

# Técnicas de Otimização para Visualização de Modelos Massivos

*Relatório Final de Projeto*

Aluno: Paulo Ivson Netto Santos  
Matrícula: 0212017-0  
Orientador: Waldemar Celes Filho  
Período: 2006.2

# Sumário

<b>1</b>	<b>Introdução</b>	<b>1</b>
1.1	Motivação	1
1.2	Contexto Atual	2
1.3	Aplicações e Usuários	3
<b>2</b>	<b>Objetivos</b>	<b>4</b>
2.1	Estudos	4
2.2	Implementação	4
<b>3</b>	<b>Técnicas Estudadas</b>	<b>5</b>
3.1	<i>Level of Detail</i>	5
3.2	<i>Frustum Culling</i>	7
3.3	<i>Occlusion Culling</i>	10
3.4	Substituição de Geometria	14
3.5	Conclusão	18
<b>4</b>	<b>Técnicas Desenvolvidas</b>	<b>20</b>
4.1	Volumes Envolventes	20
4.1.1	AABB	20
4.1.2	OBB	21
4.1.2.1	Covariância	23
4.1.2.2	Fecho Convexo	24
4.1.2.3	Caixa Mínima Aproximada	26
4.2	Hierarquia de Agrupamento Espacial	27
4.2.1	BSP-Tree	28
4.2.2	Octree	29
4.2.3	OBB-Tree	30
4.2.4	kd-Tree	31
4.2.5	Heurísticas de Construção	32
4.2.5.1	Mediana dos Objetos	33
4.2.5.2	Mediana Espacial	34
4.2.5.3	Mediana dos Centros	34

4.2.5.4	Média dos Centros	34
4.2.5.5	Equilibrar Complexidade Geométrica	34
4.2.5.6	Minimizar SAH	35
4.3	<i>Frustum Culling</i>	36
4.3.1	Extração dos Planos do <i>Frustum</i>	36
4.3.2	Algoritmo Hierárquico	37
4.3.3	Melhorias	37
4.4	<i>Occlusion Culling</i>	38
4.4.1	<i>Occlusion Queries</i>	38
4.4.2	Algoritmo Hierárquico	40
4.4.3	Coerência Temporal	41
4.4.4	Melhorias	45
4.4.4.1	Evitar Testes Consecutivos	45
4.4.4.2	Probabilístico	46
4.4.4.3	Proximidade do Observador	46
5	<b>Especificação do Sistema</b>	47
5.1	Arquitetura	47
5.2	API Desenvolvida	49
6	<b>Testes e Resultados</b>	52
6.1	Cenas de Teste	52
6.2	Volumes Envolventes	55
6.3	<i>Occlusion Queries</i>	63
6.4	Visualização	66
6.4.1	Teapots(500,30)	68
6.4.1.1	AABB	68
6.4.1.2	OBB por Covariância	69
6.4.1.3	OBB por Aproximação da Caixa Mínima	70
6.4.2	Teapots(5000,3)	70
6.4.2.1	AABB	71
6.4.2.2	OBB por Covariância	72
6.4.2.3	OBB por Aproximação da Caixa Mínima	72

6.4.3	Teapots(5000,20)	73
6.4.3.1	AABB	74
6.4.3.2	OBB por Covariância	74
6.4.3.3	OBB por Aproximação da Caixa Mínima	75
6.4.4	P-38	75
6.4.4.1	AABB	76
6.4.4.2	OBB por Covariância	76
6.4.4.3	OBB por Aproximação da Caixa Mínima	77
6.5	Otimizações de <i>Occlusion Culling</i>	78
6.5.1	Evitar Testes Consecutivos	78
6.5.2	Probabilístico	78
6.5.3	Proximidade do Observador	79
6.6	Análise dos Resultados	79
<b>7</b>	<b>Conclusão</b>	<b>81</b>
7.1	Contribuições do Projeto	81
7.2	Trabalhos Futuros	82
<b>8</b>	<b>Referências Bibliográficas</b>	<b>83</b>

# 1 Introdução

## 1.1 Motivação

Uma das principais áreas da Computação Gráfica é a visualização interativa<sup>1</sup> de modelos tridimensionais. Cada vez mais este segmento tem desempenhado importante papel nos mais diversos setores comerciais e industriais. Exemplos de destaque incluem: aplicações de modelagem em CAD, visualização de simulações de fluidos e reconstrução de objetos reais com o uso de *scanners a laser*. Contudo, recentes avanços nas tecnologias de aquisição, modelagem e simulação de dados têm resultado em modelos geométricos cada vez maiores. Estes modelos extremamente complexos são referidos como “modelos massivos”.

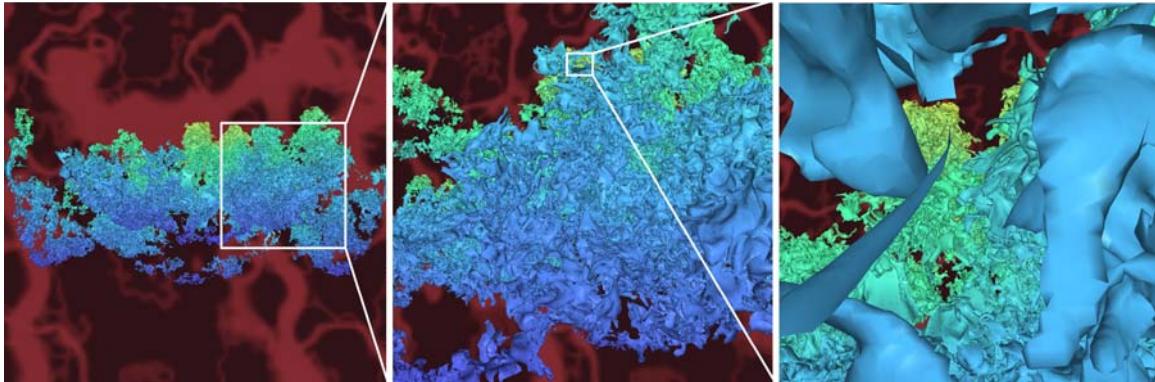
Apesar do rápido avanço no desempenho de hardwares, a visualização interativa destes modelos massivos através de abordagens clássicas tem se tornado cada vez mais difícil. Estações gráficas modernas não detém poder computacional suficiente para visualizar tamanhos volumes de dados a taxas interativas. As placas de vídeo mais rápidas atualmente são capazes de processar em torno de 1,2 bilhões de vértices por segundo. Seria possível visualizar no máximo algo por volta de 120 milhões de triângulos a uma taxa de 10 fps<sup>2</sup>. Torna-se necessário, então, o desenvolvimento de novas técnicas de otimização para melhorar o desempenho destes aplicativos 3D.



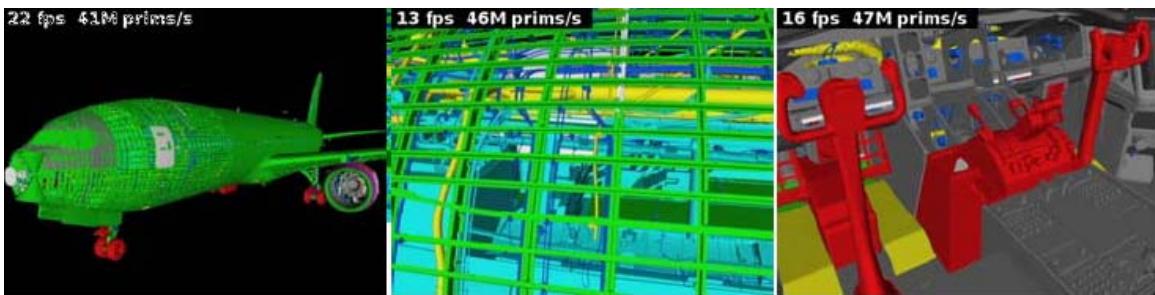
**Figura 1.1.1:** Modelo escaneado a *laser* da estátua de São Matheus: 372 milhões de triângulos [3].

<sup>1</sup> A definição que será usada corresponde a uma taxa de visualização em torno de 10 fps, ou 10 quadros por segundo.

<sup>2</sup> Caso o modelo fosse composto apenas por uma única seqüência de triângulos encadeados, o que não acontece na prática. Também desconsiderando-se o custo de processamento dos fragmentos gerados.



**Figura 1.1.2:** Isosuperfície resultante de uma simulação de fluidos de Richtmyer-Meshkov: 100 milhões de triângulos [3].



**Figura 1.1.3:** Modelo CAD de um Boeing 777: 350 milhões de triângulos [13].

## 1.2 Contexto Atual

Nos dias de hoje, a visualização de modelos tridimensionais utiliza o que se conhece por *pipeline gráfico* [1]. A cada quadro, a aplicação (CPU) envia as primitivas geométricas a serem visualizadas para o processador gráfico (GPU), que então se encarrega de processar os dados e gerar uma imagem da cena desejada.

Recentemente, as API's gráficas e os fabricantes de hardware têm provido funcionalidades para reduzir o volume de dados que devem ser transferidos a cada quadro para a GPU. Mesmo assim, a memória disponível da placa gráfica não é suficiente para acomodar tamanhos volumes de informação. Além disso, mesmo que uma parcela significativa dos dados já se encontre armazenada na placa gráfica, o grande volume de informação a ser processado ainda prejudica a taxa de visualização destes modelos.

