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.

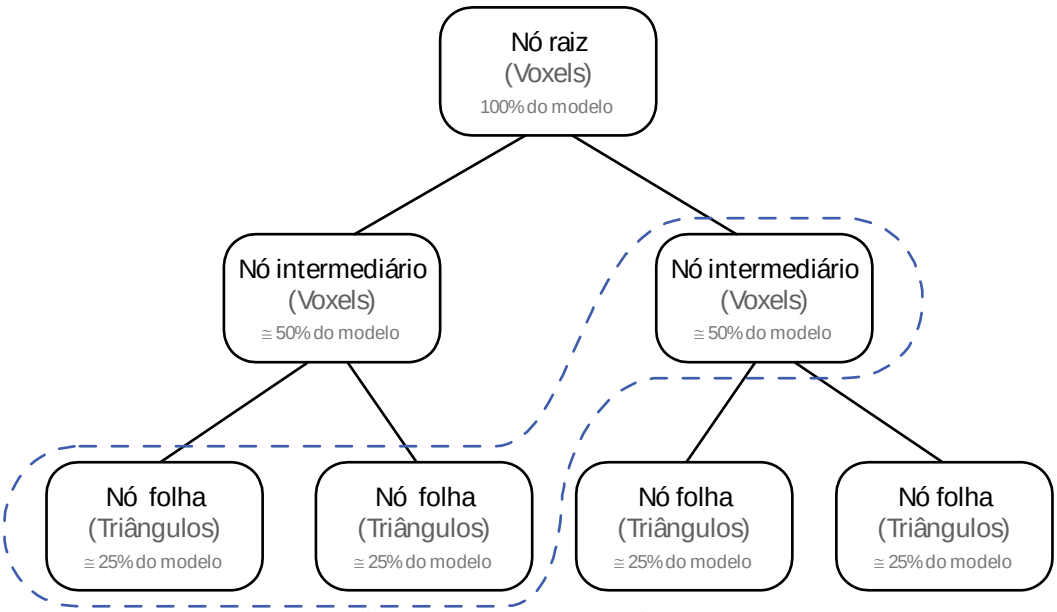

Figura 3.1 - Representação gráfica da hierarquia de níveis de detalhes de um modelo. A linha pontilhada mostra uma possível configuração de nós escolhidos para gerar uma visualização do modelo

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.

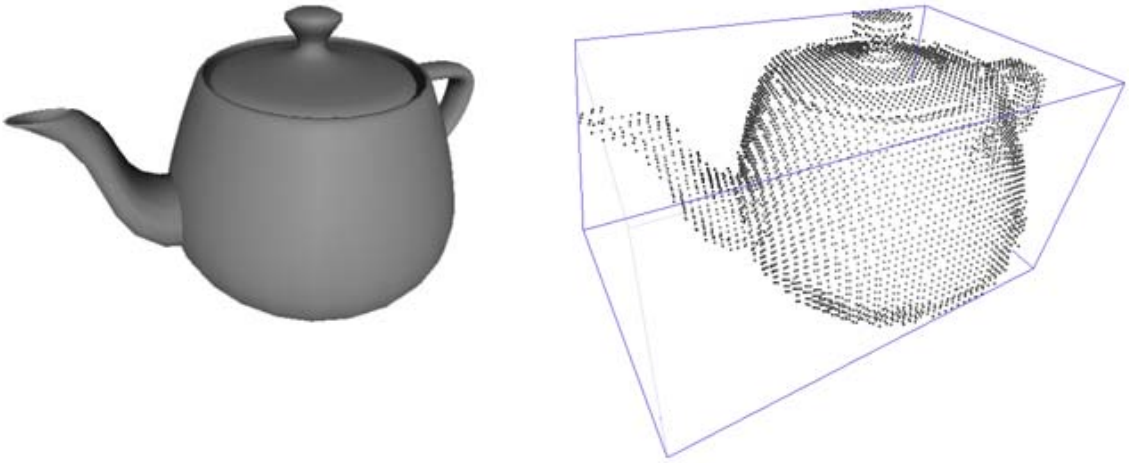

Figura 3.2 - Caixa envolvente e representação de voxel. Os voxels foram representados como pequenos pontos para notarmos sua organização.

