esteja da câmera. Essa decisão é baseada num fator definido pelo usuário que indica o tamanho aceitável que um voxel pode ter na tela. Escolhidos os níveis de LOD a serem usados, o visualizador verifica quais já estão disponíveis em memória e os renderiza. Os que não estiverem disponíveis, são agendados para serem carregados em segundo plano.

A visibilidade de cada nó renderizado é testada usando-se testes de oclusão em hardware. O algoritmo de emissão e recolhimento das consultas é baseado no algoritmo apresentado no trabalho de Bittner (2004) e tem como objetivo não deixar que a CPU fique esperando sem ter o que processar enquanto espera pelo resultado das consultas realizadas.

O arquivo binário que armazena a estrutura gerada durante o pré-processamento é dividido em duas partes. A primeira parte contém as informações que definem a hierarquia da cena, e fica sempre carregada por completo na memória. A segunda parte contém os dados dos voxels e das geometrias, necessários para se renderizar a cena. Essa parte é carregada sob demanda, dependendo da região da cena que estiver sendo visualizada.

A primeira parte do arquivo contém informações essenciais para o percurso da hierarquia como as caixas envolventes dos nós, o tamanho dos voxels usados para definir a representação simplificada em cada nó e a posição no arquivo em que se encontram os dados a serem carregados para desenhar aquele nó.

A renderização de um quadro começa na raiz da árvore de LODs gerada durante o pré-processamento e segue o algoritmo apresentado abaixo.

```
função percorrer( Nó nó )
  se estaNoFrustum( nó ) == falso então
     retornar
  // Caso esse seja um nó folha, renderizar a geometria
  // existente nele e retornar
  se nó.éFolha então
     renderizar( nó.reprEmTriângulos )
     retornar
  // Calcular o tamanho do nó quando projetado na tela
  tamProj = calcularTamanhoProjetado( nó )
  se tamProj <= nó.reprEmVoxels.tamanhoMax então
     // O tamanho da representação em voxels
     // desse nó é apropriado para ser usado
    renderizar( nó.reprEmVoxels )
  senão
     // O tamanho da representação em voxels não tem
     // resolução suficiente, temos que continuar
    // percorrendo a hierarquia e selecionar um dos
    // filhos desse nó.
    se todosOsFilhosEstaoCarregados( nó ) então
       // Continuar o percurso pelos filhos
       // do nó, começando pelos mais próximos
       // à câmera.
       ordenarPorProximidade( nó.filhos )
       para cada filho em nó.filhos faça
          percorrer( filho )
    senão
       // Nem todos os filhos desse nó estão
       // carregados, renderizar a representação
       // desse nó e pedir o carregamento deles
       renderizar( nó.reprEmVoxels )
       para cada filho em nó.filhos faça
          agendarCarregamento( filho )
```

```
função renderizar( Representação repr )
  se testeDeOclusãoHabilitado() então
     repr.idTeste = iniciarTesteDeOclusão()
     se repr.visívelNoUltimoQuadro então
        // Se no quadro anterior o nó estava visível,
        // ele é renderizado de imediato
        repr.renderizarComOpenGL()
        repr.foiRenderizada = verdadeiro
     senão
        // Caso contrário, devemos primeiro
        // renderizar a caixa envolvente do nó e
        // caso esta esteja visível, renderizar o
        // nó
        renderizarCaixa( repr )
        repr.foiRenderizada = falso
     terminarTesteDeOclusão()

     // Todos os testes realizados são armazenados
     // em uma lista para que possam ser consultados
     // após o percurso de toda a hierarquia
     testesDeOclusãoEnviados.adicionar( repr )
  senão
     repr.renderizarComOpenGL()
```

```
// Essa função é responsável por recolher todos os testes
// de oclusão realizados e é chamada logo após o
// percurso da cena.
função coletarTestesDeOclusão()
  para cada consulta em testesDeOclusãoEnviados faça
     visível = pegarResultado( consulta.idTeste )
     repr.visívelNoUltimoQuadro = visível
```

```
// Caso a caixa envolvente do nó esteja visível
// e este ainda não tenha sido renderizado,
// devemos fazê-lo
se visível e repr.foiRenderizada = falso então
   repr.renderizarComOpenGL()
```

Para cada nó percorrido, o visualizador deve decidir se a resolução do nível de detalhe existente nele é apropriada para ser usada naquele momento ou se é necessário continuar descendo na hierarquia para escolher nós mais detalhados. Essa decisão é tomada usando um fator escolhido pelo usuário que define o tamanho máximo que um voxel pode ter quando ele é projetado na tela. A melhor visualização é obtida quando esse fator é 1, indicando que cada voxel deve corresponder a aproximadamente um pixel na tela. Valores maiores podem ser escolhidos para relaxar a visualização, permitindo o uso de voxels maiores. Com voxels maiores, menos voxels têm que ser usados para renderizar o modelo, aumentando o desempenho da visualização em troca de perda de qualidade visual.

O visualizador também realiza para cada nó percorrido o teste tradicional de descarte de objetos fora da pirâmide de visão, descartando nós que não contribuam para a imagem final.

Caso um nó tenha que ser refinado, o percurso prossegue pelos filhos daquele nó. Os testes de oclusão em hardware (*Occlusion Queries*) realizados exigem que os nós mais próximos à câmera sejam renderizados antes dos nós mais distantes, para que possam servir de oclusores durante os testes. Para isso, basta percorrer os filhos de cada nó por ordem de proximidade à câmera.

Ao escolher um determinado nó para ser renderizado, ele deve ser enviado para a placa. Caso esse nó tenha sido determinado como visível no quadro anterior, o mesmo é renderizado normalmente, realizando um teste de oclusão para determinar se ele está visível. Caso ele tenha sido determinado como invisível no quadro anterior nós renderizamos, num primeiro momento, apenas sua caixa envolvente realizando teste de oclusão e desabilitando as devidas máscaras de desenho para que essa renderização não altere o framebuffer. Após

o percurso de toda a hierarquia de cena, todas as consultas realizadas são recolhidas e todos os nós que ainda não tiverem sido renderizados e estiverem visíveis são renderizados.

Após todo o modelo ter sido percorrido e todos os testes de oclusão terem sido realizados, a aplicação realiza uma nova passada recolhendo os resultados e renderizando os nós que tenham sido determinados como visíveis mas ainda não tenham sido renderizados.

Os nós intermediários, formados por voxels, são enviados para a placa 3D em uma única chamada de glDrawArrays() usando primitivas GL_POINT do OpenGL para cada tipo de voxel existente naquele nó. O *vertex shader* apropriado para calcular a aparência desejada dos voxels é carregado antes de cada chamada de glDrawArrays(). Como foi explicado na seção 3.1, existem dois tipos de *vertex shaders*: um simples, para superfícies planas e um mais complexo, que permite que um mesmo voxel tenha materiais distintos de acordo com a direção de onde é visualizado.

Os nós de geometria são organizados em estruturas tradicionais de vetores de vértices desenhados por chamadas de glDrawElements() que usam como primitivas triângulos ou *strips* de triângulos. Para reduzir o número de chamadas do OpenGL, todos os triângulos existentes em um nó de geometria são agrupados de acordo com seus materiais, sem levar em consideração as antigas divisões em elementos/objetos.

Freqüentemente, os dados de voxel ou geometria não estarão disponíveis em memória quando for necessário enviá-los à placa. Nesse caso, um pedido de carregamento desses dados é adicionado a uma lista ordenada por prioridades que será consultada em outra *thread* pelo módulo de carregamento para escolher a ordem em que os nós serão carregados do disco. A prioridade atribuída a um nó é definida de acordo com o tamanho dos voxels existentes no pai daquele nó quando projetados na tela. Nós que tenham pais com voxels maiores quando projetados são carregados primeiro.

À primeira vista, usar o tamanho projetado do voxel do pai de um nó pode parecer não fazer sentido. Essa idéia fica mais clara quando notamos que, caso um determinado nó não esteja disponível em memória, teremos que continuar

usando a representação do seu pai até que ele tenha sido carregado. Durante esse espaço de tempo, o pai estará sendo desenhado com voxels que, quando projetados na tela, estarão maiores do que foi definido pelo usuário como tamanho máximo aceitável para um voxel projetado. Assim, já que os nós com maiores voxels projetados na tela são os que causarão mais artefatos na visualização, esses nós deverão ser os primeiros a ter seus filhos carregados em memória.

O envio de primitivas para a placa a todo frame deve ser evitado ao máximo, mesmo nas placas mais recentes, com portas com altas taxas de transferência de dados. Isso pode ser conseguido com o uso de funções que permitam que essas primitivas fiquem armazenadas na memória de vídeo para serem usadas novamente nos quadros seguintes, como *Display Lists* ou *Vertex Buffer Objects* (VBOs).

As *display lists* apresentam duas desvantagens que as tornam inviáveis para serem usadas no visualizador implementado. Elas gastam muito processamento para serem criadas, o que tornaria lento o processo de carregar novos nós do disco em segundo plano durante a visualização. Outra desvantagem é que elas consomem muita memória. Para permitir que o *driver* gerencie o uso de memória da placa 3D, este tem que manter uma cópia adicional dos dados enviados na memória da CPU.

VBOs, por outro lado, não exigem tanto processamento para serem criados e não precisam ser duplicados em memória. Isso os torna bem mais vantajosos que *display lists* neste visualizador.

Durante a visualização de um nó, o visualizador tenta, sempre que for possível, usar VBOs para evitar transferências desnecessárias para a placa 3D. Caso o nó a ser renderizado já tenha sido armazenado em um VBO, ele é usado para acelerar a sua renderização. Caso contrário, é iniciado o processo de criação de um novo VBO. Primeiro, é alocada a quantidade de memória necessária em memória de vídeo. Caso essa memória esteja disponível e a alocação seja realizada com sucesso, o visualizador envia os dados usando VBOs. Caso ela não esteja disponível, esses dados são enviados da forma tradicional, sem *display lists* ou VBOs até que a memória necessária para alocar

o VBO seja liberada. Essa memória se tornará disponível quando outros VBOs forem desalocados pelo visualizador, sempre que não forem usados durante a renderização de um frame.

O módulo de carregamento de dados em segundo plano consulta continuamente a lista de pedidos ordenada por prioridades. Assim que estiver disponível, o nó de maior prioridade é retirado da lista e é feita uma chamada ao módulo de leitura/escrita, que deverá ler os dados contidos na posição do arquivo correspondente ao nó desejado e convertê-los em um nó válido para ser inserido no grafo. Esse nó será guardado para ser inserido no grafo de cena após o fim do quadro atual, de forma que o grafo não seja modificado enquanto estiver sendo renderizado.

3.5. Voxels com filtro de anti-serrilhamento

Um problema importante encontrado durante a visualização dos modelos utilizados neste trabalho, quando simplificados com voxels, é que nessa representação, sempre que um objeto muito pequeno é representado por um voxel ele assume o tamanho deste. Em casos em que existem vários objetos pequenos muito próximos (como os corrimões finos da figura 3.11 e 3.12), essa falha se torna bem visível. Na prática, é como se os objetos “inchassem”. O maior desconforto não é causado por termos uma representação diferente da representação usando geometria, mas pela transição brusca entre níveis de detalhe com aparências diferentes que ocorrerá durante a navegação.

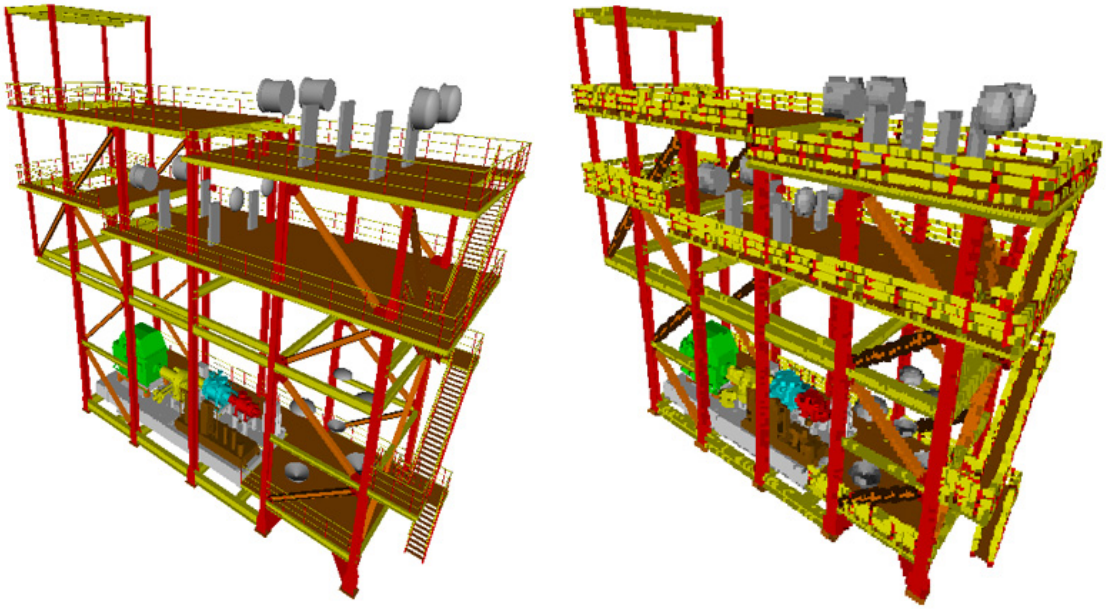

Figura 3.11 – Falhas visuais geradas por objetos muito finos sendo representados por voxels. Na figura da esquerda, representada totalmente por geometria, podemos ver que os corrimões são bem mais finos que o tamanho escolhido para os voxels, na figura da direita. Nessa figura, os voxels foram ampliados para ilustrar a falha.

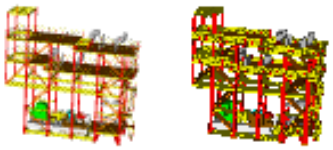

Figura 3.12 – As mesmas falhas da figura 3.11. Mesmo representadas sem ampliação, a diferença entre a representação usando geometria (esquerda) e usando voxels (direita) ainda é visível.

Esse tipo de objeto está presente em boa parte dos modelos usados em nossos testes e, além de distorcer a aparência da simplificação, causam "pulos" durante a transição entre diferentes níveis de voxels ou destes para geometria. Esse problema exigiu uma adaptação do algoritmo de Voxels Distantes para tratar os modelos de estruturas marítimas analisados neste trabalho.

Uma primeira solução óbvia seria quebrar o voxel em voxels menores, de forma a conseguir aproximar melhor a forma do objeto. Além de aumentar a quantidade de primitivas necessárias para renderizar a cena, criar mais voxels vai de encontro à idéia dos Voxels Distantes, que é ter voxels com tamanhos próximos ao tamanho de um pixel na tela.

Uma solução mais interessante encontrada foi tentar simular o mesmo

resultado obtido com o uso do anti-serrilhamento presente nas placas 3D atuais. Quando um objeto é renderizado com anti-serrilhamento, todos os pixels que estejam apenas parcialmente ocupados pela geometria do objeto são desenhados com um fator de transparência aplicado. Esse fator é proporcional à área do pixel que é ocupada pelo objeto e o resultado visual obtido é que as bordas dos objetos ficam mais suaves e, no caso de objetos pequenos, estes passam a contribuir menos para a imagem final, principalmente quando são menores que um pixel.

No caso de voxels, temos que diminuir a contribuição que um voxel tem na cena aplicando um fator de transparência semelhante ao fator aplicado no anti-serrilhamento em placa. Não basta apenas habilitar o anti-serrilhamento em placa, já que, quando gerados sem esse cuidado, o voxel já estará gerando uma representação maior que a esperada para qualquer objeto pequeno contido dentro dele.