Dessa forma, torna-se necessário reduzir a quantidade de primitivas geométricas necessárias para a correta visualização destes modelos massivos. Portanto, esta redução deve ser efetuada sem prejudicar a qualidade final da imagem.

Este constitui o principal desafio enfrentado pelas técnicas mais recentes de otimização para visualização de modelos massivos: como reduzir este volume de informação a ser enviado e processado pela placa gráfica de modo a melhorar o desempenho das aplicações, sem prejudicar a qualidade final da imagem.

### 1.3 Aplicações e Usuários

Este projeto está voltado para desenvolvedores de aplicações de visualização 3D. Neste documento, serão explorados conceitos e assuntos relacionados aos algoritmos, estruturas de dados e bibliotecas utilizadas neste tipo de aplicação.

O escopo deste projeto está limitado a aplicações que utilizam OpenGL como API de renderização<sup>3</sup>. Contudo, as técnicas estudadas e discutidas se estendem facilmente para outros ambientes de desenvolvimento.

Além disso, os paradigmas e desafios discutidos podem ser encontrados também em outros tipos de ambientes de desenvolvimento e suas correspondentes aplicações. Desse modo, o projeto constitui uma base de estudo sólida e versátil para todos que enfrentam o desafio de visualizar modelos massivos.

---

<sup>3</sup> Será utilizado o termo “renderizar” como um empréstimo lingüístico do verbo *to render* em inglês. No contexto deste projeto, o termo significará o processo de transformar uma informação geométrica em um atributo visual, ou seja, em sua imagem correspondente.

## 2 Objetivos

### 2.1 Estudos

O objetivo principal deste projeto é o estudo das varias técnicas de otimização presentes na literatura, procurando identificar suas vantagens e limitações. Apesar da visualização 3D ser uma área consolidada da Computação Gráfica, ainda restam muitas questões em aberto ou sem solução definitiva. A cada dia são desenvolvidas novas tecnologias que solucionam antigos problemas mas que também trazem consigo novas necessidades e desafios.

### 2.2 Implementação

Como forma de consolidar os estudos efetuados e contribuir para o desenvolvimento de aplicações de visualização, constitui-se como segundo objetivo deste projeto a implementação mais completa possível de uma das técnicas estudadas. Além disso, esta implementação servirá como base de testes para novas propostas e algoritmos capazes de aprimorar ainda mais a visualização de modelos massivos.

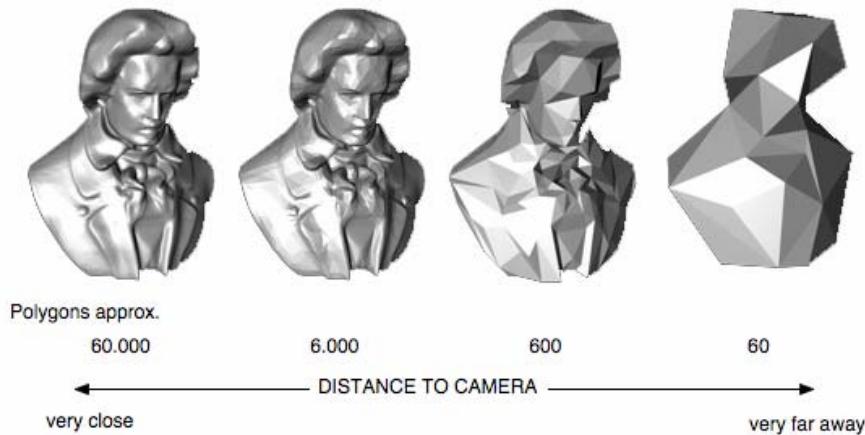
Desse modo, também será discutido neste projeto o desenvolvimento de uma biblioteca portátil que pode ser facilmente acoplada a uma aplicação de visualização já existente. Esta biblioteca tem por objetivo principal prover diferentes funcionalidades para otimizar a visualização de modelos massivos por parte da aplicação cliente. Além disso, diversas outras técnicas de processamento de dados geométricos estarão disponíveis para as mais variadas aplicações de visualização 3D.

## 3 Técnicas Estudadas

### 3.1 *Level of Detail*

Mais conhecida como *LOD* [2], esta técnica de simplificação de malhas explora o fato de que o olho humano não é capaz de perceber detalhes de objetos muito pequenos (i.e. muito distantes). Dessa forma, a idéia principal deste algoritmo é reduzir a complexidade da geometria de objetos que se encontram distantes do observador em uma dada cena, de modo a evitar o envio e processamento de detalhes geométricos que não seriam percebidos pelo usuário da aplicação.

A abordagem clássica para isso consiste em gerar para a geometria original da cena diferentes versões da mesma com resoluções cada vez menores, ou seja, com menor número de triângulos para representar o mesmo objeto. À medida em que o observador se distancia da geometria, são escolhidas versões mais simples da mesma para serem visualizadas. Alternativamente, quando o observador se aproxima de um objeto, envia-se para a placa gráfica uma versão mais complexa da geometria em questão.



**Figura 3.1.1:** Idéia central de um algoritmo de *Level of Detail*.

Esta simplificação pode ocorrer em uma etapa de pré-processamento da cena, gerando níveis pré-definidos de resolução (*LOD discreto*) ou então a simplificação da geometria pode ocorrer em tempo de execução, dependendo de fatores como a distância do observador (*LOD contínuo ou adaptativo*). Existem ainda propostas que tentam combinar os benefícios de ambas as abordagens (*geomorphing*).

A principal vantagem do *LOD discreto* é não adicionar nenhum *overhead* durante a visualização da cena. Neste momento, basta escolher o nível de resolução desejado

dentre os já computados e esta escolha pode ser feita sem nenhum impacto significativo na performance da aplicação. Uma outra vantagem desta abordagem é a seguinte: como os níveis de resolução são construídos em pré-processamento, estes podem ser armazenados previamente na placa gráfica, reduzindo assim a quantidade de geometria a ser enviada a cada quadro. Atenta-se para o fato, porém, de que modelos massivos com muitos níveis de resolução facilmente consomem toda a memória disponível na placa gráfica, o que inviabiliza esta otimização.

Além disso, a principal desvantagem do *LOD discreto* é a limitação dos níveis de resolução disponíveis. Por conta disso, podem ocorrer artefatos visuais<sup>4</sup> quando alterase a complexidade da representação geométrica de um mesmo objeto. Para evitar este problema, são estudados e desenvolvidos os mais variados algoritmos para identificar quando e como esta mudança pode ser feita sem que o usuário perceba, procurando manter a mesma qualidade da imagem final. A discretização dos níveis de detalhe também limita as escolhas de representação do modelo, muitas vezes forçando uma representação complexa demais ou simples demais para uma dada situação.

Para contornar estas dificuldades, a proposta de *LOD contínuo* procura gerar níveis de detalhe diferentes para uma mesma geometria enquanto a cena está sendo visualizada. Este processamento em *real-time* tem a vantagem de poder ser baseado em parâmetros correntes da cena, como a posição atual do observador, de forma a construir a representação mais simples possível de um objeto que ainda seja capaz de gerar sua imagem sem perda aparente de qualidade. Entretanto, este processamento se torna custoso demais para ser efetuado em grandes bases de dados, prejudicando o desempenho final da aplicação.

Uma proposta ainda mais refinada chama-se *LOD adaptativo*. Esta abordagem aplica a mesma idéia do *LOD* mas para parcelas de uma mesma geometria. Ou seja, diferentes partes de uma mesma geometria são renderizadas com diferentes níveis de detalhe. Estes são obtidos em tempo de execução, de forma similar ao *LOD contínuo*. O *LOD adaptativo* é especialmente adequado para a visualização de terrenos extensos, por exemplo.

A utilização de *geomorphing* procura aliar a vantagem do pré-processamento do *LOD discreto* com a precisão do *LOD contínuo*, reduzindo o *overhead* em tempo de execução da aplicação. Entretanto, muitas dificuldades surgem na escolha e construção da combinação dos diferentes níveis de detalhe pré-computados. Diferentes propostas encontram soluções para cenas ainda muito particulares.

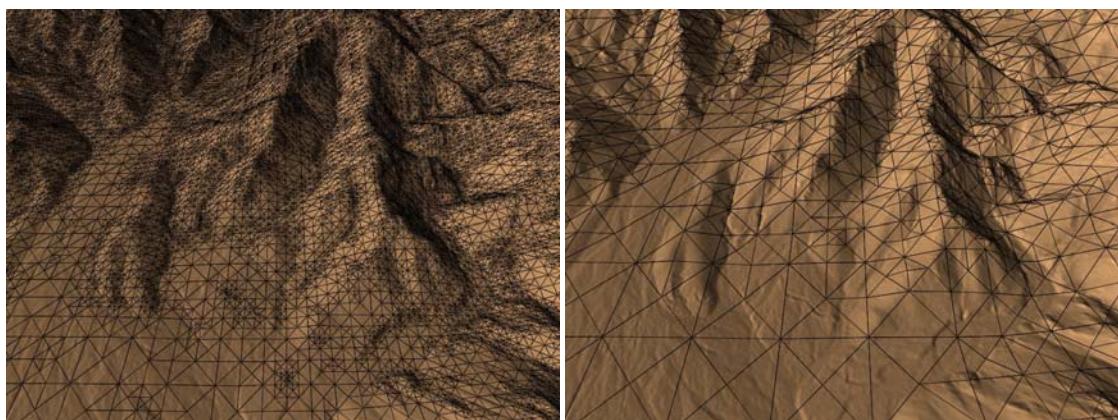
---

<sup>4</sup> Falhas na imagem gerada, geralmente por conta de uma representação pobre da geometria, ou da deformação abrupta da mesma.