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.

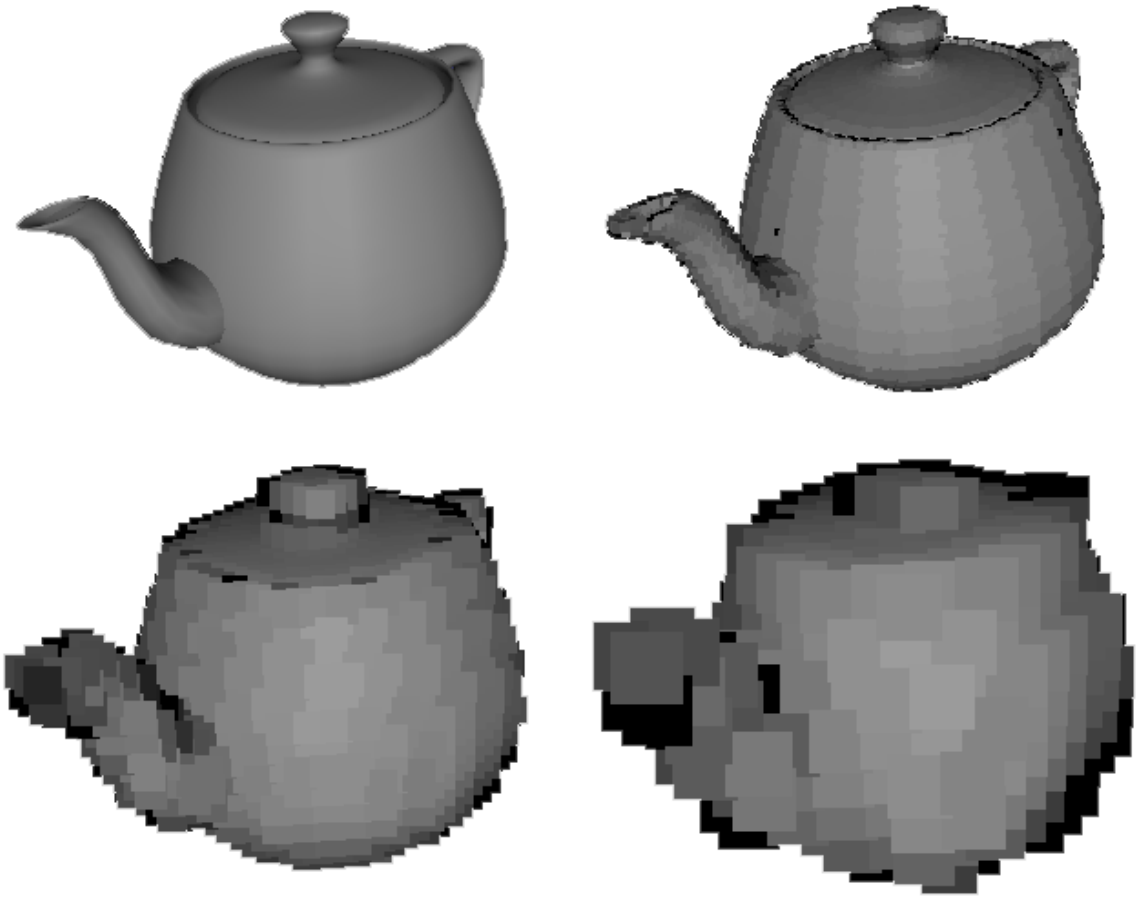

Figura 3.3 - Diferentes níveis de detalhe para representar o modelo do Teapot

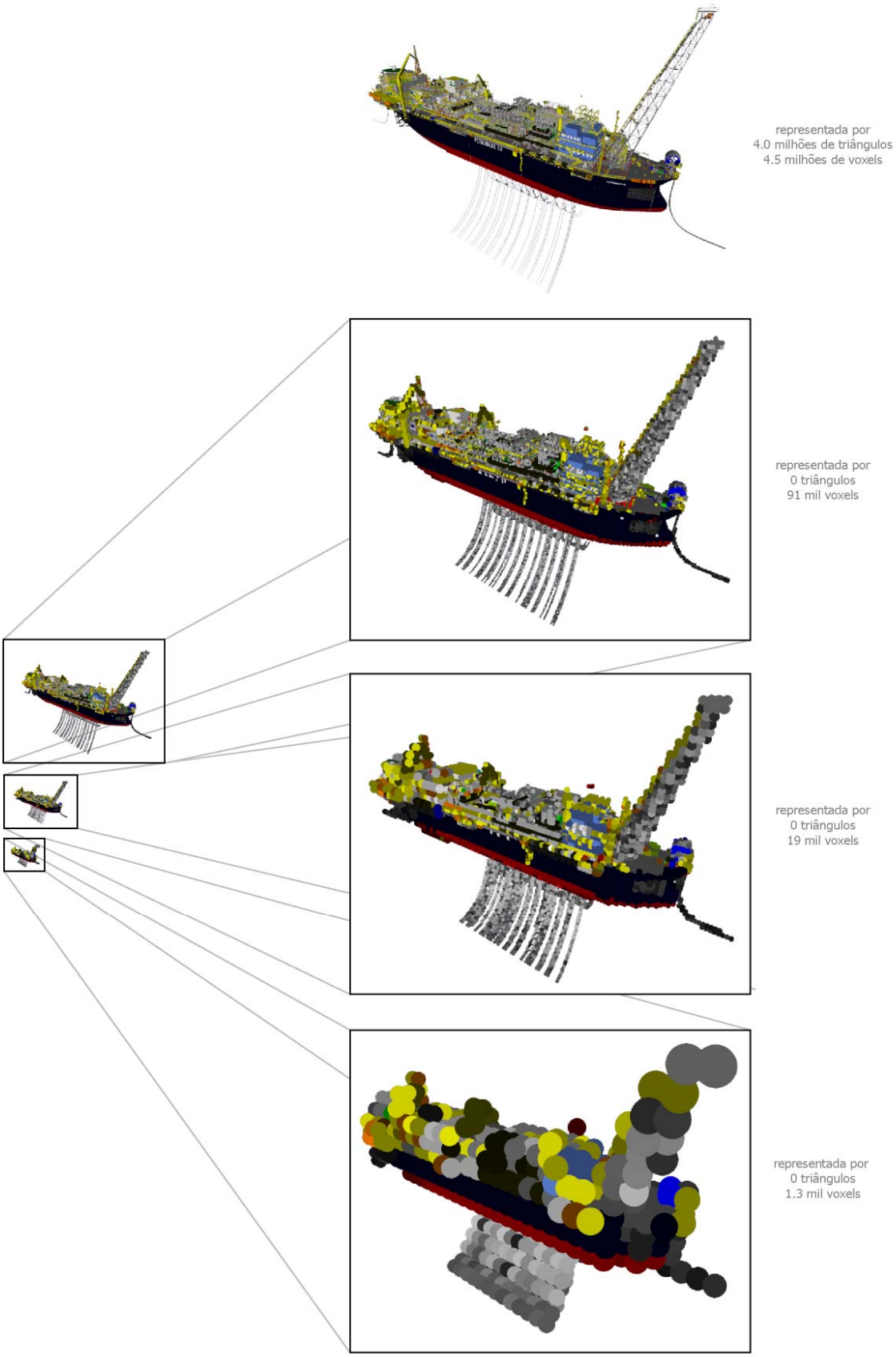

Figura 3.4 - Diferentes níveis de detalhe para representar a plataforma P-50. As imagens da esquerda mostram o tamanho que a representação usando voxels deve ter para não causar perda de qualidade visual. À direita, temos os voxels ampliados.

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.