Idealmente, teríamos fatores de transparências diferentes para cada possível direção de visualização, proporcionais à área ocupada pelo objeto dentro do voxel quando vistos daquela direção. Como isso ocuparia memória demais durante a visualização, temos que adotar uma simplificação dentre duas possíveis. A primeira seria escolher uma transparência única para todo o voxel, para ser usada quando este é visto de qualquer direção. Essa representação é apropriada quando a transparência varia pouco nas várias direções. A segunda simplificação segue a linha dos voxels que podem assumir diferentes representações apresentados na seção 3.2. Além de atribuirmos cores diferentes para o voxel, podemos atribuir transparências diferentes para cada uma das seis principais direções de visualização (±x, ±y, ±z) a partir de onde esses voxels são visualizados. De forma semelhante ao cálculo da cor final desses voxels, a transparência usada para representar o voxel será a combinação das transparências associadas aos três eixos mais próximos da direção de visualização ponderada de acordo com o quanto essa direção de visualização está próxima de cada um desses eixos. O modelo renderizado com voxels transparentes pode ser comparado com o modelo original, renderizado apenas com geometria e com os voxels sem transparência nas Figuras 3.13 e 3.14.

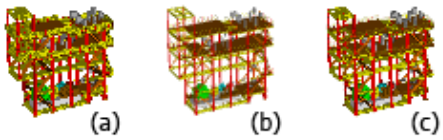

Figura 3.13 – Filtro de anti-serrilhamento no modelo SKID_ABC. Modelo representado por voxels sem nenhum filtro (a); Modelo representado apenas por geometria (b) e modelo representado por voxels com o filtro anti-serrilhamento (c)

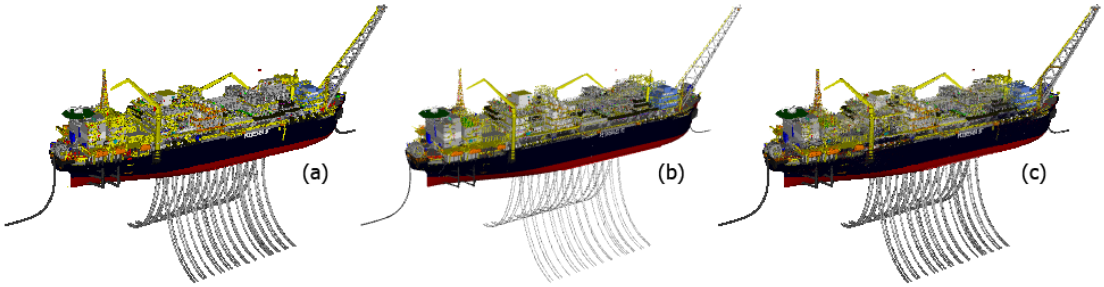

Figura 3.14 – Filtro de anti-serrilhamento no modelo P-50: Modelo representado por voxels sem nenhum filtro (a); Modelo representado apenas por geometria (b) e modelo representado por voxels com o filtro anti-serrilhamento (c)

Para determinar quais voxels devem ser representados como transparentes, podemos usar, durante o algoritmo de traçado de raios, a informação sobre quantos raios passaram através de cada voxel sem atingir nenhuma geometria. Quando um voxel é transpassado por muitos raios traçados a partir de uma determinada direção, podemos assumir com segurança que a geometria contida naquele voxel ocupa apenas uma pequena área do voxel, quando este é visto a partir daquele ângulo. Esse voxel poderá ser renderizado, quando visto dessa direção, com um fator de transparência proporcional à razão entre a quantidade de raios que transpassou o voxel e a quantidade de raios que atingiu alguma superfície dentro do voxel. O modelo da figura 3.15 foi gerado com uma codificação de cores que permite que vejamos como essa razão se distribui pelo modelo.

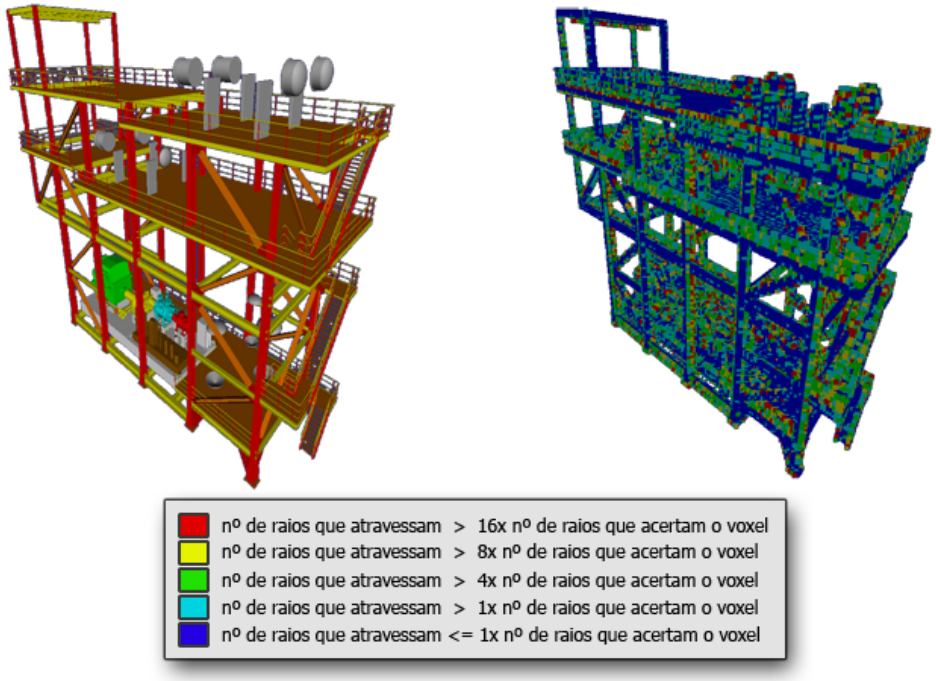

Figura 3.15 – Voxels que são atravessados por muitos raios indicam a existência de objetos que ocupam apenas uma pequena parte do volume do voxel. O modelo da direita foi codificado de forma a destacar os voxels que foram transpassados por muitos raios durante o pré-processamento.

É importante não generalizar a informação de transparência e tornar um voxel totalmente transparente quando esse for transpassado por raios vindos de apenas algumas direções. Em diversos casos, objetos grandes podem ocupar apenas uma parte do voxel (Figura 3.16) de forma que sejam atingidos por raios vindos de apenas algumas direções. Esse voxel pode ser renderizado com transparência quando visualizado a partir das direções de onde os raios que transpassaram o voxel foram gerados, mas deve permanecer totalmente opaco quando visto das outras direções para evitar que esses objetos se tornem transparentes durante a visualização.

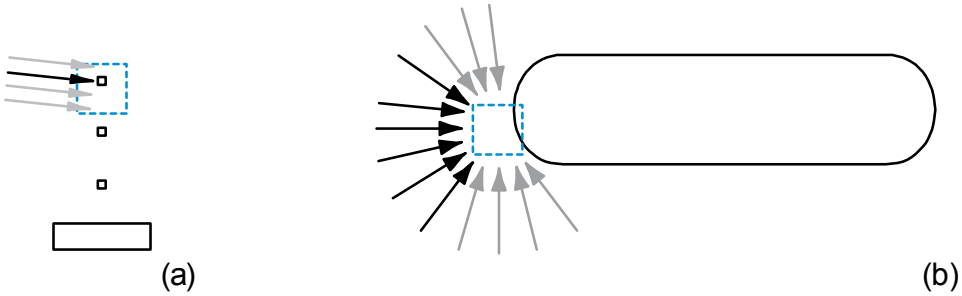

Figura 3.16 – Casos comuns de voxels transparentes encontrados durante o traçado de raios. No caso (a), o voxel poderá ser representado por um único fator de transparência, já que sua visibilidade é semelhante a partir de todas as direções. No caso (b), aplicar transparência nas direções marcadas em preto permitiria ver através do objeto.

A informação de transparência obtida para cada direção de visualização durante o traçado de raios deve ser condensada em uma estrutura mais compacta para ser usada durante a visualização. Caso as direções dos raios que transpassaram o voxel não estejam distribuídas de forma uniforme, devemos usar a representação que permite transparências independentes para cada uma das seis principais direções de visualização (±x, ±y, ±z) explicada anteriormente. Caso as direções apresentadas pelos raios se distribuam de forma uniforme, podemos codificar essa transparência como um valor único para o voxel e usar uma representação mais simples para ele.

Um problema em se usar qualquer geometria transparente em renderizações OpenGL é que temos que desenhá-las após toda a geometria opaca e ordená-las de trás para frente para que todas contribuam de forma correta para a formação da cena.

Desenhá-las depois dos objetos opacos é simples, mas gera algumas complicações quando esse tipo de voxel é usado em conjunto com os Testes de Oclusão apresentados na seção 3.4, obrigando o desenho desses voxels a ser realizado após todo o ciclo de realização dos testes de oclusão e seu recolhimento. Poderíamos fazer um novo ciclo de testes de oclusão para esses voxels transparentes, mas isso se mostrou desnecessário. Esses voxels são pouco numerosos, representando menos de 10% dos voxels no modelo da P-50, e seu maior custo é a rasterização, que será evitada no caso dos voxels invisíveis pelo descarte antecipado por Z (*Early Z Cull*) que será bastante eficiente já que a maior parte da geometria da cena já está representada. O descare antecipado por Z é realizado nas placas 3D modernas. Nessa técnica, fragmentos que estejam

completamente ocultos por outros elementos já desenhados no *famebuffer* são descartados antes da etapa de rasterização, evitando que este tenha que ser processado nessa etapa para ser descartado depois. Essa técnica é especialmente útil quando *shaders* complexos são usados.

Ordenar todas as primitivas de trás pra frente causaria problemas de desempenho. Realizar qualquer operação de ordenação por primitiva em software irá contribuir para transferir o gargalo para a CPU, além de obrigar as primitivas ordenadas a serem transferidas para a GPU com maior freqüência. Por sorte, como a resolução dos voxels é escolhida de forma que eles ocupem apenas um pixel na tela, mesmo sem realizar nenhuma ordenação, as falhas visuais causadas são praticamente invisíveis, criando apenas variações na transparência percebida quando estes são vistos de um ângulo que sobreponha vários voxels na ordem errada. Para reduzir as chances de que muitos voxels sejam sobrepostos na ordem errada, basta organiza-los em ordem aleatória dentro do vetor de vértices.

Por outro lado, os grupos de voxels correspondentes a um nó na hierarquia de cena, podem ser ordenados de trás para frente sem causar problemas de desempenho.

3.6. Limitações

Abordagens de LOD Hierárquico em geral impõem uma grande limitação quanto à seleção e movimentação de objetos da cena. Como essas implementações funcionam agrupando os objetos existentes para gerar a hierarquia de níveis de detalhe, as divisões entre os objetos existentes originalmente no modelo são perdidas. Além de dificultar atividades como a seleção de objetos, essa característica torna impossível movimentar os objetos. Apesar de os objetos estarem representados em suas formas originais nas folhas da estrutura hierárquica gerada e ser possível movê-los nesse nível, nos níveis intermediários esses objetos estarão combinados com inúmeros outros para formar a representação simplificada do conjunto. Assim, fica impossível atualizar de forma simples a representação criada para esses níveis.

Por outro lado, mesmo que objetos individuais no modelo não possam ser movidos, ainda é possível mover o modelo por inteiro. Assim, podemos ter uma cena que integre vários modelos otimizados com a técnica de Voxels Distantes e cada um deles poderia ser movimentado livremente (Figura 3.17).

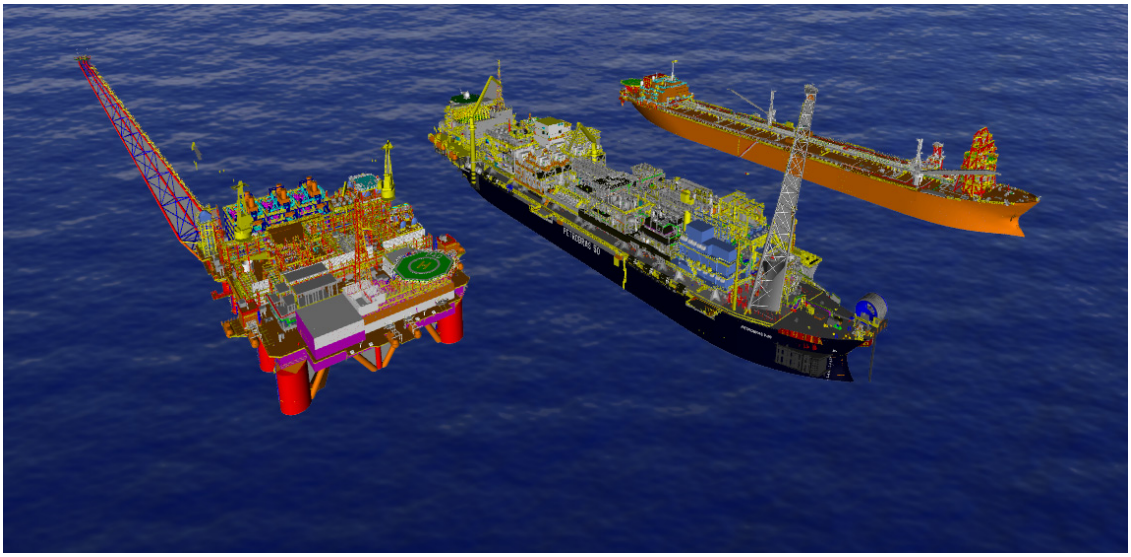

Figura 3.17 – Visualização envolvendo várias plataformas que podem ser movimentadas individualmente. Porém, seus objetos internos, não podem ser modificados, uma vez que cada plataforma possui uma hierarquia pré-definida de níveis de detalhe, ficando inviável sua modificação a cada instante.

Outro fator é que o processo de geração de voxels leva em consideração a resolução da tela e o ângulo de abertura da câmera para determinar quais voxels podem deixar de ser representados, o que obriga que os modelos sejam gerados de acordo com a visualização em que serão usados. Na realidade, o modelo otimizado não terá uma dependência direta com a resolução da tela nem com a abertura da câmera, e sim com o tamanho que um voxel terá quando projetado na tela, que pode ser obtido a partir destes dois parâmetros.

Outra limitação é o elevado tempo de pré-processamento gasto pelo algoritmo de geração de voxels (ver capítulo 5). Isso torna inviável o uso dessa técnica de otimização em operações que exijam uma conversão imediata do modelo.

4
Implementação

Neste capítulo serão explicados detalhes da implementação do pré-processador e do visualizador.

Por estarmos lidando com modelos compostos por vários milhões de triângulos, mostrou-se importante que o algoritmo de geração de voxels fosse desenvolvido de forma eficiente, apesar dessa ser uma etapa de pré-processamento. Uma implementação descuidada poderia elevar o tempo de pré-processamento de algumas horas para alguns dias. A operação mais freqüente realizada durante essa etapa é a amostragem de pontos usando o algoritmo de traçado de raios, o que criou a necessidade de uma estrutura espacial que permitisse que o tempo gasto na realização dos testes de interseção dos raios traçados com a geometria não fosse diretamente proporcional à complexidade da cena.

Foram implementadas duas versões dessa estrutura. Na primeira, usou-se uma *octree* semelhante à usada na geração da hierarquia, mas com mais níveis, de forma que as folhas da hierarquia fossem formadas por uma quantidade pequena de triângulos. Porém, essa implementação simples não apresentou desempenho satisfatório.

Na segunda implementação foi usado o motor de simulação física PhysX (2005), que tem recebido bastante atenção dos desenvolvedores de jogos e aplicações interativas por conseguir acelerar a simulação física usando hardware dedicado. Essa biblioteca, como qualquer motor físico, exige que seja criada uma estrutura de indexação para organizar os objetos da cena e permitir que testes de colisão entre objetos sejam realizados de forma eficiente. O algoritmo de pré-processamento implementado nesta dissertação tira vantagem dessa estrutura usando o suporte a traçado de raios oferecido pela biblioteca, que é mais eficiente que a primeira estrutura simples implementada.