De qualquer forma, a técnica de *Level of Detail* é capaz de reduzir consideravelmente volume total de geometria enviado para a placa gráfica, melhorando o desempenho da aplicação. Quando utilizada com sucesso, é capaz de fazê-lo sem prejudicar a qualidade final da imagem.

Devido às dificuldades mencionadas, esta técnica é tipicamente utilizada para visualização de terrenos e superfícies extensas e contínuas, onde grande parte da geometria encontra-se distante do observador (i.e. próximo ao horizonte) e apenas uma pequena parcela do modelo está próxima. Além disso, estes tipos de superfícies são mais fáceis de serem processadas, particionadas e combinadas durante a construção dos diferentes níveis de detalhes.

Ainda estão sendo estudados algoritmos e estratégias que procuram agrupar muitos objetos em *clusters*, para depois simplificá-los corretamente. Chamada de *Quick-VDR* [3], tal abordagem não somente resolve as mesmas questões mencionadas anteriormente, mas se destaca como uma futura possibilidade de otimização para modelos massivos em geral.



**Figura 3.1.2:** Níveis de detalhe aplicados adaptativamente a uma superfície de terreno.

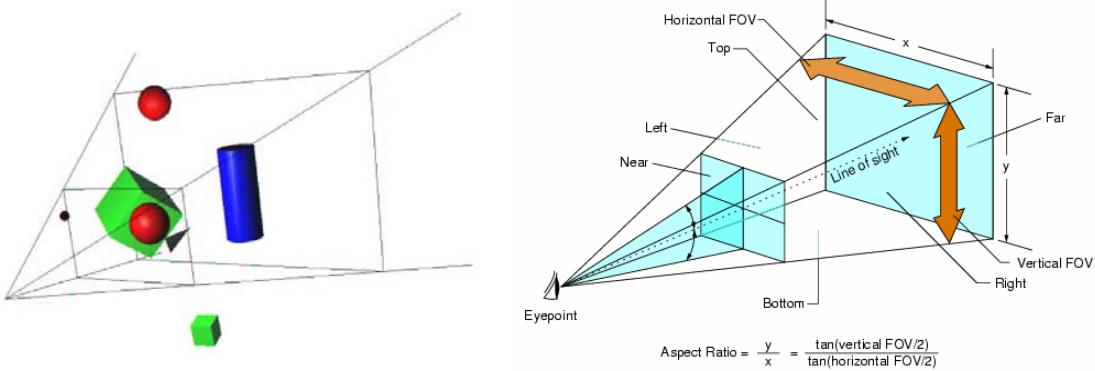
### 3.2 *Frustum Culling*

Em geral, quando o usuário observa uma cena tridimensional, dificilmente ele estará vendo todos os objetos que constituem esta cena. Estes objetos não-visíveis<sup>5</sup> ocorrem principalmente por dois motivos: ou eles se encontram fora do volume de visão<sup>6</sup>, ou eles se encontram escondidos por detrás de objetos maiores ou mais próximos do observador.

<sup>5</sup> Será utilizado o termo não-visível para objetos que não contribuem para a imagem final de uma cena.

<sup>6</sup> Conhecido por *view frustum* ou *frustum* de visão, inclui todos os objetos da cena que serão renderizados pela placa gráfica.

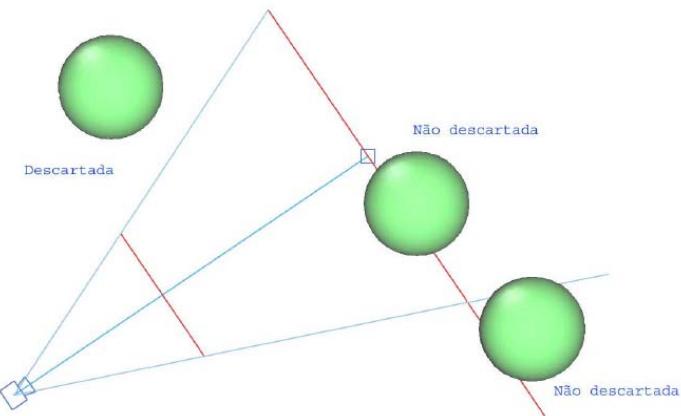
A técnica conhecida como *Frustum Culling* [15] tem por objetivo identificar os objetos que se encontram fora do volume de visão e que, por consequência, não contribuirão para a imagem final da cena. Muitos algoritmos e propostas já se encontram consolidados na literatura. Todos compartilham uma mesma abordagem, descrita a seguir.



**Figura 3.2.1:** *Frustum* de visão para uma vista em perspectiva.

Primeiramente, constrói-se uma representação que preserve a forma do objeto, incluindo totalmente seu volume mas simplificando sua complexidade geométrica. Diferentes abordagens utilizam diferentes tipos de volumes envolventes, cada um com suas vantagens e desvantagens.

Estes volumes são então testados contra o *frustum* de visão e, caso se encontrem totalmente fora do mesmo, evita-se o envio da geometria mais complexa em seu interior. Este teste normalmente é feito calculando-se a interseção destes volumes com os 6 planos que definem o *frustum*: *near*, *far*, *left*, *right*, *top* e *bottom*. A interseção contra um plano pode indicar se o volume se encontra no lado positivo ou negativo do plano em questão. Ao adotar-se a convenção de que todos os planos do *frustum* possuem suas normais voltadas para dentro do volume de visão, pode-se utilizar o resultado destes testes para descartar ou não um dado volume envolvente.



**Figura 3.2.2:** Descarte de esferas envolventes contra o volume de visão [16].

Diferentes escolhas de volumes envolventes proporcionam um ajuste mais preciso da geometria em troca de um teste de interseção mais custoso. Além disso, a construção de volumes bem ajustados à geometria envolvida constitui uma área à parte de pesquisa e otimização, envolvendo conhecimentos variados de Geometria Computacional. Os volumes envolventes mais utilizados na literatura são (em ordem crescente de custo de interseção e em ordem crescente de qualidade de ajuste): esfera envolvente, AABB<sup>7</sup> e OBB<sup>8</sup>.

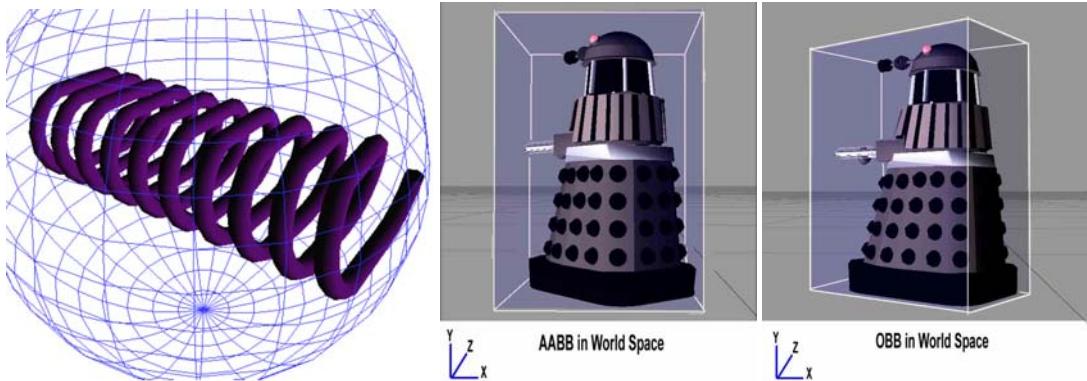


Figura 3.2.3: Volumes envolventes mais comuns (da esq. para a dir.): esfera envolvente, AABB e OBB.

Para evitar a necessidade de teste de todos os volumes de todos os objetos da cena, constrói-se em pré-processamento uma hierarquia de volumes envolventes. Dessa forma, o descarte de um volume maior acarreta no descarte de todos os volumes menores envolvidos pelo primeiro, sem necessidade de nenhum teste adicional. Esta hierarquia é construída através de um algoritmo de agrupamento espacial.

Como forma de otimizar ainda mais o descarte contra o volume de visão, procura-se utilizar a coerência temporal entre um quadro e outro. Esta técnica baseia-se no fato de que se um volume foi descartado contra um dos 6 planos do *frustum* no quadro anterior, existe uma chance muito alta do mesmo volume ser descartado novamente por este mesmo plano.