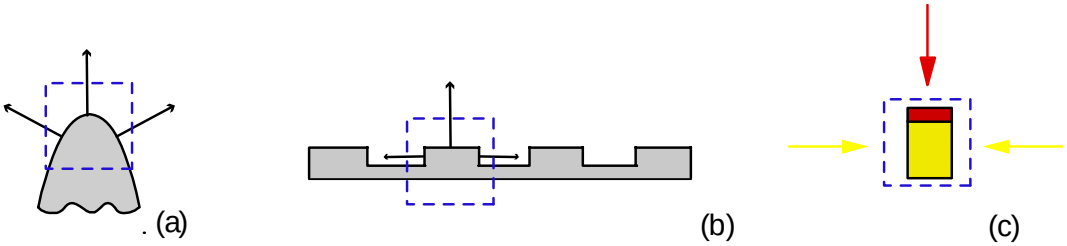

Figura 3.5 - Exemplos de voxels com atributos que variam com a direção de visualização: (a) e (b) ilustram normais variando de acordo com a posição de observação; (c) ilustra a variação de cor.

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.

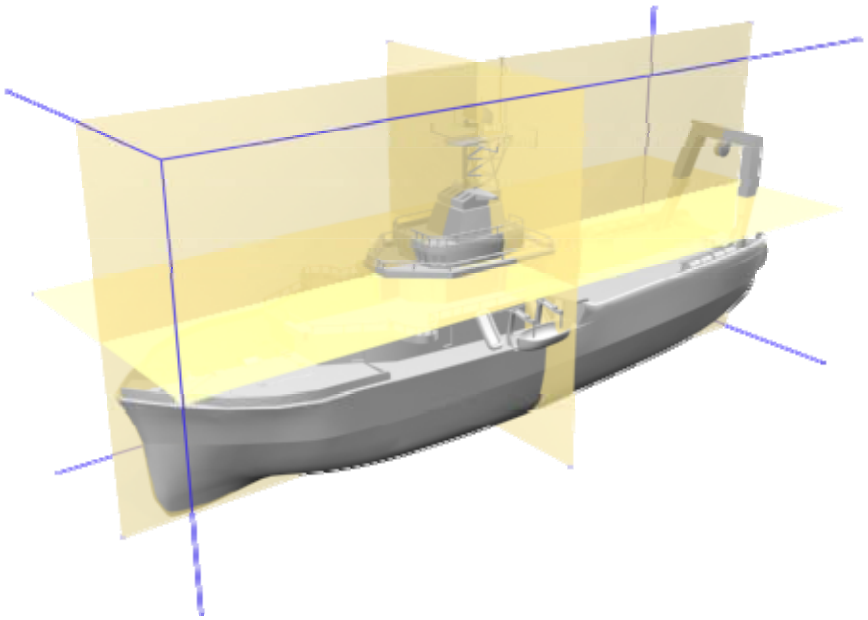

Figura 3.6 - Planos de corte de uma *octree*.

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.

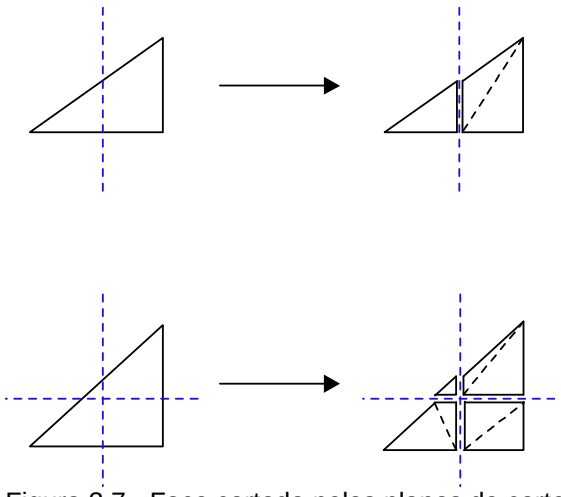

Figura 3.7 - Face cortada pelos planos de corte

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.