Um problema em se usar o PhysX, uma biblioteca voltada para o mundo dos jogos, para trabalhar com modelos massivos é que ela não é planejada para lidar nem com tantos objetos nem com objetos tão complexos compostos por

muitos triângulos. Nenhuma dessas limitações está descrita no manual da biblioteca, e foram descobertas durante o desenvolvimento do pré-processador. Essa característica teve que ser contornada através da criação de um novo grafo de cena antes dos dados serem fornecidos ao motor do PhysX.

A biblioteca impõe que os objetos usados com ela tenham no máximo 60.000 triângulos. O tratamento dado aos objetos muito tesselados é simples, bastando quebrá-los em 2 ou mais objetos menores de forma que cada parte tenha uma quantidade aceitável de faces.

Outra limitação é de que cada "cena" do PhysX contenha no máximo 32.000 objetos. Uma solução simples consistiu em quebrar o modelo todo em diversas cenas com a quantidade de objetos suportados. Para evitar que o tempo gasto no teste de interseção dos raios comece a aumentar de forma proporcional ao número de cenas e, consequentemente, ao tamanho do modelo, podemos particionar o modelo em pequenos pedaços de forma que cada cena do PhysX seja criada apenas com objetos de uma região limitada do modelo. Assim, podemos fazer um teste de interseção do raio com a caixa envolvente de cada uma dessas regiões, e pedir que o motor físico trabalhe apenas com as regiões em que houve interseção com a sua caixa envolvente.

As operações usadas com mais freqüência no pré-processador foram desenvolvidas de forma a usar múltiplos processadores caso esses estejam disponíveis. Esse suporte foi dado usando a API OpenMP 2.0 criada por Dagum & Menon (1998), que hoje é suportada em diversos compiladores e linguagens. Essa API permite executar em paralelo códigos que possam ser estruturados em laços (*loops*). Seu uso é simples e utiliza diretivas *#pragma* como *#pragma omp parallel for* para realizar a execução de laços em paralelo e *#pragma omp critical* para obrigar que trechos do laço sejam executados em apenas uma *thread* por vez. Por utilizar diretivas *#pragma*, o mesmo código paralelizado ainda funciona, sem precisar de alterações, em compiladores mais antigos. As diretivas também podem ser facilmente desabilitadas para que o código funcione em apenas um processador. Um código típico usando OpenMP é escrito da seguinte forma:

```
função gerarVoxelsDeUmNó( Nó nó )
   // Esse for será executado em paralelo, com uma thread
   // para cada processador disponível na máquina
   #pragma omp parallel for
   para i = 1 até totalDeRaios faça
      raio = calcularRaioAleatório();

      // Calcular a interseção do raio com a cena
      // é a operação mais cara do pré-processamento
      // e pode ser executada concorrentemente
      // em várias threads[1].
      inter = calcularInterseção( nó, raio )

      // Essa é uma região crítica, que altera dados.
      // e consequentemente tem que ser executada em
      // apenas um processador por vez
      #pragma omp critical
         salvarResultado( voxels, raio, inter )
```

O visualizador foi implementado usando a biblioteca de grafos de cena *OpenSceneGraph* (Burns & Osfield., 2004). Essa biblioteca nos forneceu as classes necessárias para criarmos a hierarquia da cena e percorrê-la para gerar a renderização com descarte dos objetos fora da pirâmide de visão. O comportamento desejado do visualizador foi obtido através da extensão das classes existentes no grafo de cena, sendo que duas delas merecem destaque.

A primeira classe estendida tem a função de renderizar os nós de voxels, sendo responsável por carregar o *shader* apropriado para cada nó e enviar para a placa os vetores de parâmetros que contêm os dados que são acessados no *shader* para renderizá-lo. Essa classe também é responsável por verificar a

[1] Na realidade, por uma limitação da versão atual do PhysX este passo não pode ser paralelizado. O paralelismo foi, entretanto, utilizado em diversos outros pontos do programa.

disponibilidade de memória em vídeo para a criação de um *VertexBufferObject*, que permite a reutilização dos dados enviados à placa nos quadros seguintes.

A segunda classe reimplementada é responsável por redirecionar o percurso da hierarquia de cena de forma a escolher os níveis de detalhe apropriados em cada momento da visualização. Uma instância dessa classe é criada e inserida no grafo em cada posição em que for necessário decidir sobre usar a representação simplificada de voxels existente naquele nível ou continuar percorrendo os nós inferiores da hierarquia para usar representações de maior resolução. Essa decisão é baseada em parâmetros como a distância da câmera àquele nó, a resolução da superfície de visualização, a abertura da câmera e dados como tamanho e posição da representação de voxels disponível. Essa classe também é responsável por enviar os pedidos de carregamento de dados ao carregador que está rodando em segundo plano.

5
Resultados

Neste capítulo serão apresentados dados recolhidos sobre os modelos gerados em pré-processamento. A seguir, serão apresentados os testes realizados para verificar o desempenho do visualizador.

Os resultados obtidos podem ser divididos em dois grupos. No primeiro, temos os resultados que tentam analisar a qualidade da estrutura hierárquica gerada durante o pré-processamento. São eles os dados sobre distribuição de alturas da árvore gerada e as estatísticas de triângulos e voxels gerados por nó. No segundo grupo, temos testes de desempenho realizados usando o visualizador, que visam verificar se os diversos componentes funcionam bem em conjunto.

Tanto o pré-processamento quanto os testes de desempenho foram realizados em um computador com processador Athlon X2 64 4200+, 4 GB de memória RAM, placa 3D nVidia GeForce 8800 GTX com 768 MB de memória. A etapa de pré-processamento não fez nenhum uso da placa gráfica e foi executada no Windows Professional 64-bits, para poder usar toda a memória disponível. Os testes de desempenho foram realizados numa janela de 854x641 pixels e o driver da placa 3D foi configurado para usar filtro de anti-serrilhamento 4x e anti-serrilhamento de transparência por multi-amostragem, e foram rodados no Windows Professional 32-bits.

5.1
Modelos usados nos testes

Para testar o funcionamento e o desempenho do visualizador desenvolvido, foram usados 3 modelos de CAD de estruturas marítimas em operação pela Petrobras. Na tabela 5.1 apresentamos as características de cada um desses modelos. Nessa tabela também apresentamos as características do modelo otimizado gerado a partir deles para a visualização e o tempo que a etapa de pré-processamento gastou em cada um deles.

Modelo	Faces	Vértices	Tempo de geração	Nós de Voxels	Nós de Geometria	Faces por nó (média)	Voxels por nó (média)
P-38	8.4 milhões	9.7 milhões	4 h	195	150	56.237	24.313
P-40	22.3 milhões	19.3 milhões	22 h	843	701	31.811	19.986
P-50	29.8 milhões	37.9 milhões	27 h	945	770	38.701	21.237

Tabela 5.1. Características dos modelos usados nos testes.

O primeiro resultado do pré-processamento analisado visa medir o balanceamento da árvore gerada na etapa de criação da hierarquia. Como foi explicado na seção 3.1, uma KD-Tree é recomendada nesse tipo de divisão já que garante esse balanceamento. A *octree*, por outro lado, gera naturalmente uma árvore com menos níveis, o que é interessante para a nossa implementação. Os histogramas das figuras 5.1, 5.2 e 5.3 são criados a partir das alturas dos nós obtidas na divisão feita por *octree* nos nossos modelos de teste. Em todos os casos, a altura ficou limitada a no máximo 12 níveis com a maior parte dos nós folha concentrados em níveis próximos na hierarquia.

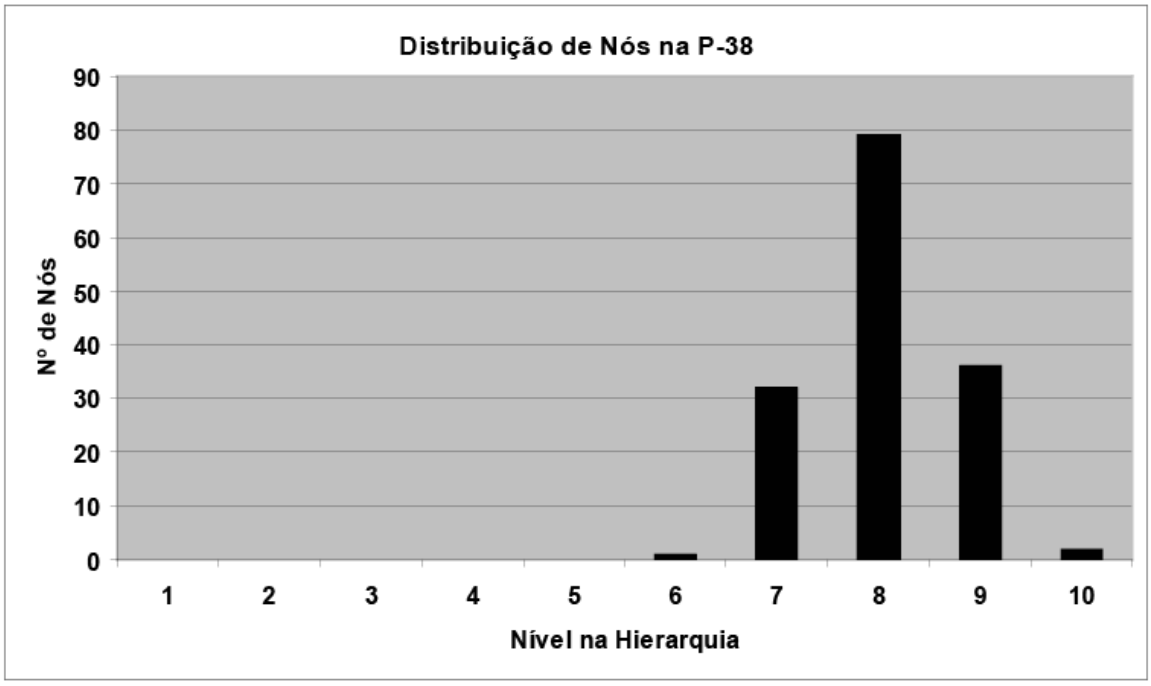

Figura 5.1 - Distribuição de nós na P-38

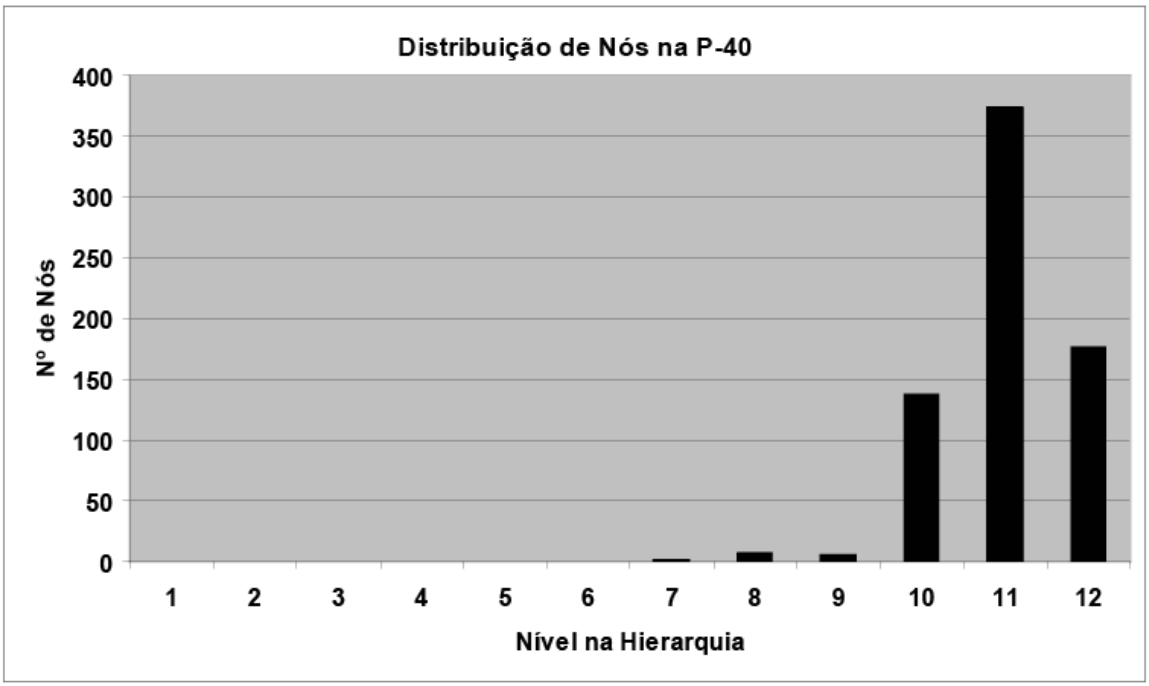

Figura 5.2 - Distribuição de nós na P-40

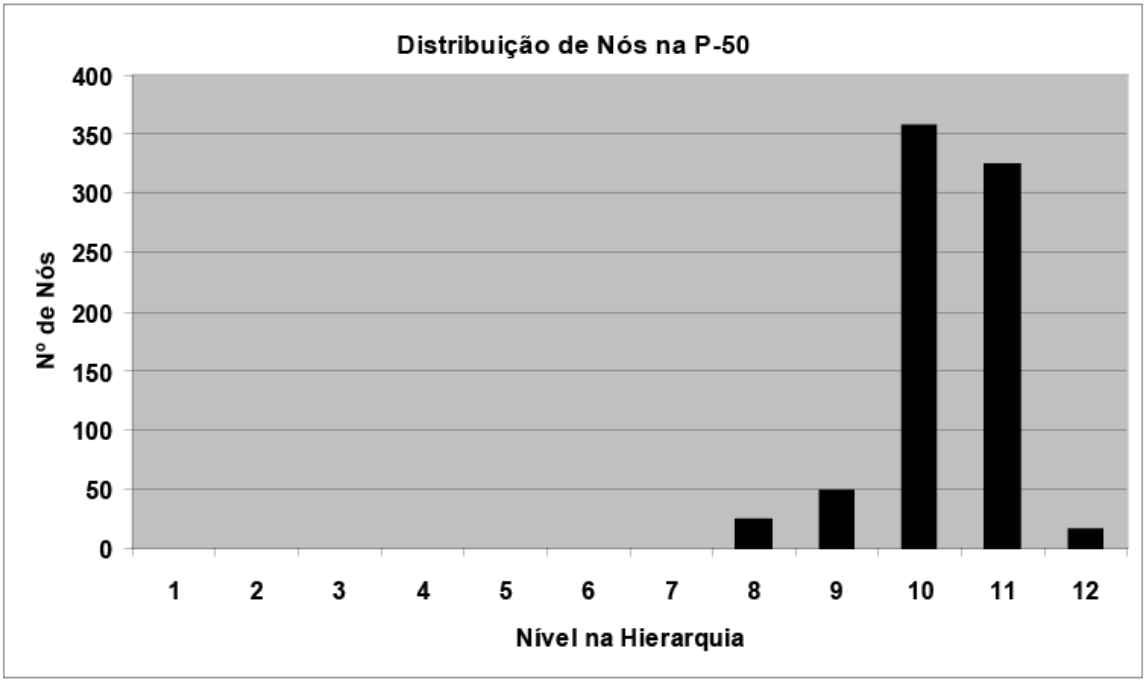

Figura 5.3 - Distribuição de nós na P-50

5.2
Testes de desempenho do algoritmo

O principal teste realizado foi o teste de desempenho. Esse teste simula um caminhamento típico dentro do modelo usando uma animação de câmera

pré-gravada. Para cada quadro renderizado durante o percurso recolhemos dados sobre a visualização e calculamos a taxa de quadros para gerar os gráficos comparativos. A seguir, realizamos o mesmo percurso modificando diversos parâmetros da renderização para descobrir qual estágio da cadeia de renderização está limitando o desempenho durante cada trecho da navegação.