Isto porque na grande maioria das vezes os movimentos do observador são suaves e geram pequenas transições na câmera de um quadro para o outro. Desse modo, testa-se primeiro contra este plano em particular. Caso este teste indique que o volume agora se encontra pelo menos parcialmente dentro do *frustum*, torna-se necessário avaliá-lo contra os demais planos. Na maioria das vezes, este primeiro teste irá indicar novamente que o volume continua não-visível.

<sup>7</sup> *Axis-Aligned Bounding Box*, ou caixa orientada aos eixos. Estes são os eixos do sistema coordenado *x*, *y* e *z*.

<sup>8</sup> *Oriented Bounding Box*, ou caixa orientada. Similar à AABB, porém sua orientação não é fixa de acordo com os eixos *x*, *y* e *z*. A caixa pode então seguir a orientação da geometria envolvida, por exemplo, proporcionando um melhor ajuste.

Outra otimização no descarte contra o volume de visão utiliza a hierarquia de volumes envolventes para evitar testes desnecessários, utilizando uma forma de coerência espacial. Supõe-se que um volume envolvente maior foi testado contra um dado plano e, segundo o mesmo, encontra-se totalmente no lado positivo. Pela construção da hierarquia de volumes, pode-se afirmar com certeza que os volumes menores descendentes do volume maior também serão classificados da mesma forma. Portanto, para um dado volume envolvente, não é necessário testá-lo contra os mesmos planos do volume de visão para os quais seus ancestrais na hierarquia já foram classificados como no lado positivo dos planos.

Existem ainda duas técnicas para otimizar ainda mais o descarte contra o volume de visão. Uma delas partitiona o *frustum* ao meio ao longo de cada uma de suas extensões. Com isto, obtém-se oito regiões chamadas *octantes*. Ao determinar-se em qual *octante* o volume envolvente se encontra, basta testá-lo contra três planos exteriores, evitando assim o teste contra todos os planos do *frustum*.

Uma última otimização mais específica assume que os movimentos do observador são limitados a simples translações e rotações. Neste caso, para cada volume envolvente utiliza-se informações do quadro anterior para determinar exatamente os únicos planos que poderão ainda descartar o dado volume. Esta última técnica é capaz de obter melhorias significativas no desempenho dos testes de descarte.

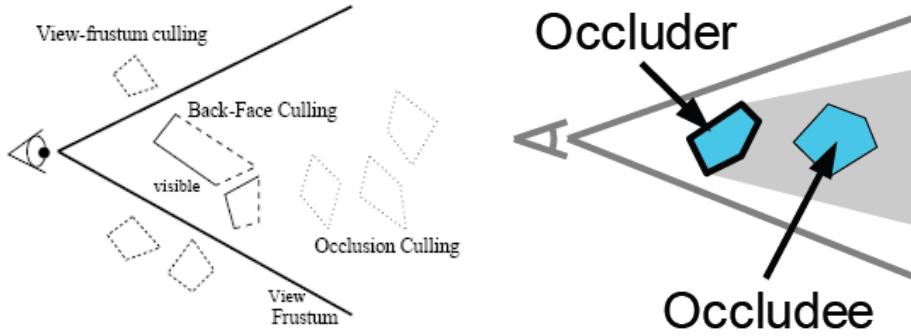
A técnica de *Frustum Culling* é capaz de eliminar rapidamente muitos objetos não-visíveis, desde que os mesmos se encontrem fora do volume de visão. Portanto, sua aplicação se limita a situações específicas durante a visualização de uma cena e, por si só, não é capaz de garantir um bom desempenho da aplicação no caso geral.

Caso o observador esteja visualizando uma grande parcela da cena, muitos objetos estarão dentro do *frustum* e ainda restará um volume enorme de processamento para a placa gráfica. Para estes casos, procura-se eliminar o segundo tipo de objetos não-visíveis mencionado: aqueles que se encontram ocultos por detrás de outros.

### 3.3 *Occlusion Culling*

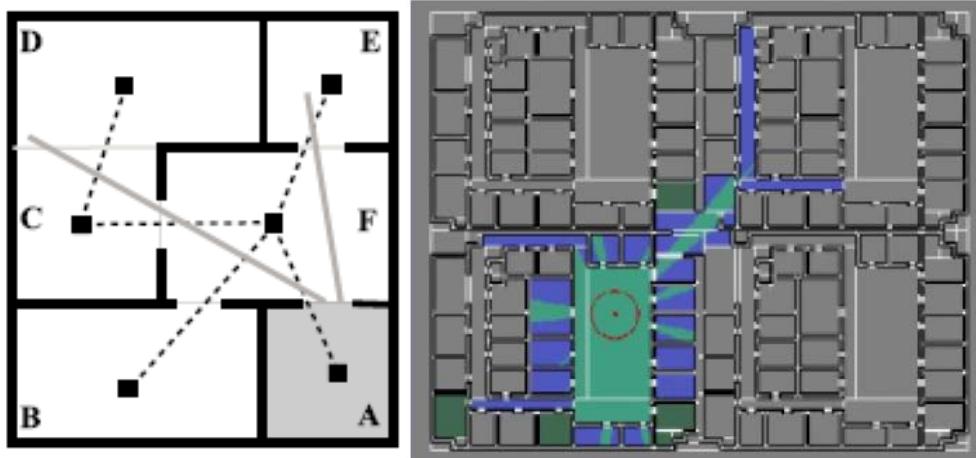
Em uma cena composta por múltiplos objetos de diferentes tamanhos, é muito comum uma parcela significativa deles estar oculta por detrás dos objetos mais próximos ao observador, principalmente se estes forem de tamanho maior. Ao identificar estes objetos ocultos, evita-se seu envio para o processador gráfico e, dessa forma, não será necessário o processamento de geometrias que não contribuiriam para a imagem final da cena.

Esta técnica, chamada de *Occlusion Culling*, constitui um tipo de *Visibility Culling* [4]. Apesar da idéia parecer tão simples quanto o caso do *Frustum Culling*, determinar os objetos ocultos de uma cena é um desafio ainda sem solução definitiva, existindo muitas propostas sendo pesquisadas até hoje.



**Figura 3.3.1:** À esquerda, diferentes técnicas de *Visibility Culling*. À direita, um objeto não-visível e seu oclusor.

As técnicas pioneiras, como *Cells*, *Portals* e *PVS* [5,6,7], eram voltadas exclusivamente para modelos de arquitetura e caminhamentos internos de aposentos bem-definidos. Sua aplicação bem-sucedida para cenas gerais nunca foi comprovada. Estas técnicas são chamadas técnicas *offline*, pois a determinação da visibilidade dos objetos em uma cena é feita em pré-processamento, sendo então construída uma estrutura capaz de indicar, em tempo de visualização, quais objetos estão visíveis dada a posição do observador na cena. Como esta classificação é feita para cada subdivisão espacial, estas técnicas também são classificadas como *from-region*.



**Figura 3.3.2:** À esquerda, um esquemático de *Cells* e *Portals*. À direita, o *PVS* de uma dada célula.

As demais técnicas de *Occlusion Culling* estudadas são aplicadas *online*, ou seja, a visibilidade dos objetos é determinada enquanto é efetuada a visualização da cena. Para isto, utiliza-se parâmetros como a posição e orientação do observador para determinar

quais objetos se encontram visíveis em um dado momento. Por conta disso, estas técnicas também são classificadas como *from-point*, pois dependem do ponto de vista corrente em que se observa a cena.

Uma proposta chamada *hierarquical z-buffer* [8] estende o *z-buffer* de uma aplicação para uma pirâmide de níveis discretos, contra os quais pode-se determinar superfícies ocultas mais rapidamente. Entretanto a proposta necessita de suporte em hardware para se tornar viável para grandes modelos.

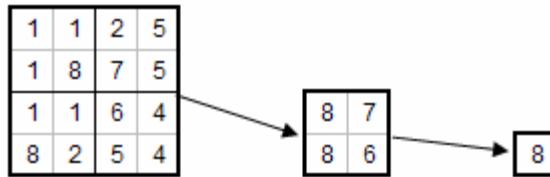


Figura 3.3.3: Uma hierarquia simples de *z-buffers*, os números são os valores em *z* de cada *pixel*.

Outra técnica conhecida por *shadow frustra* [9], utiliza projeções de sombra de objetos uns nos outros. A idéia é simular o observador como um foco de luz e calcular quais objetos estão iluminados (visíveis) e quais estão em sombra (não-visíveis). Porém, como este cálculo é custoso, a técnica é mais eficiente para modelos com um pequeno número de grandes objetos oclusores<sup>9</sup>, o que nem sempre é o caso. Além disso, identificar estes “bons oclusores” é uma tarefa que até o momento só foi parcialmente resolvida por intermédio de variadas heurísticas que levam a “constantes mágicas” intrínsecas a cada cena.

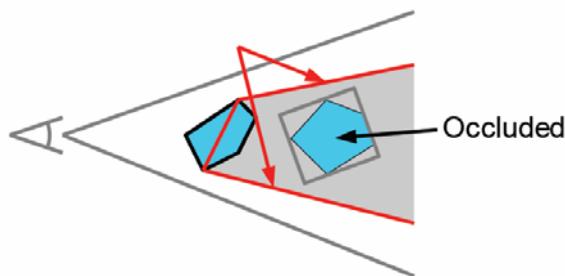
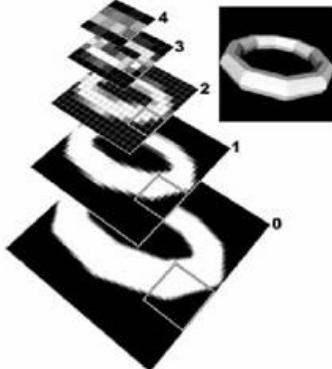


Figura 3.3.4: *Shadow-frusta*: definem-se múltiplos planos de sombra para cada “bom oclusor”.