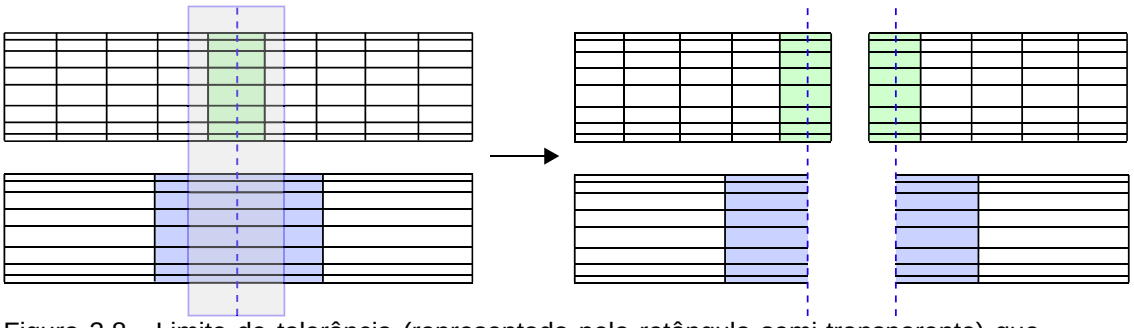

Figura 3.8 - Limite de tolerância (representado pelo retângulo semi-transparente) que determina quando as faces deverão ser duplicadas, caso do objeto de cima ou cortadas, caso do objeto de baixo.

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.

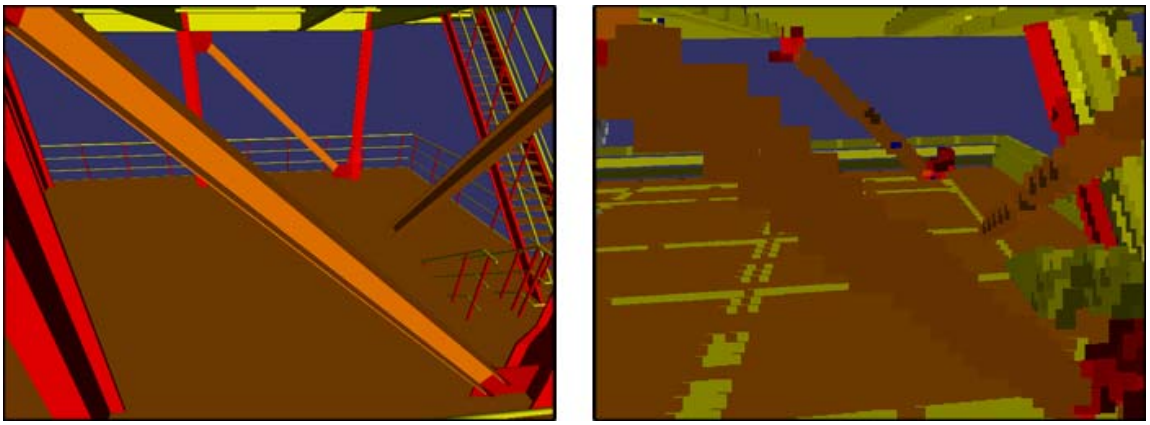

Figura 3.9 – A figura da direita mostra voxels gerados com cores erradas devido a um tratamento incorreto das superfícies ocultas. Na figura, a estrutura em amarelo que estava abaixo sob o chão dessa parte do modelo acabou sendo representada na sua versão simplificada com voxels.