Os dados recolhidos durante a renderização para testar o desempenho geram um gráfico formado por três linhas, denominado "Desempenho de Visualização" e apresentados a seguir. Essas linhas comparam o desempenho obtido pelo algoritmo para visualização usando voxels desenvolvido nesse trabalho com um algoritmo de visualização básico, sem suporte a LODs, mas com algumas otimizações mínimas para conseguir lidar com modelos massivos. Além disso, o desempenho do visualizador também é comparado com o obtido pelo mesmo algoritmo de voxels sem os testes de oclusão em hardware. Isso nos permite analisar separadamente o ganho obtido com o uso do LOD Hierárquico usando voxels e com os testes de oclusão em hardware. Nos gráficos que serão apresentados, o algoritmo com testes de oclusão é identificado como "Voxels com testes de oclusão" e o algoritmo sem esses testes é identificado como "Voxels sem testes de oclusão".

O algoritmo de visualização básico usado nessa comparação usa apenas triângulos para representar o modelo, agrupados em clusters para otimizar o envio de sua geometria para a placa gráfica. Assim, ele também consegue evitar que seja criado um gargalo em CPU durante o percurso da cena para seleção de objetos a serem renderizados. Esse algoritmo também descarta objetos contidos fora da pirâmide de visão, ordena por materiais as primitivas a serem renderizadas, para evitar a troca desnecessária de estados no OpenGL e envia os dados para a placa gráfica usando VBOs (*VertexBufferObjects*) para evitar o envio desnecessário desses dados em quadros posteriores através do barramento do PC.

Os testes realizados para identificar o gargalo da visualização são apresentados divididos em dois gráficos para cada modelo, sob o título "Análise de Gargalo". Esses testes envolvem eliminar ou reduzir de forma controlada o processamento nos diversos estágios da cadeia de renderização. A comparação

dos dados obtidos com essas modificações aplicadas permitem isolar com bastante segurança qual estágio é responsável por limitar o desempenho do visualizador. Essa informação pode ser usada posteriormente para guiar novos desenvolvimentos ou otimizações no mesmo. Foram realizadas navegações com 4 variações do algoritmo.

Na primeira variação, chamada de "Voxels sem desenho" nos gráficos que serão apresentados a seguir, todo o envio de dados à placa foi suprimido, mas o visualizador continua percorrendo normalmente a estrutura hierárquica gerada. Um ganho de desempenho nesse teste em comparação com os testes usando toda a cadeia de renderização indica que a placa gráfica é a responsável pelo gargalo. Caso o desempenho continue inalterado, temos a indicação de que a renderização está sendo limitada pelo desempenho da CPU.

Na segunda variação, denominada "Voxels sem iluminação" o cálculo de iluminação, que é realizado por vértices tanto para voxels como para geometria, foi removido. Assim, o estágio de transformação de vértices foi simplificado de forma que pudesse trabalhar sob menos carga. A geometria formada por triângulos passou a ser desenhada sem iluminação e a parte do *shader* responsável por calcular a iluminação do voxel foi removida. Outros cálculos realizados no *shader* dos voxels como a transformação da posição do vértice para coordenadas de *clipping* e o cálculo do tamanho do voxel tiveram que continuar sendo realizados, para não alterar a carga exercida sobre o estágio de rasterização. Um ganho de desempenho durante os testes com essa variação do algoritmo indica que o gargalo está sendo causado pelo estágio de transformação e iluminação de vértices.

A terceira variação, chamada de "Voxels sem anti-serrilhamento em placa" nos gráficos a seguir, consiste em usar o mesmo algoritmo usado na navegação com testes de oclusão, mas com o anti-serrilhamento da placa gráfica desabilitado. Um ganho de desempenho nesse teste indica que a rasterização é um gargalo.

Procurar o gargalo em uma placa gráfica como a GeForce 8800 GTX pode ser uma tarefa mais complicada do que em placas mais antigas. Por ser construída com uma arquitetura unificada, em que processadores genéricos

podem atuar tanto na etapa de transformação de vértices como na etapa de rasterização, é esperado que essas placas nunca apresentem gargalos em apenas uma dessas etapas. Como será demonstrado nas análises apresentadas a seguir, esse comportamento em que o gargalo está em dois estágios simultaneamente só foi observado em um dos três modelos usados.

5.2.1
Testes de desempenho com o modelo da P-38

O primeiro modelo testado foi o modelo CAD da P-38, nosso modelo mais leve. A Figura 5.4 mostra a trajetória da câmera usada durante a navegação, a Figura 5.5 mostra imagens tiradas em cada instante da visualização e a Figura 5.6 mostra a taxa de quadros obtida em cada instante da trajetória.

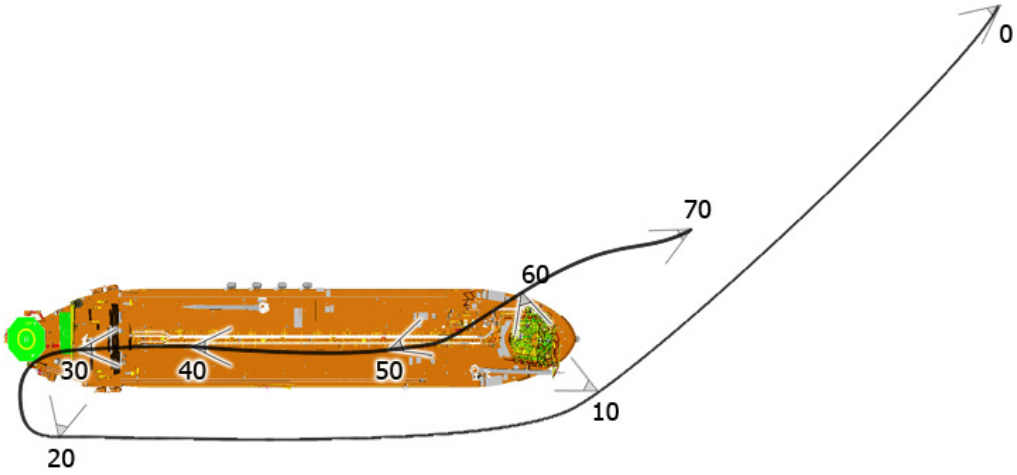

Figura 5.4 - Caminho percorrido durante o teste de desempenho na P-38. Os números indicam onde a câmera está em cada instante da animação

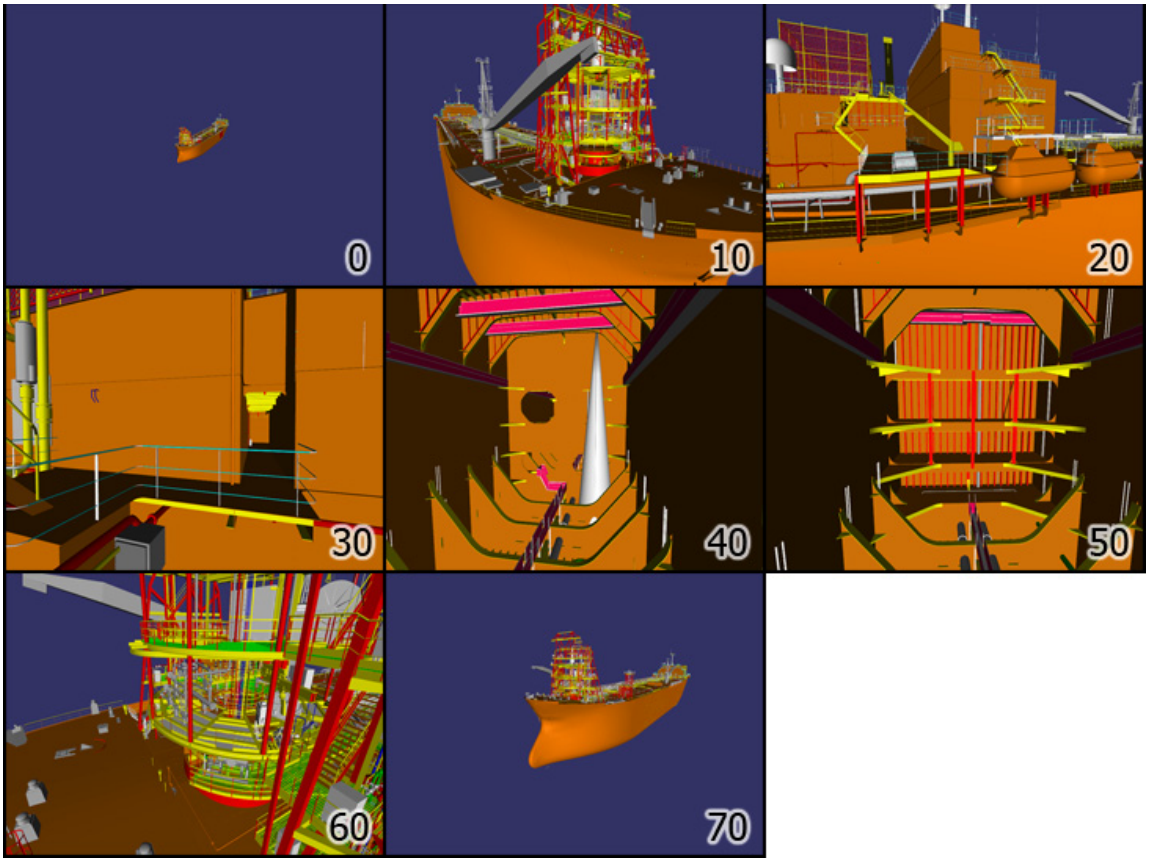

Figura 5.5 – Imagens de cada trecho do caminho percorrido durante os testes na P-38

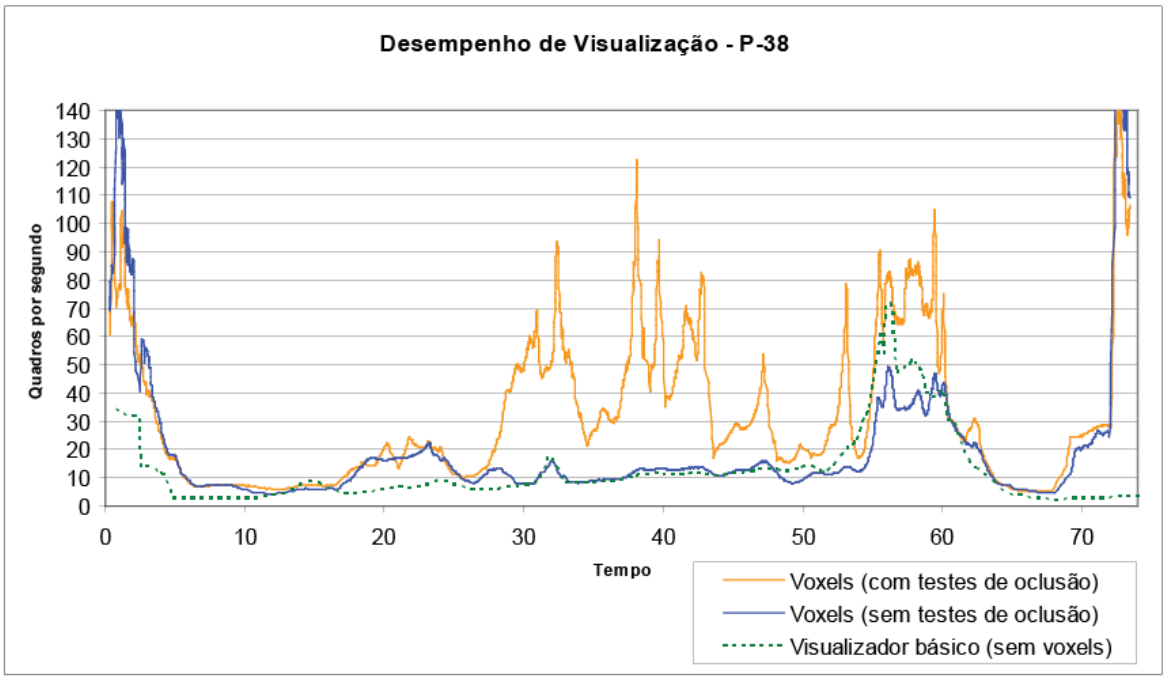

Figura 5.6 - Resultado em quadros por segundo dos testes de desempenho na P-38. Os valores de tempo se relacionam com as posições indicadas na Figura 5.4 e 5.5

O resultado mostra que, para modelos muito leves, a otimização usando voxels implementada nesse trabalho apresenta pouca vantagem. As maiores

diferenças são sentidas quando o modelo é visualizado de longe, como pode ser visto antes do tempo 10 e após o tempo 70, na figura 5.6. Os testes de oclusão conseguem dar um bom ganho quando a navegação ocorre em partes internas do modelo (entre os instantes 30 e 50).

Os próximos testes realizados visam identificar em quais etapas da cadeia de renderização ocorreram gargalos durante a navegação nesse modelo. Os resultados são apresentados nas Figuras 5.7 e 5.8 e são analisados a seguir.

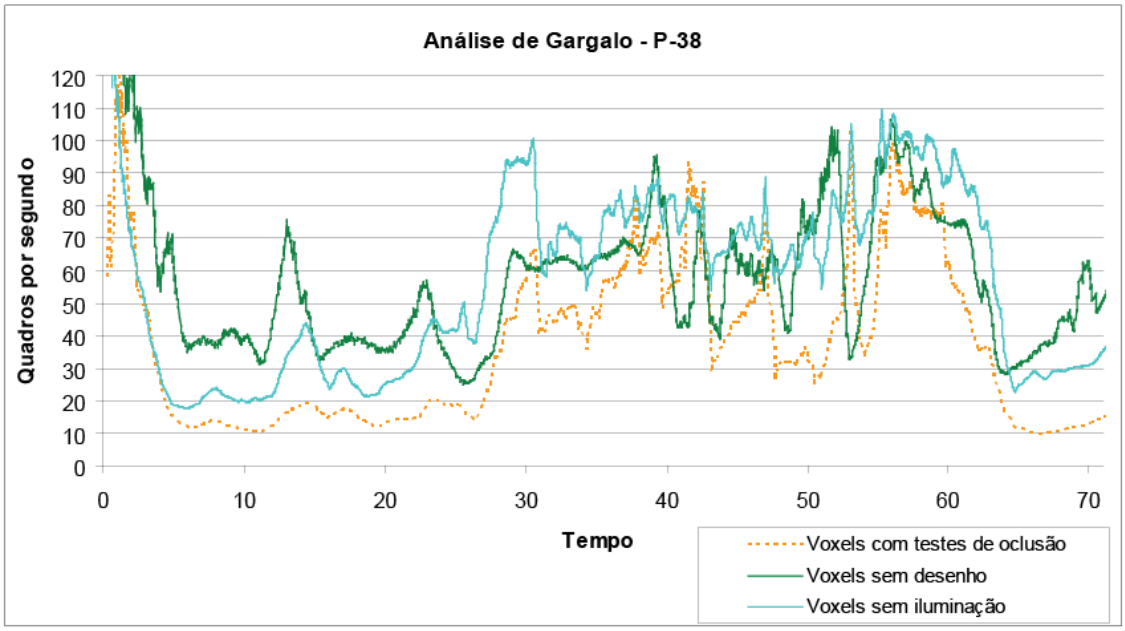

Figura 5.7 – Primeiro gráfico com resultados da análise realizada para localizar o gargalo na navegação da P-38. Esse gráfico compara os resultados obtidos na visualização normal, os resultados obtidos sem que nenhuma primitiva fosse enviada à placa e os resultados obtidos com o estágio de vértice simplificado