Uma técnica desenvolvida que utiliza *ray casting* [10] também possui um alto custo que a limita a poucos e “bons” oclusores. Outro algoritmo gera *occlusion maps* [11] para determinar se alguma parcela de um objeto é visível de um determinado ponto de vista, mas também depende da identificação prévia de potenciais oclusores em uma

<sup>9</sup> Um objeto freqüentemente visível que encobre totalmente um outro objeto, tornando este último não-visível.

cena. Além disso, o processamento de visibilidade é feito de forma seqüencial, podendo gerar atrasos entre a CPU e a GPU. Por consequência, sua aplicação se limita a algumas cenas específicas e dificilmente poderá ser adaptado para uso geral.



**Figura 3.3.5:** Occlusion Maps hierárquicos.

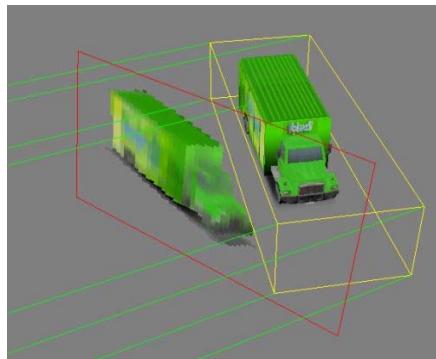
Placas gráficas mais recentes têm suportado testes de oclusão em hardware, chamados *occlusion queries*. Tais testes procuram aproveitar o potencial de computação do hardware gráfico para calcular a visibilidade de um ou mais objetos. O processo é simples: a aplicação é capaz de indagar quantos *pixels* foram gerados durante a renderização de determinados objetos.

Para utilizar esta funcionalidade, explora-se mais uma vez o uso de volumes envolventes. Antes de se enviar um objeto razoavelmente complexo para o processador gráfico, primeiro envia-se seu volume envolvente (de complexidade inferior). Desabilita-se a escrita no *color buffer* e *depth buffer* para que a renderização do volume não prejudique a visualização dos demais objetos da cena. Caso o volume gere um número de *pixels* maior do que um valor pré-determinado, considera-se o objeto visível e em seguida envia-se o mesmo para renderização normal. Ao explorar uma hierarquia de volumes envolventes, define-se a proposta de um algoritmo conhecido por *Coherent Hierarchical Culling*[12].

Porém, existem muitas questões de difícil solução para este algoritmo: balancear o *trade-off* entre testar um volume e renderizar diretamente um objeto, balancear a quantidade de *queries* para não sobrecarregar o processador gráfico, evitar que a CPU trave esperando o resultado de uma *query*, etc. Apesar de tudo, esta técnica é explorada com sucesso em outras soluções de otimização como uma forma adicional de melhorar o desempenho da visualização 3D [3, 13].

### 3.4 Substituição de Geometria

Em uma tentativa de reduzir a quantidade de informações geométricas enviadas para a placa de vídeo, pode-se substituir uma parcela da geometria de uma cena por representações que produzam o mesmo efeito visual. Propostas incluem o uso de impostores [18] e renderização volumétrica [14]. O primeiro tende a produzir muitas falhas na imagem final por ser difícil a tarefa de gerar e posicionar corretamente o impostor. O segundo, por sua vez, exige texturas enormes que muitas vezes estão além da capacidade da placa gráfica, não se adaptando ao caso de modelos massivos.



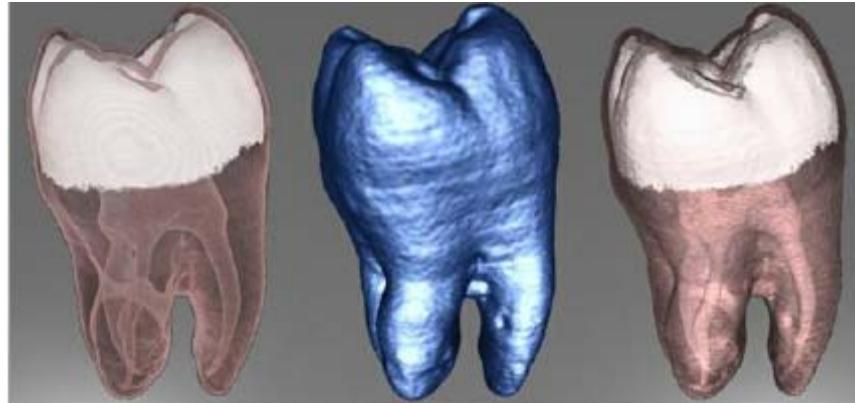
**Figura 3.4.1:** Geração de um impostor de baixa resolução.



**Figura 3.4.2:** A grandes distâncias, pode-se substituir a geometria por seu impostor sem diferença perceptível na qualidade da imagem.



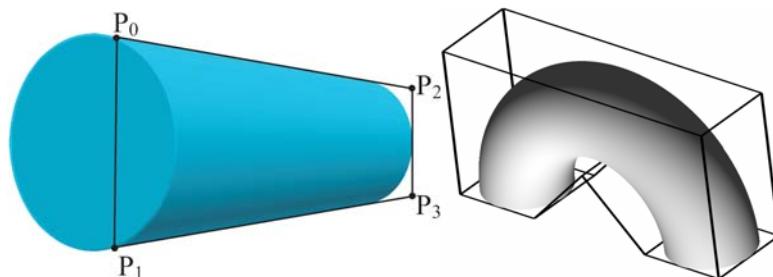
**Figura 3.4.3:** Comparação da geometria com o impostor gerado.



**Figura 3.4.4:** Exemplos de renderização volumétrica.

Outras técnicas recentes procuram explorar o uso cada vez mais freqüente do *pipeline* programável das placas gráficas. O uso de *shaders*<sup>10</sup> não somente possibilita os mais variados efeitos visuais em uma cena, mas também pode ser utilizado para explorar o hardware otimizado da placa gráfica. Além de ser paralelizado, o hardware gráfico é voltado para operações específicas de Álgebra Linear e operações aritméticas em geral.

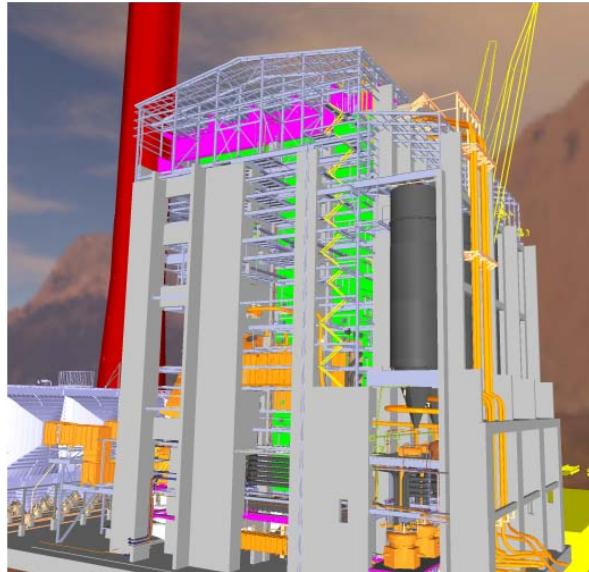
A técnica conhecida por *GPU Primitives* [17] utiliza *shaders* para renderizar certos tipos de objetos de maneira mais eficiente utilizando um volume menor de informação geométrica. Para isso, é necessário identificar na cena objetos chamados parametrizáveis, ou seja, que possuam uma forma paramétrica. Exemplos incluem cilindros e torus. Estes objetos são especialmente comuns em modelos industriais do tipo CAD: cerca de 90% de um modelo conhecido por *PowerPlant*, *benchmark* comum na comunidade de modelos massivos, é composto unicamente por objetos paramétricos.



**Figura 3.4.5:** Cilindro e torus paramétricos.

---

<sup>10</sup> Programa que pode ser carregado e executado na placa gráfica, no lugar do processamento tradicional de vértices ou de fragmentos.