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

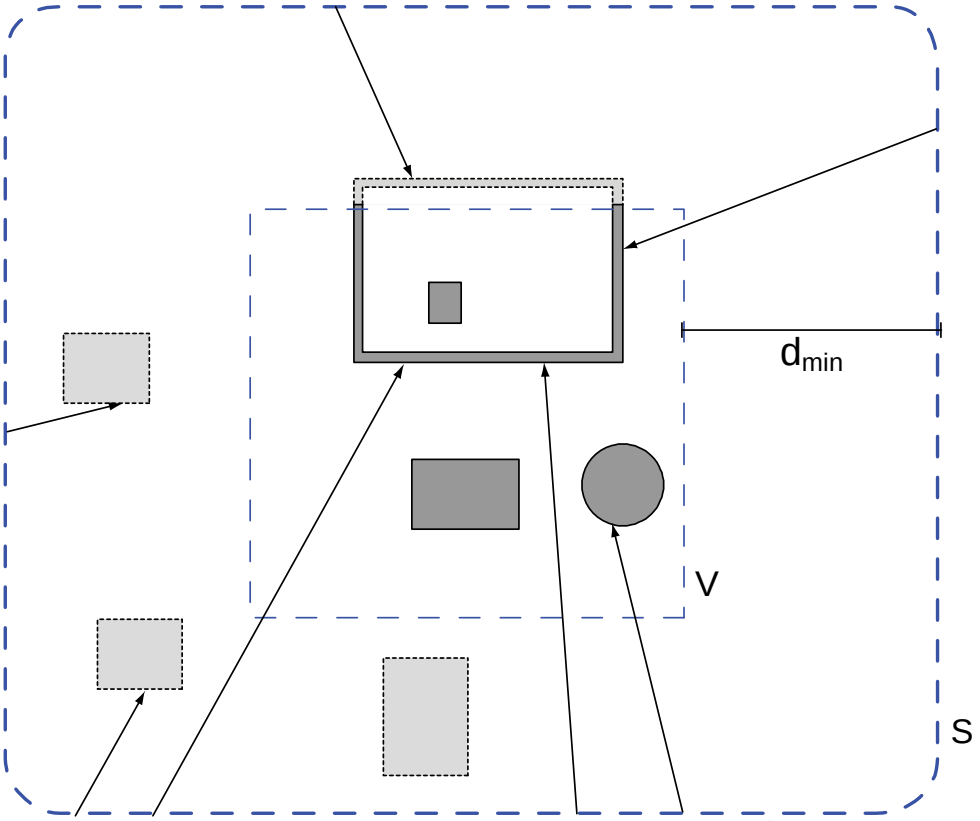

Figura 3.10 – Exemplo de configuração de um traçado de raios típico.

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.

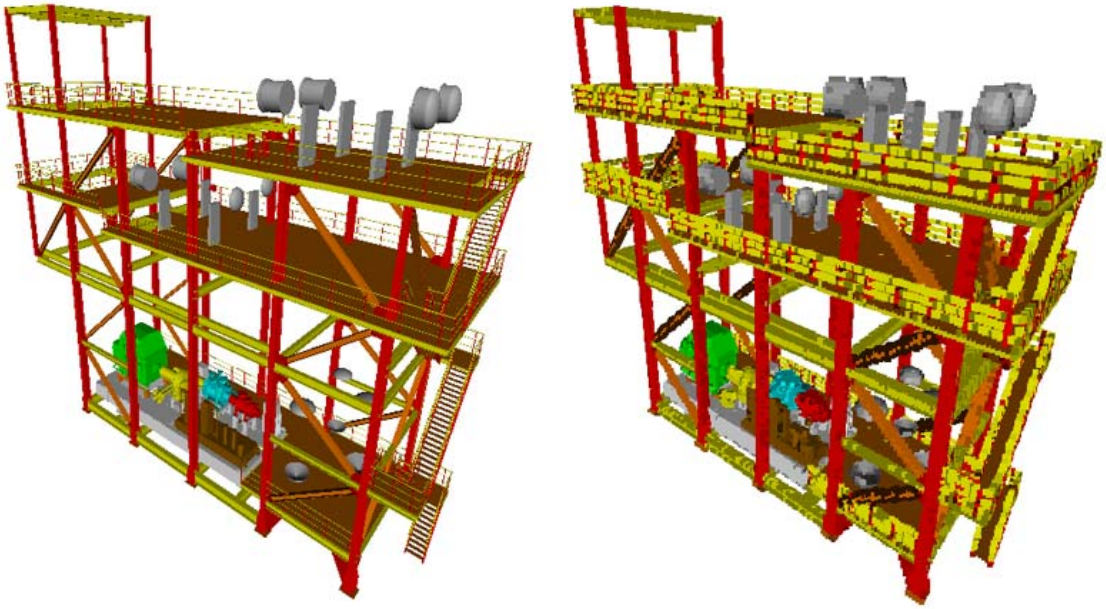

Figura 3.11 – Falhas visuais geradas por objetos muito finos sendo representados por voxels. Na figura da esquerda, representada totalmente por geometria, podemos ver que os corrimões são bem mais finos que o tamanho escolhido para os voxels, na figura da direita. Nessa figura, os voxels foram ampliados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

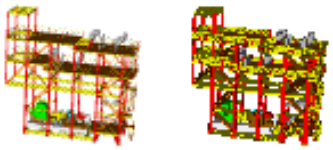

Figura 3.12 – As mesmas falhas da figura 3.11. Mesmo representadas sem ampliação, a diferença entre a representação usando geometria (esquerda) e usando voxels (direita) ainda é visível.