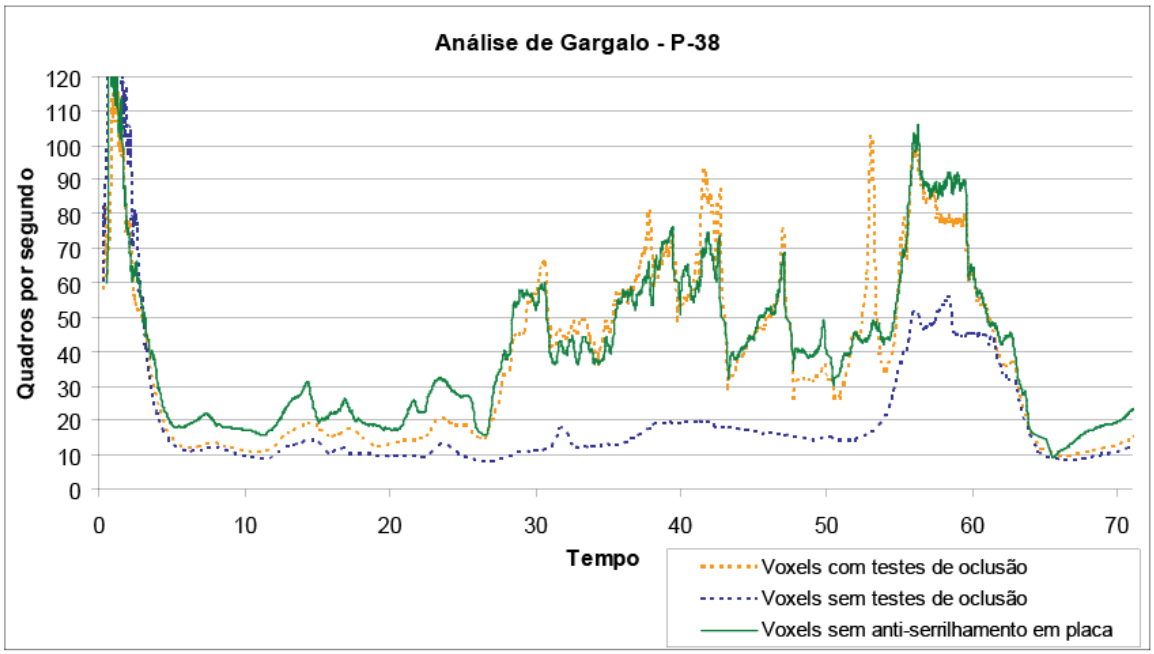

Figura 5.8 – Segundo gráfico com resultados da análise realizada para localizar o gargalo na navegação da P-38. Esse gráfico compara os resultados obtidos na visualização normal com os resultados obtidos sem o uso de anti-serrilhamento em hardware

Os primeiros resultados, apresentados na Figura 5.7, permitem afirmar que o gargalo não estava na CPU durante toda a navegação, já que desabilitar o desenho dos objetos e reduzir a complexidade do estágio de transformação de vértices trouxe ganhos de desempenho significativos durante todo o tempo do teste.

O ganho de performance obtido com a simplificação do estágio de transformação de vértices indica que este é um gargalo.

Os resultados da Figura 5.8 indicam dois resultados diferentes para o estágio de rasterização. A partir do tempo 25 do percurso, não há ganho de desempenho quando o anti-serrilhamento é desligado na placa gráfica, o que indica que o estágio de rasterização não é um gargalo durante a visualização nesse trecho.

Por outro lado, entre o tempo 5 e 25, houve ganho no desempenho quando o anti-serrilhamento foi desligado. Isso indicaria que o gargalo, durante essa parte da navegação, é a rasterização. Por outro lado, também há ganho quando a etapa de transformação de vértices é simplificada. Isso pode indicar que, nessa situação, a placa gráfica está conseguindo balancear a carga entre os estágios de

transformação de vértice e de rasterização como esperado na nova arquitetura unificada.

É interessante notar que a perda de desempenho por causa do anti-serrilhamento na placa gráfica só é percebida nos trechos em que os testes de oclusão não conseguem eliminar uma parte significativa do modelo, ou seja, nos trechos em que o desempenho com testes de oclusão não é muito maior que a o desempenho sem esses testes.

5.2.2 Testes de desempenho com o modelo da P-40

O segundo modelo testado foi o modelo de CAD da P-40, um modelo intermediário em termos de números de polígonos. Uma característica a ser notada nesse modelo é seu formato quadrado, que contribui para aumentar a concentração de geometria em torno da câmera em comparação com os outros dois modelos, em que a geometria se distribui ao longo de um comprimento maior. A Figura 5.9 mostra a trajetória da câmera usada durante a navegação, a figura 5.10 mostra imagens tiradas em cada instante da visualização e a Figura 5.11 mostra a taxa de quadros obtida em cada instante da trajetória.

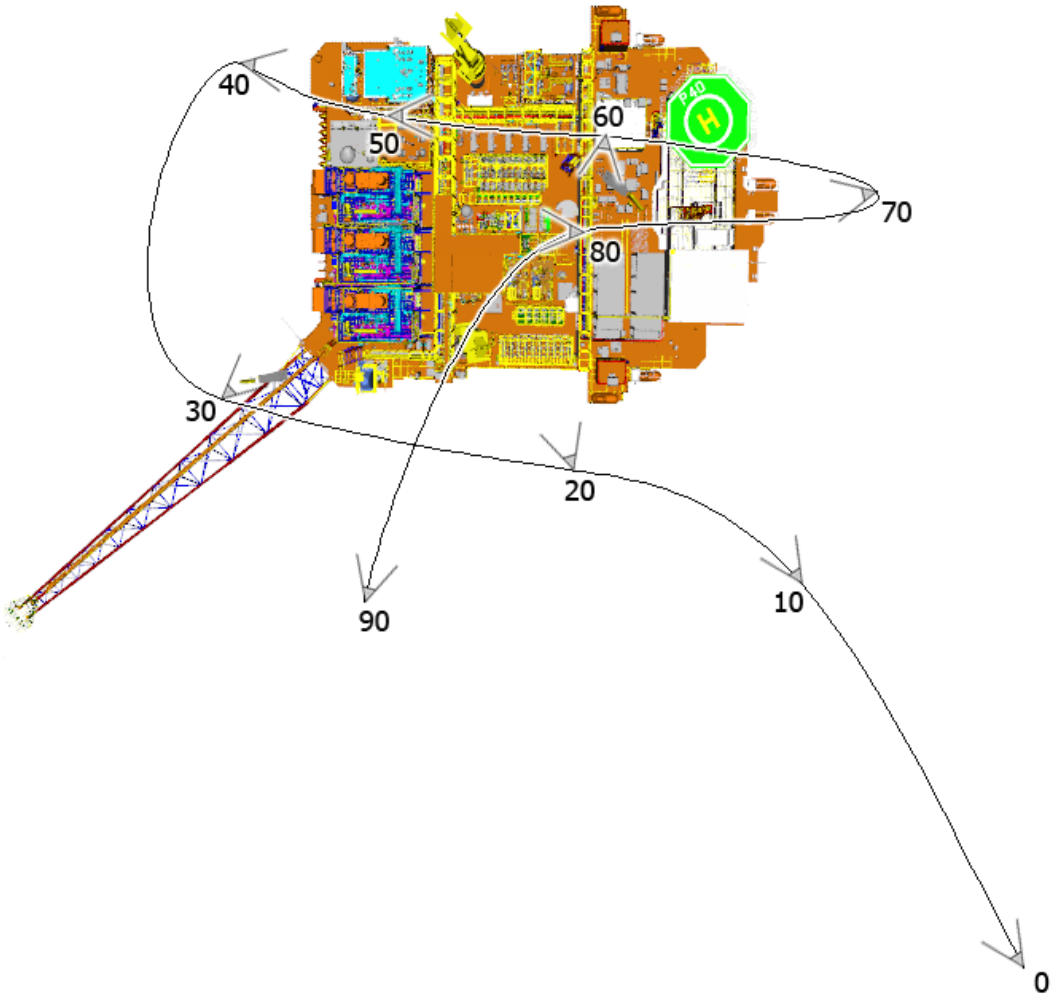

Figura 5.9 - Caminho percorrido durante o teste de desempenho na P-40. Os números indicam onde a câmera está em cada instante da animação

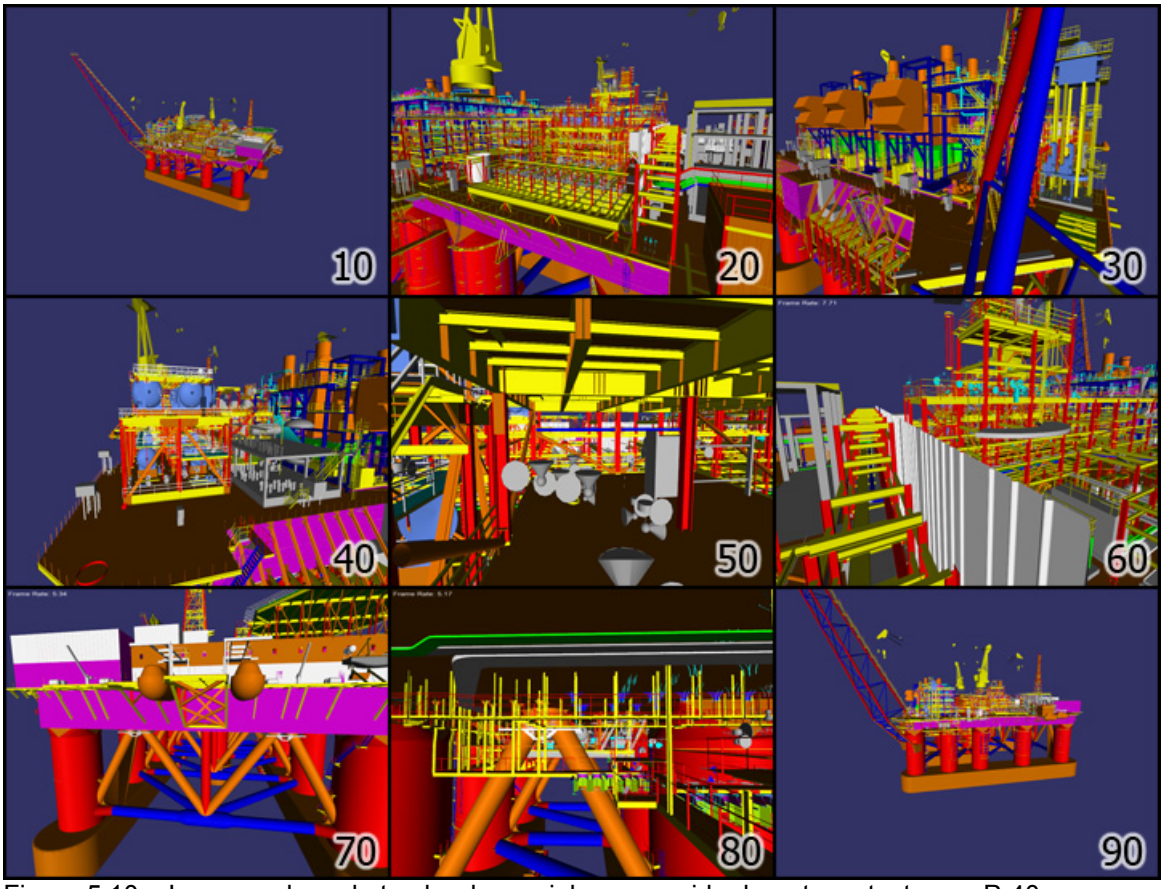

Figura 5.10 – Imagens de cada trecho do caminho percorrido durante os testes na P-40

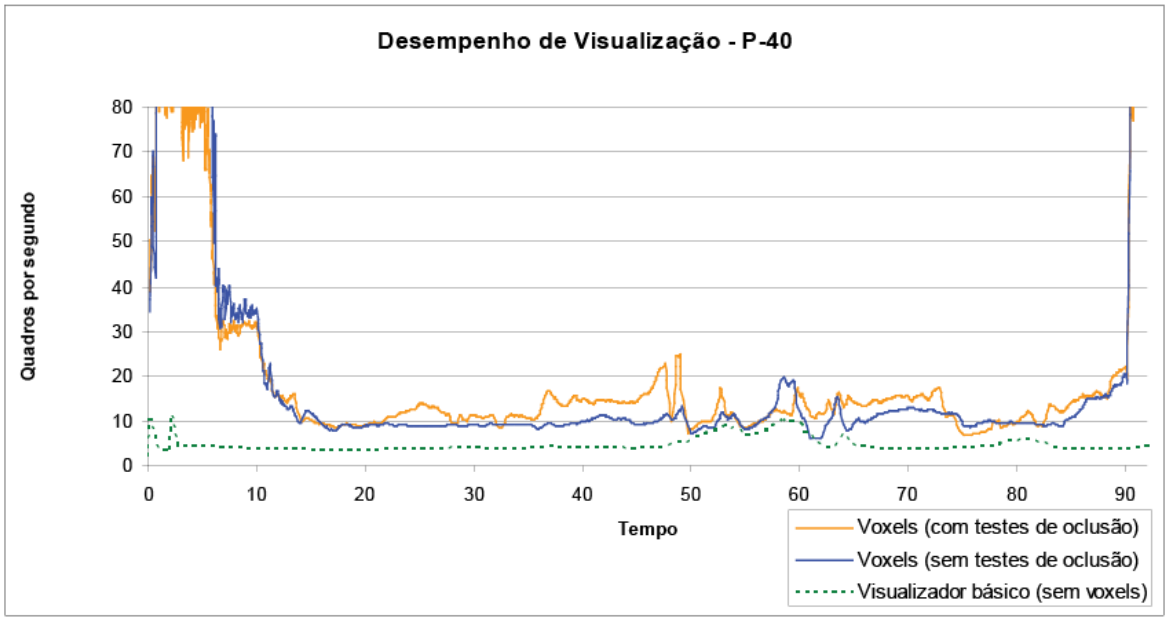

Figura 5.11 - Resultado em quadros por segundo dos testes de desempenho na P-40.
Os valores de tempo se relacionam com as posições indicadas nas Figura 5.9 e 5.10

O resultado apresentado mostra que, para modelos razoavelmente complexos, a otimização usando Voxels Distantes implementada nesse trabalho

apresenta um ganho de desempenho razoável em praticamente todos os trechos da navegação. Os testes de oclusão realizados conseguem melhorar ainda mais o desempenho em diversos trechos, principalmente naqueles em que a navegação ocorre dentro de áreas do modelo que estejam envolvidas por estruturas.

Os próximos testes realizados visam identificar em quais etapas da cadeia de renderização ocorreram gargalos durante a navegação nesse modelo. Os resultados são apresentados nas Figuras 5.12 e 5.13 e são analisados a seguir.