**Figura 3.4.6:** Imagem da *Powerplant*: 13 milhões de triângulos.

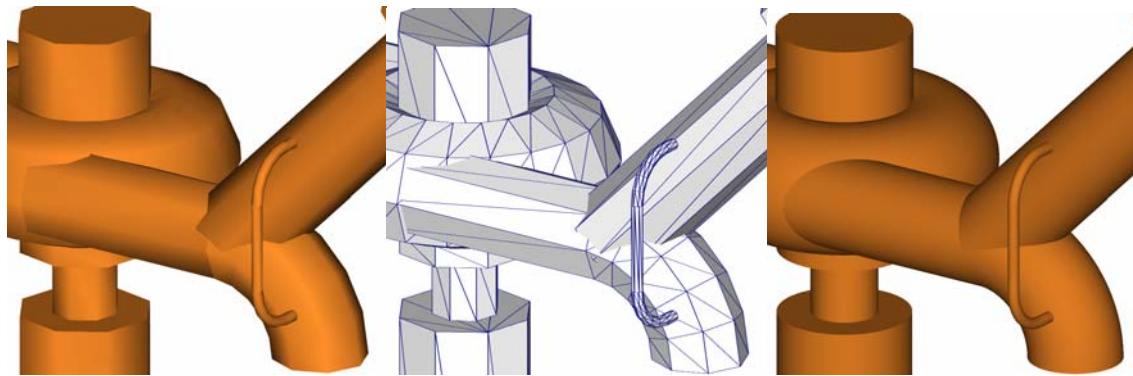
Geralmente, estas primitivas podem ser identificadas e exportadas em sua forma paramétrica através de um suporte na própria aplicação CAD utilizada durante a confecção do modelo. Contudo, caso estas geometrias sejam fornecidas apenas na sua forma já triangulada, torna-se necessário o uso de técnicas de engenharia reversa para que se possa identificar, dentre todas as malhas da cena, aquelas que correspondem a primitivas parametrizáveis.

Ao substituir um objeto por sua forma paramétrica, reduz-se a quantidade de informação que precisa ser enviada para a placa gráfica: um cilindro paramétrico pode ser visualizado com apenas quatro vértices, por exemplo. Informações adicionais por primitiva podem ser pré-computadas e armazenadas em texturas para acelerar seu acesso no hardware gráfico.

Um *shader* especializado e otimizado para cada tipo de primitiva é responsável por traduzir a informação paramétrica em fragmentos que mais tarde comporão os *pixels* da imagem final do objeto. Dessa forma, ao substituir uma grande quantidade de objetos de uma cena por sua forma paramétrica, é possível reduzir consideravelmente o volume de dados a ser visualizado, possibilitando uma maior performance para uma mesma cena.

Uma vantagem adicional desta técnica é sua qualidade visual. Normalmente, para um objeto ter muitos detalhes é necessário representá-lo com muitos vértices, o que prejudica o desempenho da aplicação de visualização. A partir de um certo momento, a resolução do objeto é tão grande que o olho humano não consegue mais perceber seu nível de detalhe. Ainda assim, caso o observador se aproxime ainda mais do objeto, eventualmente ele poderá observar novamente os vértices que definem seu contorno.

Mas no caso das primitivas paramétricas, o nível de detalhe de um objeto é o maior possível: cada *pixel* do objeto final é gerado proceduralmente dentro da placa gráfica. Em outras palavras, enquanto um cilindro paramétrico necessita de apenas quatro vértices para ser visualizado, seu nível de detalhe corresponde efetivamente a um cilindro de infinitos vértices: não importa o quanto próximo o observador esteja do objeto, ele sempre verá o seu contorno da forma mais detalhada possível.



**Figura 3.4.5:** À esquerda, um modelo renderizado com triângulos. No meio, sua respectiva visualização em *flat-shading*. À direita, o mesmo modelo visualizado através da técnica de *GPU Primitives*.

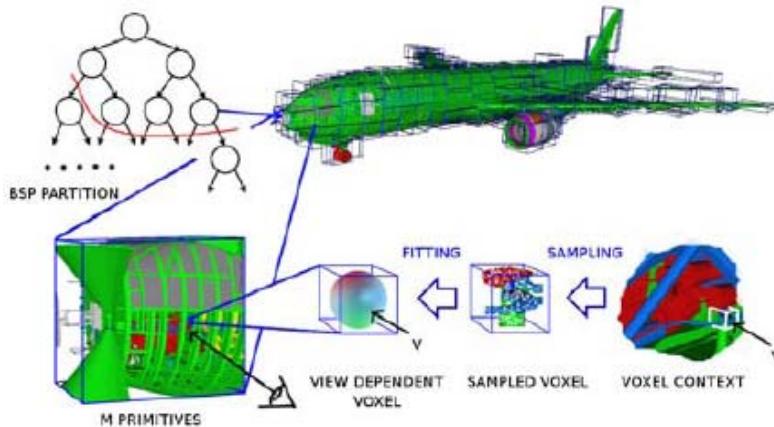
É importante destacar que esta técnica pode ser facilmente integrada à renderização normal de outras geometrias em uma mesma cena. Isto porque os fragmentos gerados pelos *shaders* possuem informação correta de z. Dessa forma, o *z-buffer* se encarrega de combinar precisamente a visualização das primitivas paramétricas e outras formas geométricas renderizadas por triângulos.

Finalmente, estudou-se uma última técnica de substituição de geometria que parece ser uma das mais promissoras para o caso de modelos massivos em geral. É conhecida por *Far Voxels* [13]. Sua ideia central explora o mesmo fato que já foi explorado pelo *Level of Detail* anteriormente: geometrias distantes não necessitam de muitos detalhes para serem corretamente visualizadas.

Contudo, no caso do *Far Voxels*, vai-se além da simplificação de malhas: substitui-se totalmente geometrias muito pequenas por uma representação em forma de *voxel* da mesma. Na prática, isso significa que geometrias muito pequenas (i.e. que contribuem com poucos *pixels*), são substituídas por pontos (*voxels*) que aproximam a contribuição daquela geometria na imagem final. Obviamente, para a placa gráfica, processar um ponto é muito mais rápido do que processar uma malha complexa ou um grande aglomerado de objetos.

Como em uma cena geral muitos objetos são naturalmente pequenos ou se encontram razoavelmente distantes, uma parte significativa do modelo é substituído por

sua representação simplificada. Dessa forma, consegue-se taxas interativas de visualização mesmo para modelos constituídos por milhões de objetos.



**Figura 3.4.6:** Esquemático da proposta de *Far Voxels* [13].

Uma das desvantagens do método proposto é seu pré-processamento, que leva cerca de 4,5 horas para processar o modelo do Boeing 777 em um *cluster* de 16 máquinas dual-core de alto desempenho. Assim como o caso do *Quick-VDR* para *Level of Detail*, a técnica de *Far Voxels* parece ser o melhor resultado até o momento no que diz respeito à substituição de geometria geral e eficiente.

Vale ressaltar, entretanto, que estas duas técnicas proeminentes também fazem uso de duas outras otimizações já mencionadas: *Frustum Culling* e *Coherent Hierarchical Occlusion Culling*.

### 3.5 Conclusão

A variedade de técnicas de otimização estudadas já é um forte indício da importância cada vez maior do desempenho em aplicações de visualização em 3D. A técnica de *Frustum Culling* já se tornou corriqueira em aplicações de visualização, sendo implementada de forma eficiente na grande maioria dos softwares de visualização já existentes.

No caso do *Level of Detail*, a técnica de *Quick-VDR* demonstrou os melhores resultados e também provou-se frente a diferentes tipos de bases de dados: desde simulações de fluidos até modelos CAD. Porém, a escolha de não explorá-la inclui seu enorme tempo de pré-processamento e seu algoritmo de simplificação que é certamente o mais complexo dentre os que compõem este estudo.

Já no caso de substituição de geometria, a técnica de *GPU Primitives* aponta para uma solução eficiente para modelos CAD, enquanto a opção de *Far Voxels* se mostra mais adaptável a variados tipos de modelos. Ambas indicam uma tendência para um futuro próximo onde grandes modelos podem ser substituídos de forma eficiente por representações mais simples de sua geometria. Explorando o hardware gráfico cada vez mais programável, será possível otimizar ainda mais a visualização destes grandes volumes de dados.

A técnica de *Occlusion Culling* baseada em *occlusion queries* sobre uma estrutura hierárquica merece destaque por ter participação importante tanto na proposta de *Quick-VDR* quanto na de *Far Voxels*. É importante observar que estas propostas mais promissoras utilizam hierarquias para agrupar os testes de visibilidade e aplicar os níveis de detalhes das geometrias. O uso de uma hierarquia garante que estes métodos são escaláveis, podendo ser utilizados para modelos cada vez maiores sem grandes perdas de performance.

Na decisão de qual técnica deveria ser implementada, ponderou-se dois fatores principais. O primeiro consiste em apoiar desenvolvedores que desejam melhorar o desempenho de suas aplicações para visualização de modelos massivos. Além disso, o caráter de projeto de estudo deste trabalho favorece a exploração de técnicas que necessitam de pesquisas nas mais diversas áreas da Computação em geral. Considerando estes dois objetivos, definiu-se como técnica principal a ser explorada e desenvolvida a chamada *Coherent Hierarchical Culling*.

Esta técnica de *Occlusion Culling* exige um estudo e desenvolvimento de algoritmos de construção de volumes envolventes bem-ajustados, além de algoritmos de divisão espacial com a construção de diferentes tipos de estruturas de dados hierárquicas. Além disso, estas estruturas serviriam como base para implementação de algoritmos de *Frustum Culling* e suas otimizações. Mais uma vez, o uso de uma hierarquia irá garantir a escalabilidade da solução proposta.