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.

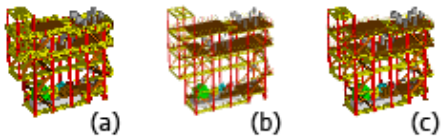

Figura 3.13 – Filtro de anti-serrilhamento no modelo SKID_ABC. Modelo representado por voxels sem nenhum filtro (a); Modelo representado apenas por geometria (b) e modelo representado por voxels com o filtro anti-serrilhamento (c)

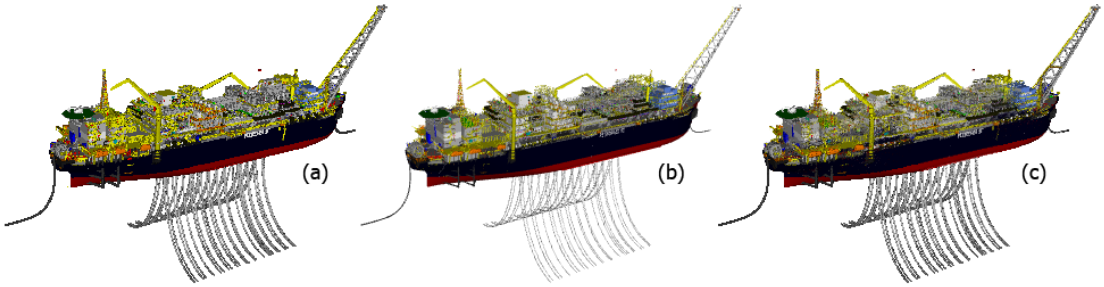

Figura 3.14 – Filtro de anti-serrilhamento no modelo P-50: Modelo representado por voxels sem nenhum filtro (a); Modelo representado apenas por geometria (b) e modelo representado por voxels com o filtro anti-serrilhamento (c)

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.

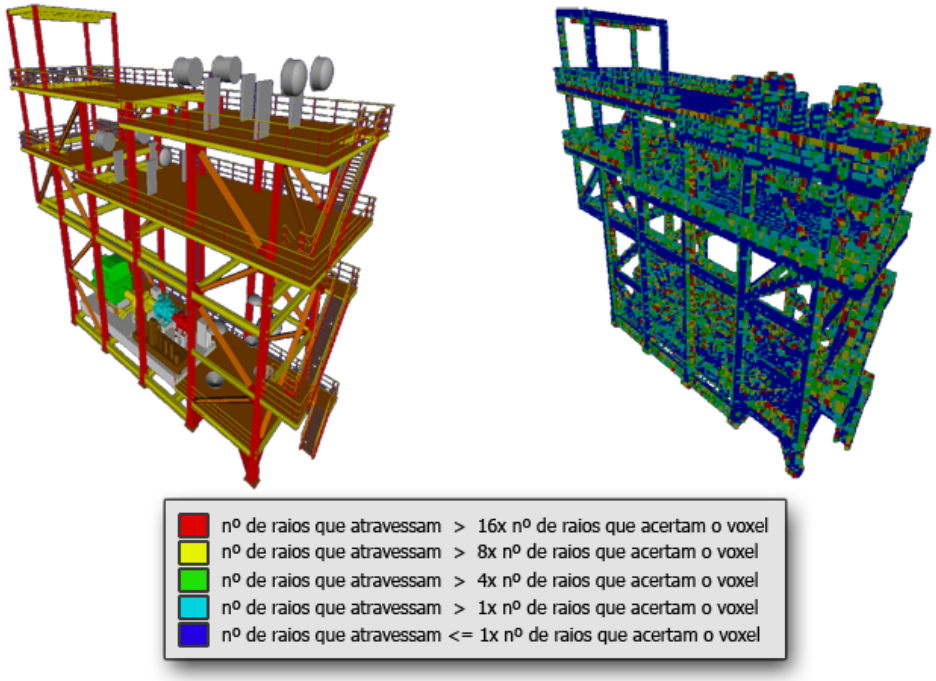

Figura 3.15 – Voxels que são atravessados por muitos raios indicam a existência de objetos que ocupam apenas uma pequena parte do volume do voxel. O modelo da direita foi codificado de forma a destacar os voxels que foram transpassados por muitos raios durante o pré-processamento.