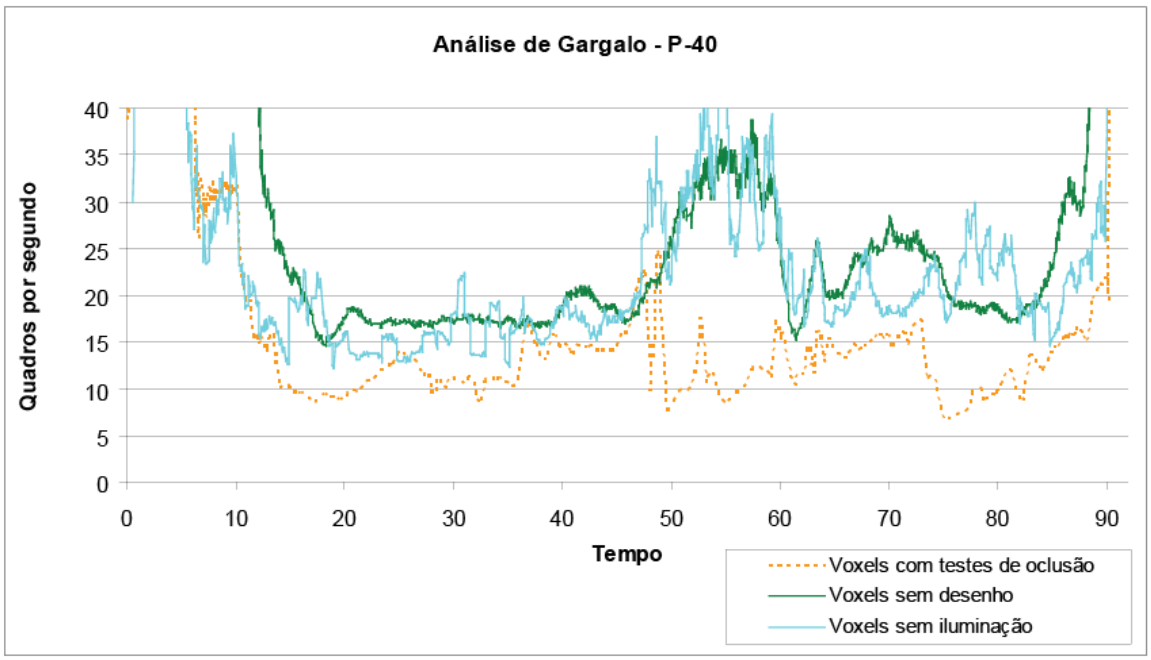

Figura 5.12 – Primeiro gráfico com resultados da análise realizada para localizar o gargalo na navegação da P-40. Esse gráfico compara os resultados obtidos na visualização normal, os resultados obtidos sem que nenhuma primitiva fosse enviada à placa e os resultados obtidos com o estágio de vértice simplificado

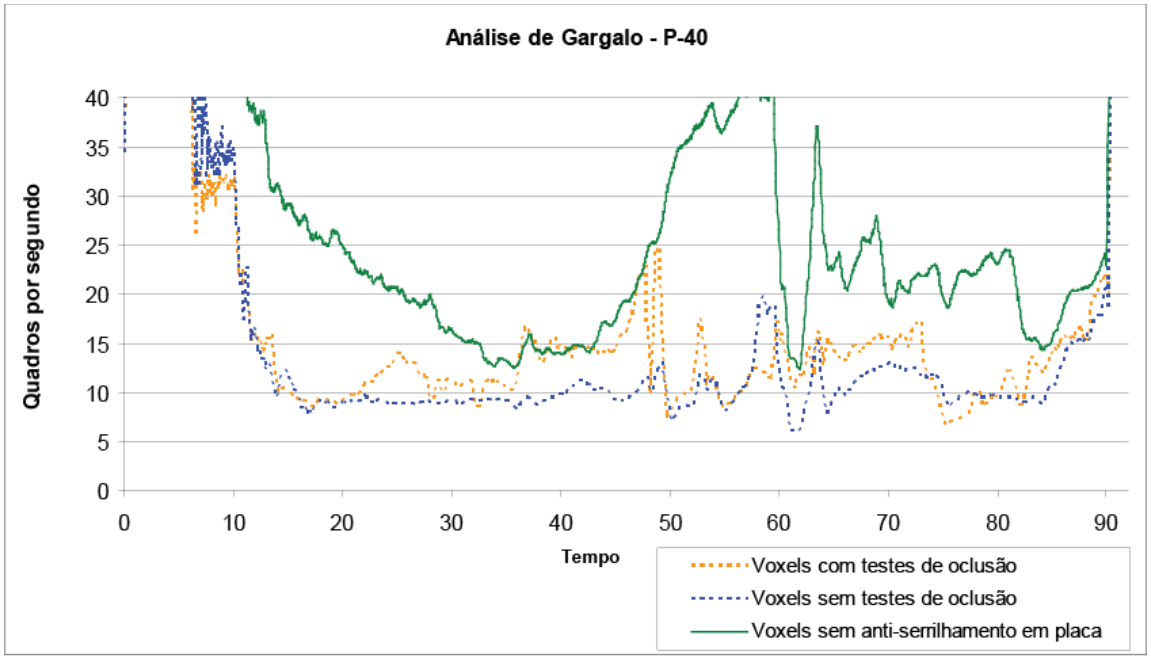

Figura 5.13 – Segundo gráfico com resultados da análise realizada para localizar o gargalo na navegação da P-40. Esse gráfico compara os resultados obtidos na visualização normal com os resultados obtidos sem o uso de anti-serrilhamento em hardware

Pelo gráfico apresentado na Figura 5.12, podemos concluir que o gargalo não está no processamento realizado em CPU, já que ao desabilitarmos o desenho, sempre houve ganho de desempenho. O mesmo gráfico aponta a sobrecarga no estágio de transformação de vértices, que pode ser comprovada pelo ganho de desempenho obtido com a navegação realizada com o estágio de vértices simplificado.

O gráfico apresentado na Figura 5.13 apresenta um resultado bem diferente do observado nos outros dois modelos. Aqui, desligar o anti-serrilhamento trouxe um ganho de desempenho considerável em todos os trechos do caminho percorrido. Assim, podemos afirmar que para a navegação nesse modelo, o desempenho está sendo limitado pela capacidade de rasterização da placa gráfica, além de estar sendo limitado pelo desempenho do estágio de transformação de vértices.

5.2.3
Testes de desempenho com o modelo da P-50

O terceiro modelo testado foi o modelo de CAD da P-50, o modelo com mais polígonos dentre os três testados nessa dissertação. A Figura 5.14 mostra a trajetória da câmera usada durante a navegação, a figura 5.15 mostra imagens tiradas em cada instante da visualização e a Figura 5.16 mostra a taxa de quadros obtida em cada instante da trajetória.

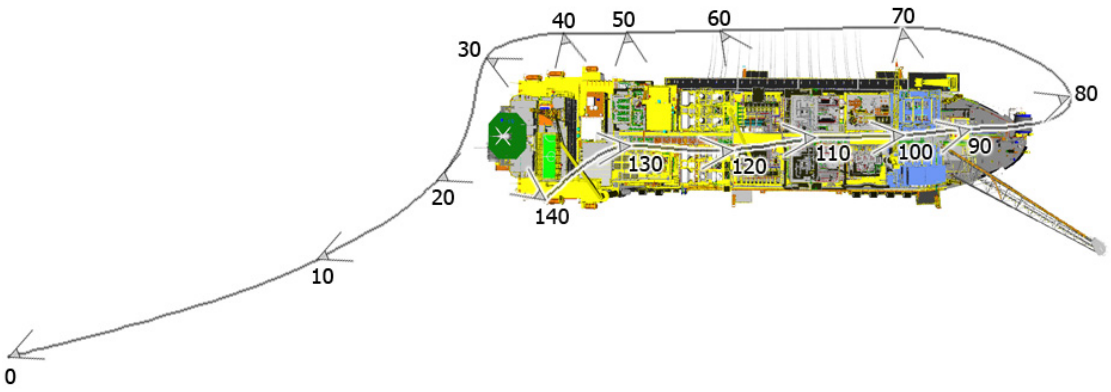

Figura 5.14 - Caminho percorrido durante o teste de desempenho na P-50. Os números indicam onde a câmera está em cada instante da animação

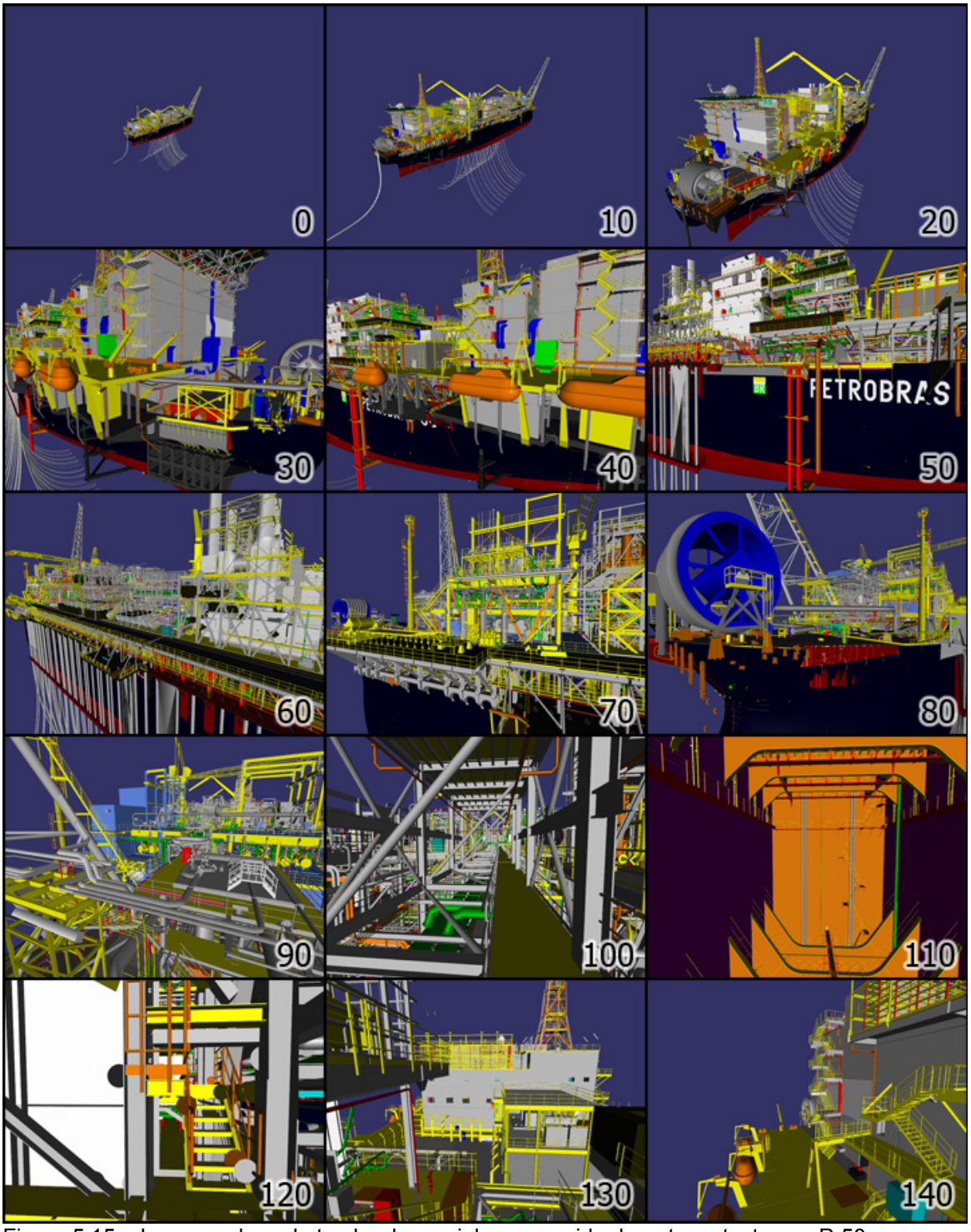

Figura 5.15 – Imagens de cada trecho do caminho percorrido durante os testes na P-50

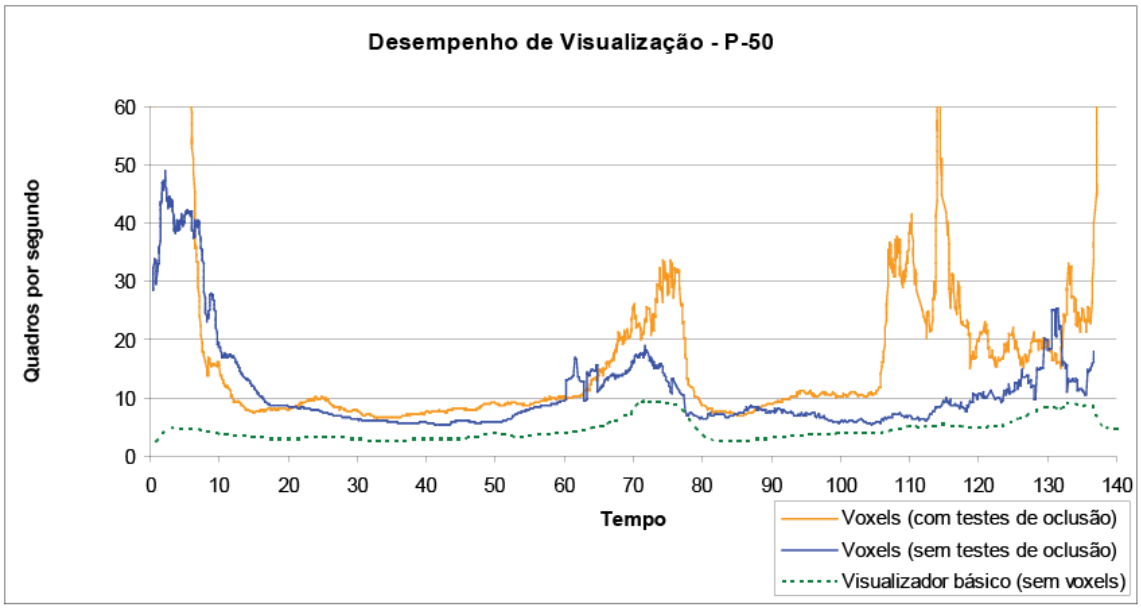

Figura 5.16 - Resultado em quadros por segundo dos testes de desempenho na P-50. Os valores de tempo se relacionam com as posições indicadas na Figura 5.14 e 5.15

O resultado apresentado mostra que, para esse modelo, o algoritmo de Voxels Distantes trouxe um bom ganho de desempenho. Os testes de oclusão realizados contribuíram para melhorar ainda mais o desempenho, principalmente nos trechos em que a navegação ocorreu nas partes internas do modelo, entre os tempos 100 e 120.

Os próximos testes realizados visam identificar em quais etapas da cadeia de renderização ocorreram gargalos durante a navegação nesse modelo. Os resultados são apresentados nas Figuras 5.17 e 5.18 e são analisados a seguir.

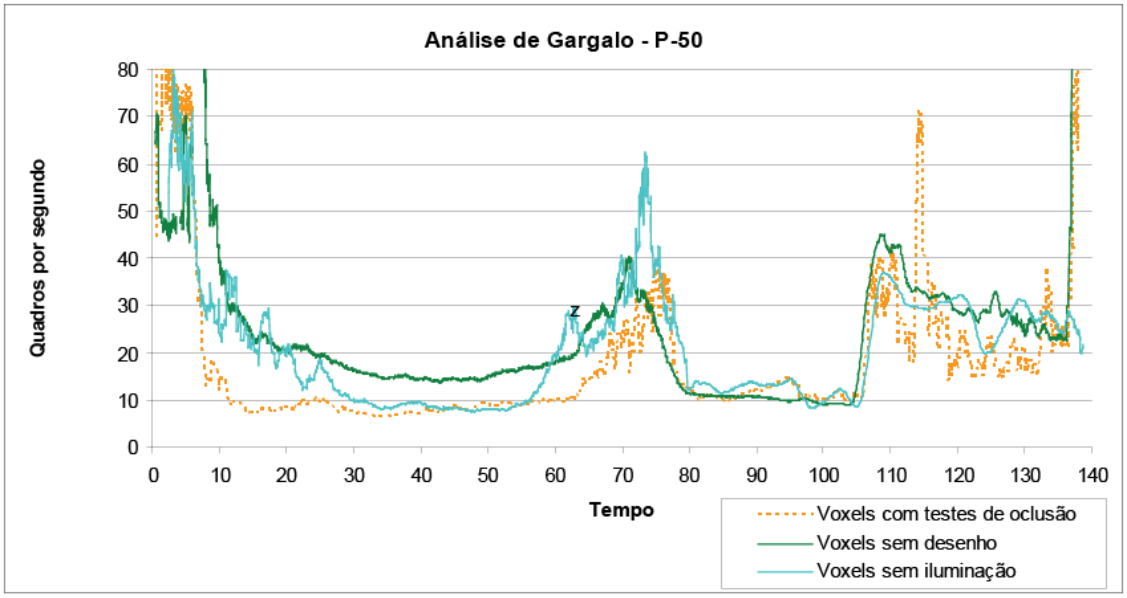

Figura 5.17 – Primeiro gráfico com resultados da análise realizada para localizar o gargalo na navegação da P-50. Esse gráfico compara os resultados obtidos na visualização normal, os resultados obtidos sem que nenhuma primitiva fosse enviada à placa e os resultados obtidos com o estágio de vértice simplificado