Ao se construir uma biblioteca com alto grau de coesão e baixo acoplamento, diferentes aplicações são capazes de utilizar apenas as partes da biblioteca que lhe sejam mais convenientes. A seguir, serão descritas em maiores detalhes as técnicas desenvolvidas neste projeto. Para maiores detalhes da arquitetura e implementação da biblioteca, vide o capítulo 5. Para uma análise dos resultados obtidos para as técnicas desenvolvidas, vide o capítulo 6.

## 4 Técnicas Desenvolvidas

Como enunciado anteriormente, a principal técnica de otimização desenvolvida é baseada em *Occlusion Culling* juntamente com o *Frustum Culling*. A seguir, serão descritas estas técnicas de otimização, assim como os algoritmos e estruturas de dados estudados que apóiam as soluções propostas. Mais tarde serão analisados as relações entre os algoritmos de caixas envolventes, de construção de hierarquia e o seu impacto no desempenho das aplicações de visualização.

### 4.1 Volumes Envolventes

Foram estudados dois dos volumes mais utilizados na literatura: AABB e OBB. Pelo seu ajuste notoriamente ruim, não foi considerado o uso de esferas envolventes neste projeto. Já no caso da AABB, seu algoritmo de construção é quase trivial porém seu ajuste é em geral pior do que uma OBB.

Além das razões acima, resultados de estudos apontam a OBB como volume envolvente de melhor custo benefício para *frustum culling* [16]. Finalmente, o uso de *occlusion queries* requer volumes envolventes com pouca complexidade geométrica e bom ajuste. Isto favorece o uso de caixas como melhor opção.

#### 4.1.1 AABB

Como o próprio nome sugere, a caixa alinhada com os eixos é composta pelos oito vértices que definem a caixa, sendo esta orientada de modo que suas seis faces sejam ortogonais aos eixos coordenados  $x$ ,  $y$  e  $z$ .

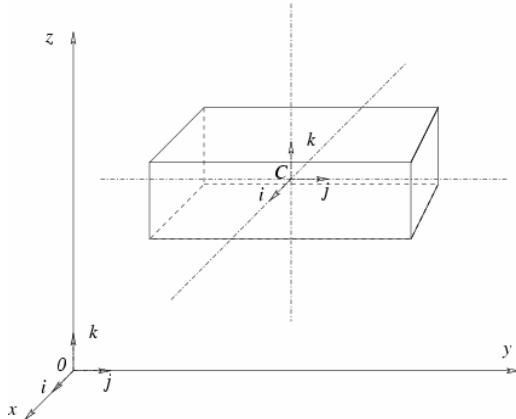


Figura 4.1.1.1: Os eixos  $i$ ,  $j$  e  $k$  de uma AABB são alinhados com os eixos  $x$ ,  $y$  e  $z$ .

O algoritmo de construção da AABB é simples: basta percorrer todos os vértices a serem envolvidos pela caixa e armazenar os valores mínimos e máximos em cada coordenada  $x$ ,  $y$  e  $z$ . Sua complexidade é, portanto  $O(n)$ <sup>11</sup>, onde  $n$  é o número de vértices a serem envolvidos pela AABB. De posse destes valores  $\mathbf{V}_{\min} = (x_{\min}, y_{\min}, z_{\min})$  e  $\mathbf{V}_{\max} = (x_{\max}, y_{\max}, z_{\max})$ , pode-se derivar os demais vértices que definem a caixa.

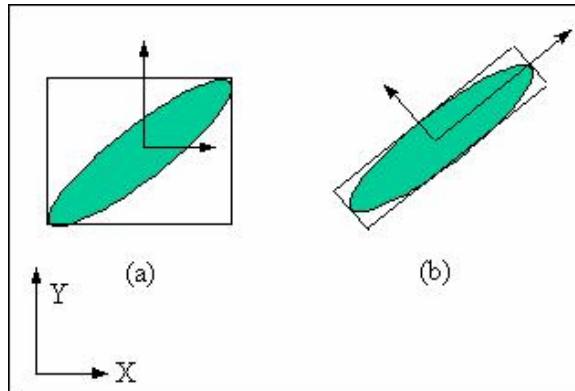


Figura 4.1.1.2: Ajuste ruim de uma AABB para a geometria 2D em verde. À direita, uma OBB exemplificando um melhor ajuste.

Pode-se observar que a AABB não é capaz de fornecer um ajuste tão preciso para certas distribuições de geometria. Por conta disso, foi dada maior atenção para a construção de uma OBB mais ajustada.

#### 4.1.2 OBB

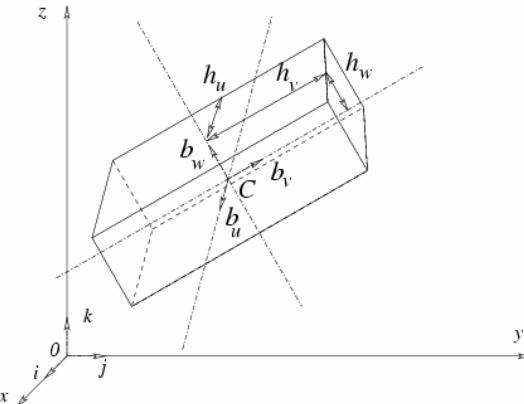
A caixa orientada possui como maior vantagem em relação à AABB a possibilidade de acompanhar a distribuição da geometria a ser envolvida. Por conta disso, existem variados algoritmos de construção de uma OBB que se propõem a encontrar a melhor orientação para a caixa. Ou seja, aquela que minimiza o tamanho final da mesma. Esta medida pode ser avaliada calculando-se o volume da caixa resultante ou, de maneira mais simples, através de sua área de superfície.

Neste projeto, será importante o cálculo eficiente da interseção de um volume envolvente com os planos do *frustum*, além de medidas de distância de um ponto qualquer ao volume sendo testado. Dessa forma, a representação escolhida para a OBB

<sup>11</sup> A notação *big-Oh* será utilizada como medida de eficiência dos algoritmos analisados ao longo deste trabalho. É particularmente importante observar o limite superior assintótico destes procedimentos já que o objetivo é lidar modelos massivos, caracterizados por enormes bases de dados.

segue a área de pesquisa onde esta se destacou: Colisão [19]. Esta representação visa simplificar cálculos de distância e interseção da caixa com demais objetos em uma cena.

A OBB é definida por três elementos: seu ponto central, seus eixos locais e metade das extensões da caixa ao longo de seus eixos locais. Estes eixos definem uma base ortonormal que representa a rotação da caixa em relação aos eixos coordenados  $x$ ,  $y$  e  $z$ . A escolha de representação das extensões da caixa como metade de seus valores, sempre positivos, é específica para acelerar os cálculos.



**Figura 4.1.2.1:** Uma OBB definida pelo seu centro  $C$ , seus eixos locais  $\mathbf{b}_u$ ,  $\mathbf{b}_v$  e  $\mathbf{b}_w$ , e suas extensões  $h_u$ ,  $h_v$  e  $h_w$ .

Uma possível desvantagem da OBB em relação à AABB é seu maior consumo de memória para ser representada. Porém, na implementação da biblioteca deste projeto não se observou nenhuma diferença significativa neste sentido. Além disso, os testes de interseção de uma OBB, apesar de otimizados, ainda são mais custosos do que os testes de uma AABB.

Como este ajuste dos volumes envolventes é de grande importância para os algoritmos de otimização a serem implementados, dedicou-se uma parcela do desenvolvimento da biblioteca às diferentes propostas de construção de OBBs mais compactas. Essencialmente, a qualidade do ajuste de uma OBB depende da orientação que se escolhe para a caixa. Depois disso, seu ponto central e extensões são computados de acordo com a orientação principal determinada. As diferenças entre os métodos descritos a seguir estão na forma de obter uma boa orientação, dadas as geometrias que serão envolvidas.

É importante ressaltar que não será detalhado o algoritmo de construção de uma OBB mínima [21], pois sua implementação é extremamente complexa. Além disso, seu tempo de processamento proporcional a  $O(n^3)$  o descarta como opção para modelos massivos, ainda que a construção das caixas seja feita em pré-processamento. A etapa

inicial deste algoritmo consiste no cálculo do fecho convexo<sup>12</sup> dos vértices a serem envolvidos. É demonstrado que a OBB mínima deve possuir uma face em comum com uma das faces do fecho convexo, ou então ter seus eixos dados por três arestas do fecho convexo que sejam perpendiculares entre si. Ao calcular-se as projeções e rotações dos vértices ao longo de cada face e cada tripla de arestas, encontra-se a caixa orientada mínima dentre todas as opções possíveis.

#### 4.1.2.1 Covariância

Para calcular a orientação da OBB, a abordagem clássica utiliza uma análise estatística sobre a distribuição da geometria a ser envolvida. Em outras palavras, procura-se identificar o eixo de maior variância de vértices. Este eixo “principal” é escolhido como um dos eixos da caixa, e os demais resultam da mesma computação do anterior. A construção da caixa, portanto, fará uso de estatísticas de primeira e segunda ordem dos vértices a serem envolvidos [20].