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.

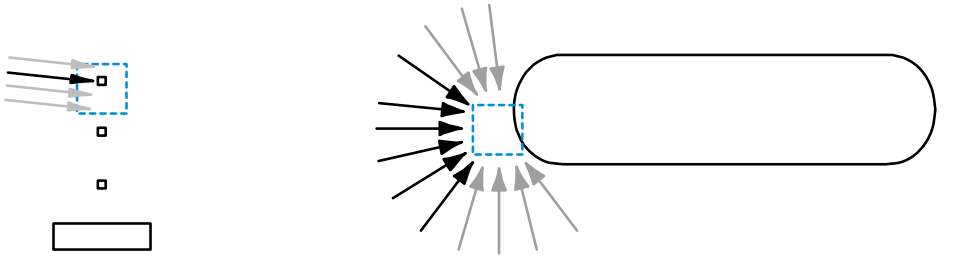

Figura 3.16 – Casos comuns de voxels transparentes encontrados durante o traçado de raios. No caso (a), o voxel poderá ser representado por um único fator de transparência, já que sua visibilidade é semelhante a partir de todas as direções. No caso (b), aplicar transparência nas direções marcadas em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

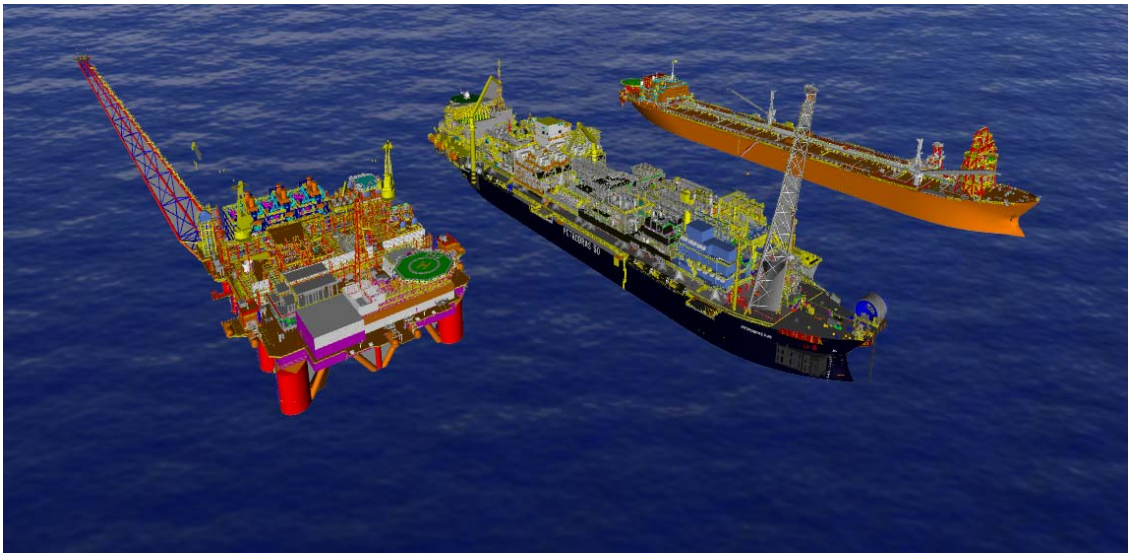

Figura 3.17 – Visualização envolvendo várias plataformas que podem ser movimentadas individualmente. Porém, seus objetos internos, não podem ser modificados, uma vez que cada plataforma possui uma hierarquia pré-definida de níveis de detalhe, ficando inviável sua modificação a cada instante.

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.