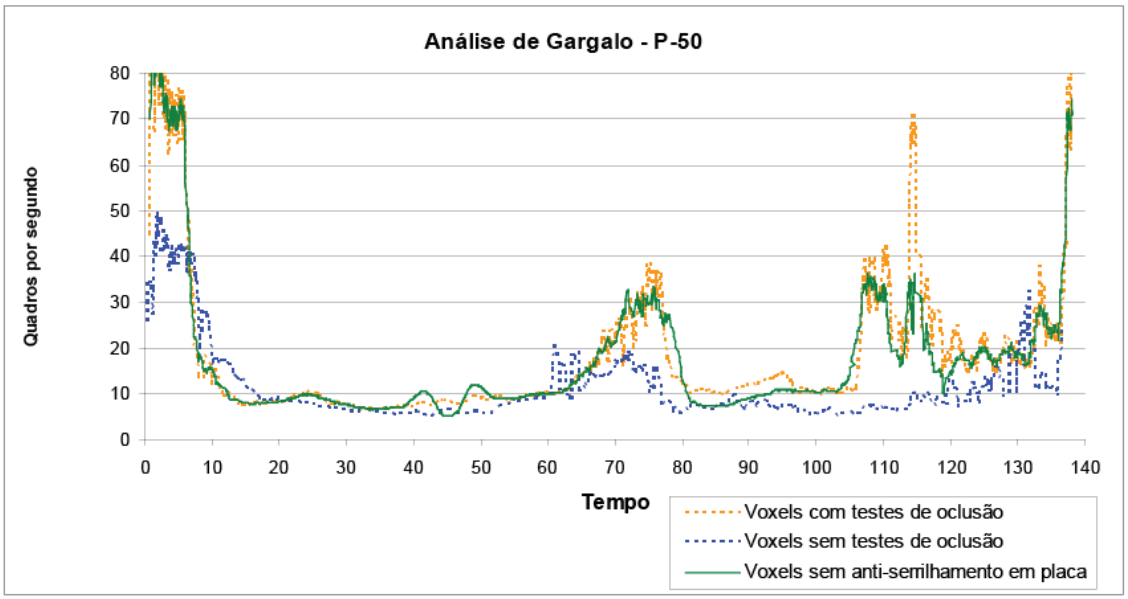

Figura 5.18 – Segundo gráfico com resultados da análise realizada para localizar o gargalo na navegação da P-50. Esse gráfico compara os resultados obtidos na visualização normal com os resultados obtidos sem o uso de anti-serrilhamento em hardware

O gráfico apresentado na Figura 5.17 mostra que na maior parte da navegação, o desempenho não estava sendo limitado pela CPU. Apenas nos instantes entre os tempos 80 e 105, desabilitar o desenho de primitivas não melhorou o desempenho da visualização. Podemos dizer que, nesse trecho, a

aplicação estava sendo limitada pela CPU.

O gráfico apresentado na Figura 5.18 mostra que não houve melhora no desempenho com o anti-serrilhamento da placa gráfica desligado. Isso indica que durante a navegação nesse modelo, a etapa de rasterização da placa gráfica não limitou a visualização.

A navegação realizada com o estágio de transformação de vértices simplificado apresentado na Figura 5.17 confirmou que o gargalo está na CPU no trecho entre os tempos 80 e 105, já que nesse teste, não houve ganho de desempenho. Um resultado inesperado foi encontrado no trecho entre os tempos 30 e 55. Nesse trecho, o visualizador não estava limitado pela CPU, nem pelo estágio de rasterização, e também não houve melhora significativa no desempenho quando o estágio de vértices foi simplificado. Esse resultado pode indicar que, nesse trecho, o desempenho está sendo limitado por algum outro elemento da cadeia de renderização, como por exemplo, a transferência de objetos recém carregados para a placa gráfica.

5.3 Testes de desempenho para os voxels com filtro de anti-serrilhamento

Na seção 3.5, foi apresentado o filtro que descreve um novo tipo de voxel que foi criado para contornar defeitos visuais causados por características encontradas nos modelos usados nesses testes.

Como os voxels criados usam primitivas transparentes e precisam ser renderizados após toda a geometria opaca da cena, é possível que eles tornem a visualização dos modelos que os usem mais lenta. Os testes apresentados nos gráficos das Figuras 5.19, 5.20 e 5.21 comparam o desempenho durante um percurso realizado nos modelos da P-38, P-40 e P-50, respectivamente, usando esse novo tipo de voxels e sem usá-los.

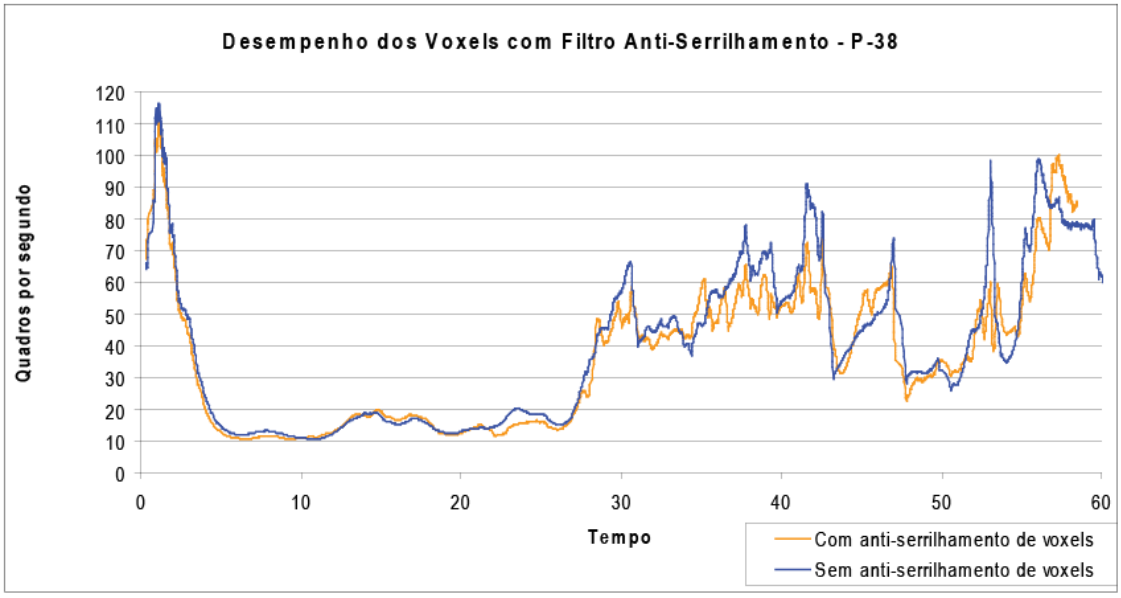

Figura 5.19 – Gráfico comparativo do desempenho usando voxels com o filtro anti-serrilhamento desenvolvido e sem usar esse filtro no modelo da P-38

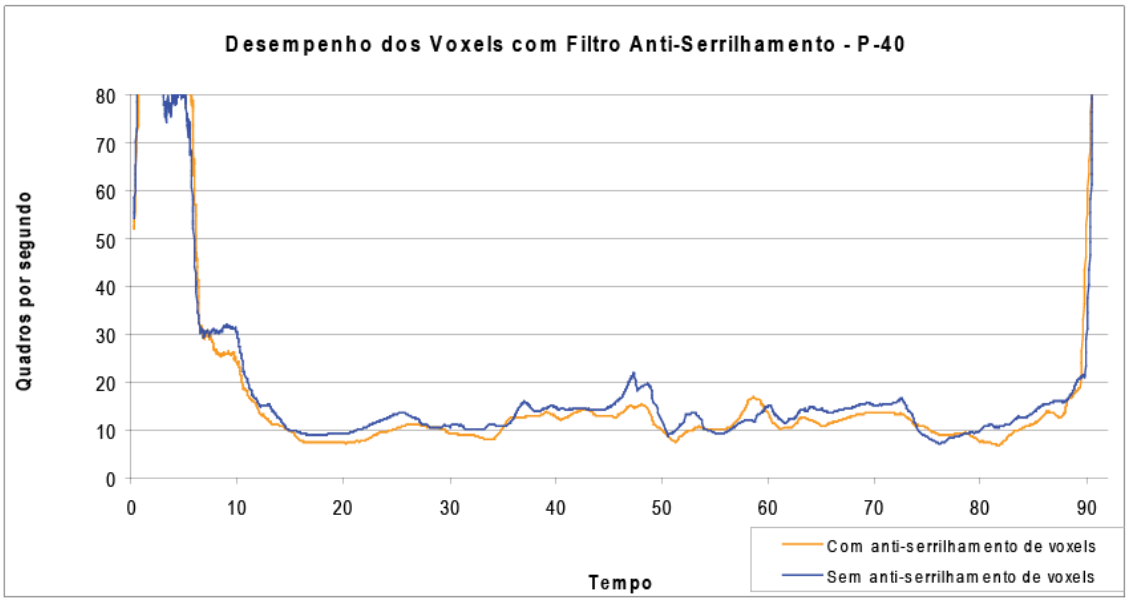

Figura 5.20 – Gráfico comparativo do desempenho usando voxels com o filtro anti-serrilhamento desenvolvido e sem usar esse filtro no modelo da P-40

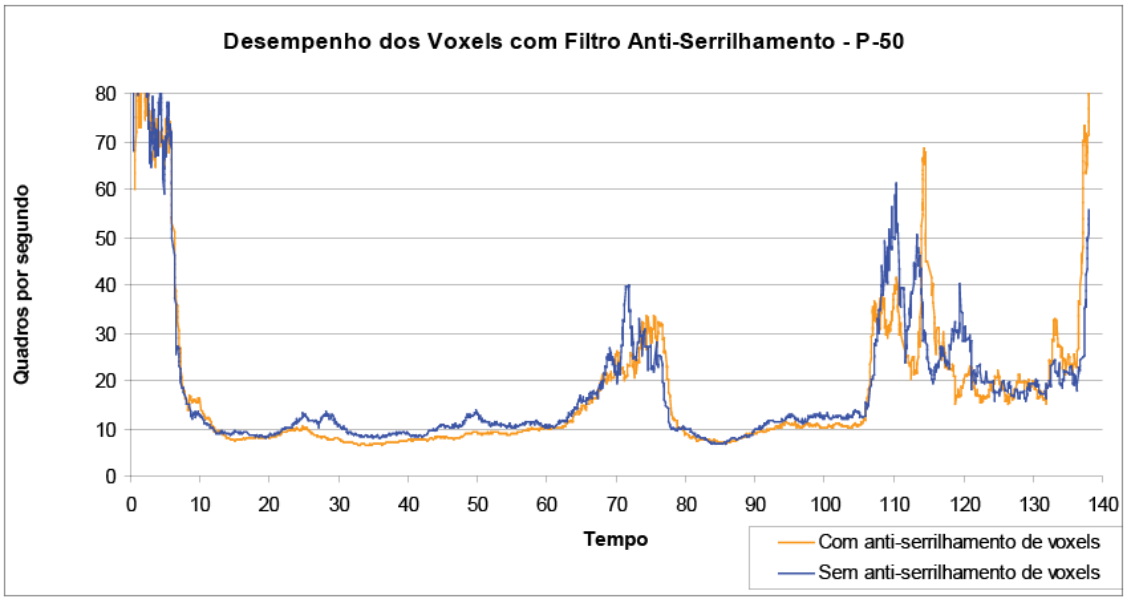

Figura 5.21 – Gráfico comparativo do desempenho usando voxels com o filtro anti-serrilhamento desenvolvido e sem usar esse filtro no modelo da P-50

Os resultados mostram que a perda de desempenho resultante do uso desses voxels é bem pequena, o que torna o seu uso perfeitamente viável na visualização dos modelos usados nesses testes.

5.4 Testes de desempenho com cópias dos modelos

O último teste realizado tem como objetivo verificar a escalabilidade do visualizador para modelos maiores. A existência de um modelo maior foi simulada com a criação de diversas cópias dos modelos existentes. Dessa forma, a complexidade total do modelo é aumentada, mas a densidade de triângulos em cada região do modelo continua a mesma. Um bom visualizador de modelos massivos deve conseguir manter um bom desempenho quando a complexidade do modelo cresce dessa forma.

No primeiro teste, foram criadas 25 cópias da P-50, o nosso modelo mais pesado, posicionadas como ilustrado na Figura 5.22. O caminho percorrido durante a navegação foi realizado em torno da P-50 posicionada no meio da cena, seguindo o caminho usado nos testes de desempenho para uma única P-50, que foi apresentado na seção 5.4. O resultado desse teste é apresentado na Figura 5.23.

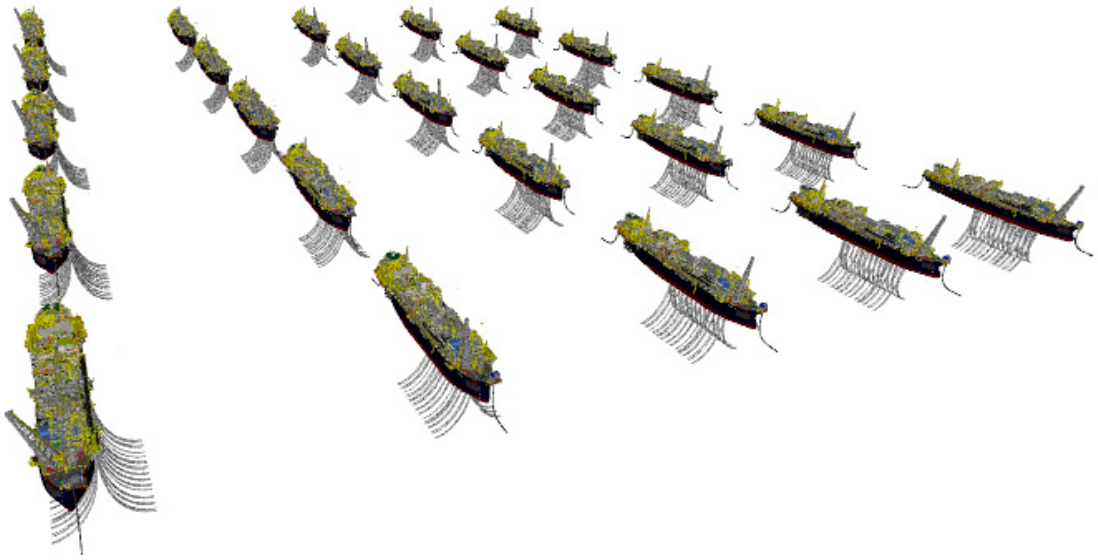

Figura 5.22 – 25 Cópias da P-50 posicionados lado-a-lado

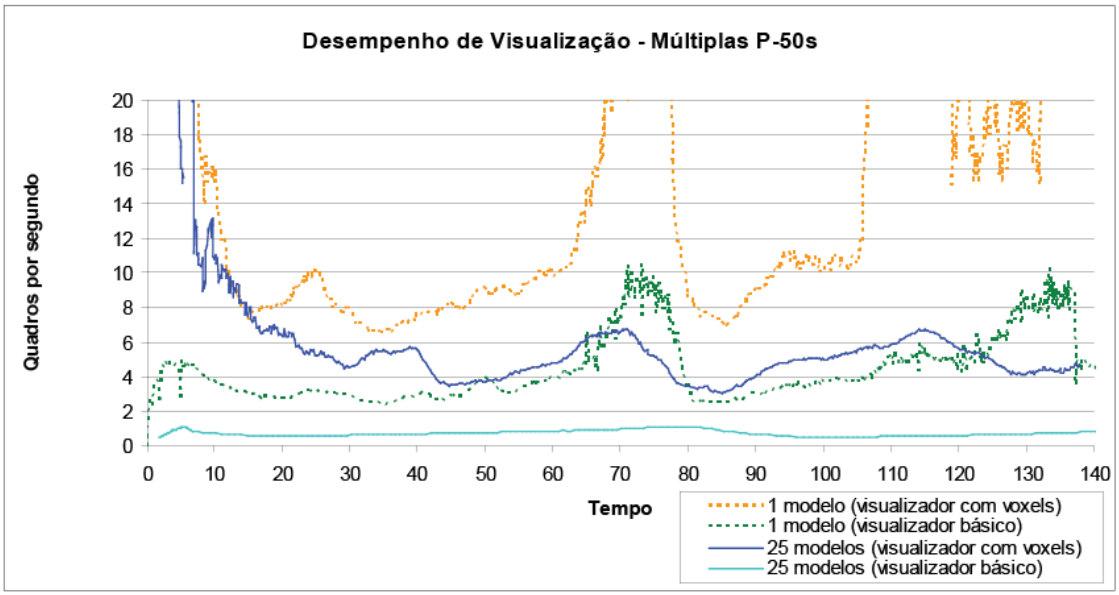

Figura 5.23 – Desempenho obtido com as 25 cópias da P-50

Nesse teste, podemos notar que houve alguma perda de desempenho devido à existência de várias plataformas, mas esta não ocorreu de forma proporcional ao número de cópias criadas. No visualizador básico, por outro lado, essa perda de desempenho foi bem mais acentuada.

No segundo teste realizado, foram criadas 9 cópias de cada uma dos 3 modelos usados nos testes de trabalho, totalizando 27 cópias. A disposição dessas cópias é apresentada na Figura 5.24.

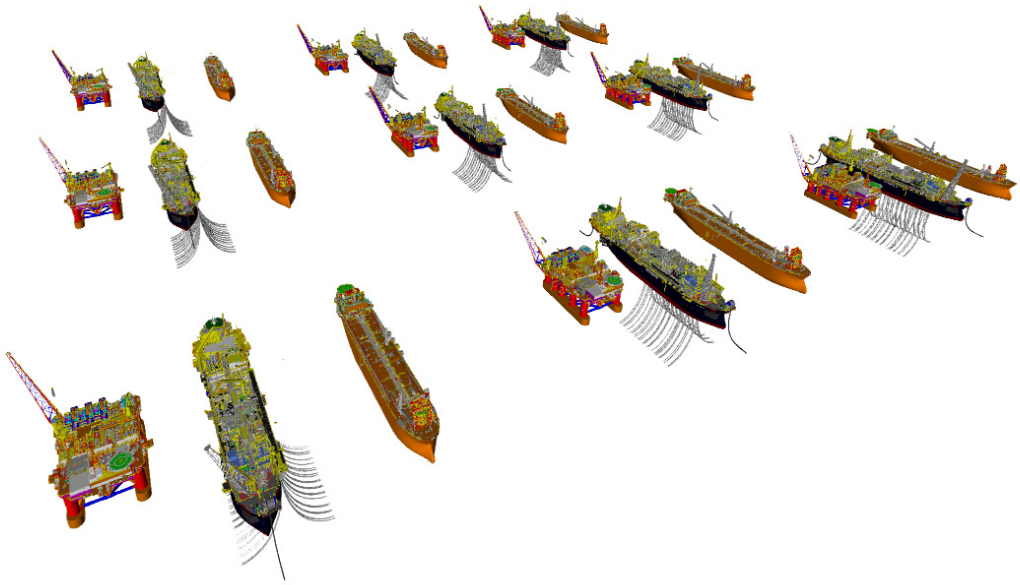

Figura 5.24 – 9 Cópias de cada um dos 3 modelos de teste posicionados lado-a-lado, totalizando 27 modelos

A navegação realizada nesse teste de desempenho também seguiu o mesmo caminho realizado no teste de desempenho com apenas uma P-50. Os modelos foram posicionados de forma que a navegação fosse realizada em torno da P-50 que está localizada no centro de todos os outros modelos. Na Figura 5.25, mostramos os resultados desse teste comparados com o resultado obtido com apenas um modelo da P-50. Como a versão básica do visualizador não é capaz de paginar dados do disco, não foi possível usá-lo nesse teste com diversos modelos diferentes.

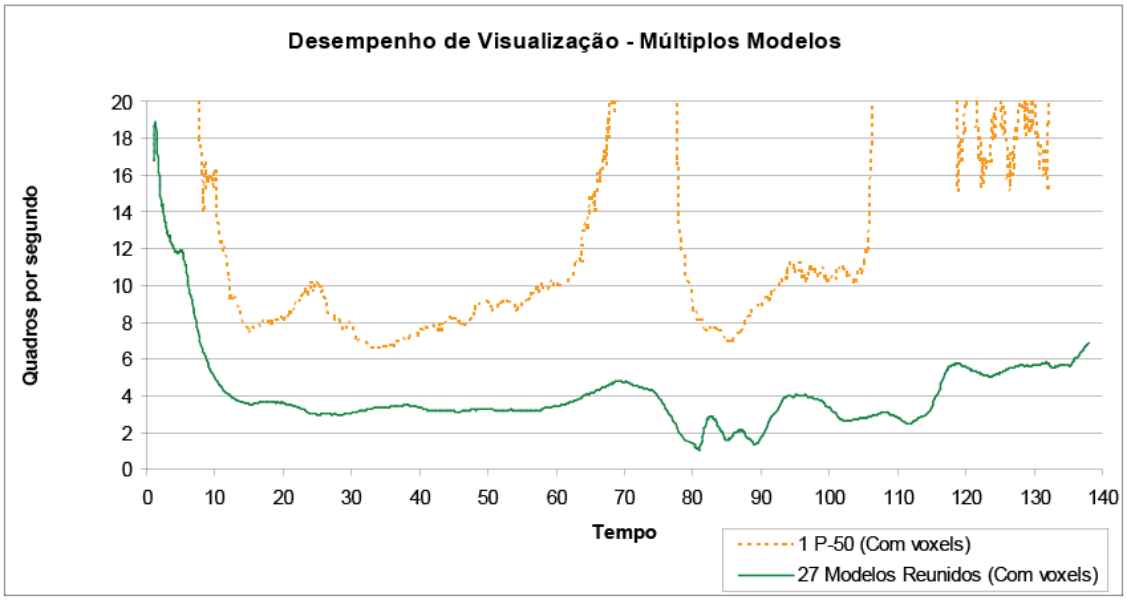

Figura 5.25 – Desempenho obtido com os 27 modelos.

Pelo gráfico da Figura 5.25, podemos notar que há alguma perda de desempenho quando comparamos essa navegação com a navegação realizada com apenas um modelo da P-50. Por outro lado, essa perda foi pequena se considerarmos o aumento de complexidade total da cena.

6
Conclusões e trabalhos futuros

Este trabalho abordou o problema de visualização interativa de modelos massivos de engenharia por meio da utilização do algoritmo de Voxels Distantes, uma das técnicas de maior evidência atualmente nessa área. Essa técnica pode ser separada em duas grandes etapas: pré-processamento e visualização.

O pré-processamento, além de gerar com bastante precisão representações simplificadas de partes do modelo, nos fornece uma informação muito útil de visibilidade. A etapa de visualização usa a informação gerada na etapa anterior para permitir a navegação no modelo.

A implementação realizada envolveu adaptações necessárias à visualização de estruturas marítimas que, devido às suas características particulares, apresentaram problemas não tratados originalmente por essa técnica. Para a melhoria da qualidade visual, foi implementado um mecanismo para a redução do efeito causado por voxels maiores que os objetos por eles representados. Para a etapa de pré-processamento, o traçado de raios foi simplificado por meio da utilização da biblioteca PhysX.

Os resultados foram avaliados em modelos reais de plataformas de petróleo e se mostraram significativamente superiores aos obtidos sem o uso da técnica de Voxels Distantes.

A implementação realizada ainda pode ser melhorada tanto no aspecto da qualidade visual quanto no aspecto do desempenho.

No aspecto visual, o filtro anti-serrilhamento pode ser melhorado para tratar de forma mais acurada os objetos menores que o voxel. Para isso, pretende-se criar um mecanismo que garanta uma amostragem regular de cada voxel para que se possa avaliar a localização e a forma dos objetos internos a ele. Assim, pode-se chegar a uma representação simplificada mais precisa dos objetos, que não precisaria estar restrita a 1 voxel por pixel.

No aspecto do desempenho, ainda pode-se melhorar aspectos do

visualizador que ainda limitam o desempenho, tais como o percurso da estrutura de cena e a forma de armazenamento de objetos na placa gráfica.

Também podem ser desenvolvidos mecanismos de predição do caminho tomado durante a navegação, que permitam antecipar a necessidade de carregar os nós necessários, para evitar que um atraso nesse carregamento cause falhas na visualização.

Outra possibilidade não abordada neste trabalho envolve o uso da informação de visibilidade produzida na etapa de pré-processamento para gerar outras formas de representação a serem usadas durante a visualização. Por exemplo, essa informação poderia ser usada apenas para ocultar objetos invisíveis durante a visualização, funcionando de forma semelhante a um PVS (Airey et al., 1990). Outro exemplo é usar essa informação para gerar níveis de detalhe usando triângulos, como em um LOD hierárquico tradicional.

Bibliografia

Airey, J., Rohlf, J. and Brooks, Jr. F. 1990. **Towards Image Realism with Interactive Update Rates in Complex Virtual Building**. *ACM Computer Graphics (Proceedings of Symposium on Interactive 3D Graphics)* 24:41-49.

Bittner, J., Wimmer, M., Piringer, H., and Purgathofer, W. 2004. **Coherent hierarchical culling: Hardware occlusion queries made useful.** Computer Graphics Forum 23, 3, 615--624.

Burns, D. and Osfield, R. 2004. **Open Scene Graph A: Introduction, B: Examples and Applications.** In *Proceedings of the IEEE Virtual Reality 2004 (Vr'04) - Volume 00* (March 27 - 31, 2004). VR. IEEE Computer Society, Washington, DC, 265. DOI= http://dx.doi.org/10.1109/VR.2004.57

Clark, J. H. 1976. **Hierarchical geometric models for visible surface algorithms.** *Commun. ACM* 19, 10 (Oct. 1976), 547-554. DOI= http://doi.acm.org/10.1145/360349.360354

Dagum L., Menon, R., **OpenMP: An Industry-Standard API for Shared-Memory Programming**, IEEE Computational Science & Engineering, v.5 n.1, p.46-55, January 1998

Dietrich A., Wald I., Benthin C., Slusallek P., **The OpenRT Application Programming Interface - Towards A Common API for Interactive Ray Tracing.** In Proceedings of the 2003 OpenSG Symposium (Darmstadt, Germany, 2003), Eurographics Association, pp. 23--31.

Duguet, F., Esteban, C., Drettakis, G., Schmitt, F. 2006, **Level of Detail Continuum for Huge Geometric Data**., Rapport De Recherche Inria, RR-5552. http://hal.inria.fr/inria-00070455/en

Dutré P., Bekaert, P., Bala, K., **Advanced Global Illumination**, A K Peters, Natick, Massachusetts, 2003.

Erikson, C. and Manocha, D. 1998 **Simplification Culling of Static and Dynamic Scene Graphs.** Technical Report. *UMI Order Number: TR98-009.*, University of North Carolina at Chapel Hill.

Erikson, C., Manocha, D., and Baxter, W. V. 2001. **HLODs for faster display of large static and dynamic environments.** In *Proceedings of the 2001 Symposium on interactive 3D Graphics* SI3D '01. ACM Press, New York, NY, 111-120. DOI= http://doi.acm.org/10.1145/364338.364376

Frisken, S. F., Perry, R. N., Rockwood, A. P., and Jones, T. R. 2000. **Adaptively sampled distance fields: a general representation of shape for computer graphics.** In *Proceedings of the 27th Annual Conference on Computer Graphics and interactive Techniques* International Conference on Computer Graphics and Interactive Techniques. ACM Press/Addison-Wesley Publishing Co., New York, NY, 249-254. DOI= http://doi.acm.org/10.1145/344779.344899

Gobbetti, E. and Marton, F. 2005. **Far voxels: a multiresolution framework for interactive rendering of huge complex 3D models on commodity graphics platforms.** ACM Trans. Graph. 24, 3 (Jul. 2005), 878-885. DOI= http://doi.acm.org/10.1145/1073204.1073277

Hoppe, H. 1996. **Progressive meshes.** In *Proceedings of the 23rd Annual Conference on Computer Graphics and interactive Techniques* SIGGRAPH '96. ACM Press, New York, NY, 99-108. DOI= http://doi.acm.org/10.1145/237170.237216

Hoppe, H. 1997. **View-dependent refinement of progressive meshes.** In *Proceedings of the 24th Annual Conference on Computer Graphics and interactive Techniques* International Conference on Computer Graphics and Interactive Techniques. ACM Press/Addison-Wesley Publishing Co., New York, NY, 189-198. DOI= http://doi.acm.org/10.1145/258734.258843

Luebke, D. and Erikson, C. 1997. **View-dependent simplification of arbitrary polygonal environments.** In *Proceedings of the 24th Annual Conference on Computer Graphics and interactive Techniques* International Conference on Computer Graphics and Interactive Techniques. ACM Press/Addison-Wesley Publishing Co., New York, NY, 199-208. DOI= http://doi.acm.org/10.1145/258734.258847

Maciel, P. W. and Shirley, P. 1995. **Visual navigation of large environments using textured clusters.** In *Proceedings of the 1995 Symposium on interactive 3D Graphics* (Monterey, California, United States, April 09 - 12, 1995). SI3D '95. ACM Press, New York, NY, 95-ff. DOI= http://doi.acm.org/10.1145/199404.199420

AGEIA PhysX: physics simulation library. AGEIA Technologies, Inc. (2005) Available: http://www.ageia.com/physx

Rusinkiewicz, S. and Levoy, M. 2000. **QSplat: a multiresolution point rendering system for large meshes.** In *Proceedings of the 27th Annual Conference on Computer Graphics and interactive Techniques* International Conference on Computer Graphics and Interactive Techniques. ACM Press/Addison-Wesley Publishing Co., New York, NY, 343-352. DOI= http://doi.acm.org/10.1145/344779.344940

Schroeder, W. J., Zarge, J. A., and Lorensen, W. E. 1992. **Decimation of triangle meshes.** In *Proceedings of the 19th Annual Conference on Computer Graphics and interactive Techniques* J. J. Thomas, Ed. SIGGRAPH '92. ACM Press, New York, NY, 65-70. DOI= http://doi.acm.org/10.1145/133994.134010

Wald, I., Dietrich, A., and Slusallek, P. 2005. **An interactive out-of-core rendering framework for visualizing massively complex models.** In *ACM SIGGRAPH 2005 Courses* (Los Angeles, California, July 31 - August 04, 2005). J. Fujii, Ed. SIGGRAPH '05. ACM Press, New York, NY, 17. DOI= http://doi.acm.org/10.1145/1198555.1198756

Xia, J. C., El-Sana, J., and Varshney, A. 1997. **Adaptive Real-Time Level-of-Detail-Based Rendering for Polygonal Models.** *IEEE Transactions on Visualization and Computer Graphics* 3, 2 (Apr. 1997), 171-183. DOI= http://dx.doi.org/10.1109/2945.597799

Yoon, S., Salomon, B., Gayle, R., and Manocha, D. 2004. **Quick-VDR: Interactive View-Dependent Rendering of Massive Models.** In *Proceedings of the Conference on Visualization '04* (October 10 - 15, 2004). IEEE Visualization. IEEE Computer Society, Washington, DC, 131-138. DOI= http://dx.doi.org/10.1109/VIS.2004.86

Yoon, S., Lauterbach, C., Manocha, D. 2006. **R-LODs: fast LOD-based ray tracing of massive models.** In *The Visual Computer (Pacific Graphics) 2006,* Tech. Report, TR06-009, Univ. of North Carolina at Chapel Hill, 2006