Inicialmente, o centro da caixa é calculado como a média de todos os vértices a serem envolvidos. Esta etapa, portanto, consome um tempo proporcional a  $O(n)$ . Esta é a estatística de primeira ordem das coordenadas dos vértices. Sejam os vértices do  $i$ -ésimo triângulo denotados por  $p^i$ ,  $q^i$  e  $r^i$  e  $n$  o número de triângulos que compõem a geometria a ser envolvida, calcula-se a média  $\mu$  dos vértices pela seguinte equação:

$$\mu = \frac{1}{3n} \sum_{i=0}^n (p^i + q^i + r^i)$$

Em seguida, calcula-se a estatística de segunda ordem dos vértices, representada pela matriz de covariância dos mesmos. A matriz de covariância é responsável por indicar como os vértices estão distribuídos em relação ao centro calculado. Ou seja, através desta matriz será possível obter as direções em que os vértices possuem maior variância em relação ao ponto central. Sejam  $j$  e  $k$  os índices de linha e coluna, respectivamente, define-se cada entrada da matriz de covariância por:

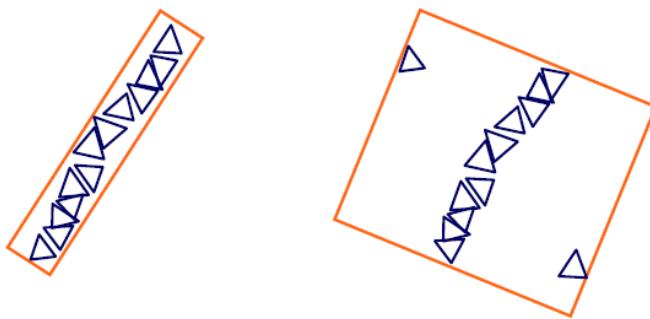
$$\mathbf{C}_{jk} = \frac{1}{3n} \sum_{i=0}^n (p_j^{ri} p_k^{ri} + q_j^{ri} q_k^{ri} + r_j^{ri} r_k^{ri}), \quad p^{ri} = p^i - \mu, q^{ri} = q^i - \mu, r^{ri} = r^i - \mu$$

Como esta matriz é necessariamente simétrica, seus autovetores são mutuamente ortogonais. Dessa forma, o autovetor correspondente ao maior autovalor da matriz representa o eixo de maior variância dos vértices. É portanto, escolhido como eixo

<sup>12</sup> O fecho convexo de um conjunto de pontos  $X$  é o menor subconjunto de pontos que formam um poliedro convexo que contém  $X$ .

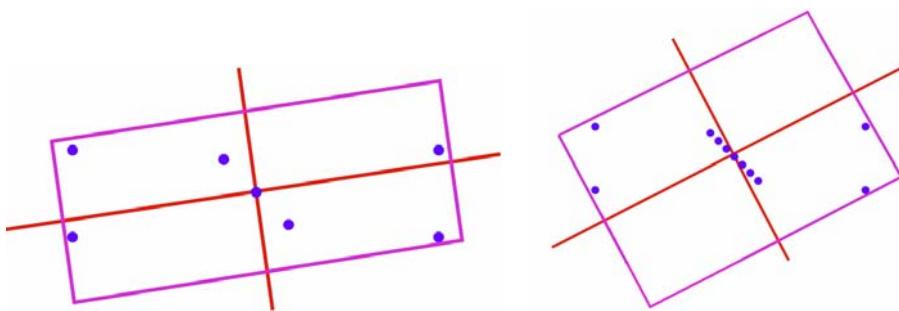
principal da caixa. Analogamente, os demais autovetores correspondem aos demais eixos da OBB.

Após esta escolha dos eixos da caixa, basta atualizar suas extensões ao longo dos três eixos computados e, finalmente, posicionar seu centro corretamente de acordo com as extensões calculadas. Todo este processo também consome um tempo  $O(n)$ , fazendo com que esta seja a complexidade final da construção de uma OBB segundo esta proposta.



**Figura 4.1.2.1.1:** Nem sempre a covariância produz OBBs bem-ajustadas.

Observa-se na figura 4.1.2.1.1 que nem sempre este método é capaz de obter uma caixa corretamente ajustada. Isto porque a covariância é influenciada pela quantidade de vértices contribuindo para cada direção principal em potencial. Ou seja, a caixa tenderá a se alinhar na direção de maior concentração de vértices, o que nem sempre resultará em um bom ajuste. Além disso, como o objetivo da OBB é envolver a geometria, seria igualmente benéfico descartar também os vértices internos da geometria para que os mesmos não influenciem erroneamente a orientação final da caixa.



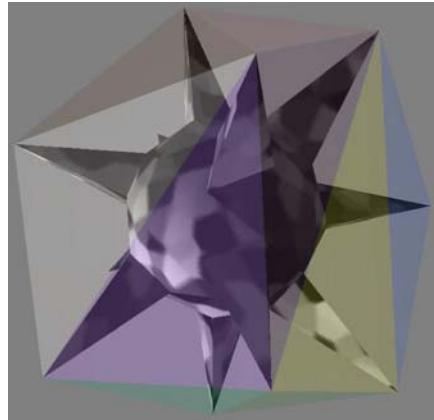
**Figura 4.1.2.2:** Vértices internos podem prejudicar orientação de uma OBB.

#### 4.1.2.2 Fecho Convexo

Esta abordagem para construção da caixa orientada procura contornar os problemas identificados anteriormente, onde se utilizam simplesmente a média e covariância dos vértices a serem envolvidos. Para eliminar os vértices internos que não

trazem nenhuma informação relevante ao cálculo da OBB, constrói-se o primeiramente o fecho convexo das geometrias envolvidas.

O fecho convexo de um conjunto de pontos é um problema bem-conhecido da Geometria Computacional, existindo muitos algoritmos eficientes para dados em 2, 3 ou mais dimensões. Escolheu-se o algoritmo conhecido por QuickHull [23], de complexidade  $O(n \lg n)$  no caso médio e  $O(n^2)$  no pior caso, análogas ao procedimento de ordenação QuickSort. Adaptou-se uma implementação disponível livremente deste algoritmo de forma a integrá-la na biblioteca desenvolvida.



**Figura 4.1.2.2:** Um poliedro qualquer envolvido por seu fecho convexo (semi-transparente).

De posse do fecho convexo dos objetos, é proposta mais uma alteração no cálculo da média e covariância: amostrar regularmente a superfície do fecho. Esta alteração procura evitar que concentrações maiores de vértices do fecho convexo ainda influenciem negativamente ao orientação final da OBB.

O primeiro estudo a propor este método [19] fornece as seguintes equações para cálculo da média e covariância segundo uma amostragem uniforme. A proposta aplica uma integral sobre toda a superfície do fecho convexo, integrando sobre a superfície de cada triângulo. A notação  $m^i$  corresponde à área do i-ésimo triângulo.

$$\mu = \frac{1}{n} \sum_{i=1}^n \left( \frac{1}{m^i} \int_0^1 \int_0^{1-t} x \, ds \, dt \right) = \frac{1}{6n} \sum_{i=1}^n \frac{1}{m^i} (p^i + q^i + r^i)$$

$$C_{jk} = \frac{1}{n} \sum_{i=1}^n m^i [(p'^j + q'^j + r'^j)(p'^k + q'^k + r'^k) + p'^j p'^k + q'^j q'^k + r'^j r'^k]$$

Entretanto, durante os estudos desta técnica em particular, foram encontradas duas outras estratégias para amostrar uniformemente sobre a superfície do fecho convexo. Na primeira proposta, denota-se por  $m^k$  o centróide do i-ésimo triângulo e por

$A^k$  sua área. O símbolo  $m$  corresponde ao centróide do fecho convexo, calculado como a média dos centróides de cada triângulo. Finalmente, o símbolo  $A_H$  corresponde à área total da superfície do fecho convexo. Dessa forma, cada entrada da matriz de covariância segue a equação:

$$\mathbf{C}_{i,j} = \sum_{k=1}^n \frac{A^k}{12A_H} [9\mathbf{m}_i^k \mathbf{m}_j^k + \mathbf{p}_i^k \mathbf{p}_j^k + \mathbf{q}_i^k \mathbf{q}_j^k + \mathbf{r}_i^k \mathbf{r}_j^k] - \mathbf{m}_i \mathbf{m}_j$$

Finalmente, um conceituado livro sobre algoritmos de visualização em 3D, chamado 3D Game Engine Design [22], propõe uma terceira forma de efetuar o cálculo da média e covariância sobre a superfície do fecho convexo. A média dos vértices é calculada segundo a seguinte equação:

$$\boldsymbol{\mu} = \frac{1}{12n} * \frac{1}{A_H} * \sum (A^K * (p_i + q_i + r_i))$$

E cada entrada de sua matriz de covariância é dada por:

$$\frac{1}{12n} * \frac{1}{A_H} * \sum (A^K * (p_i^k \cdot p_i^k + p_i^k \cdot q_i^k + p_i^k \cdot r_i^k + q_i^k \cdot p_i^k + q_i^k \cdot q_i^k + q_i^k \cdot r_i^k + r_i^k \cdot p_i^k + r_i^k \cdot q_i^k + r_i^k \cdot r_i^k))$$

Como o objetivo desta etapa do projeto é o estudo e avaliação das diferentes propostas existentes, optou-se por implementar as três metodologias encontradas para cálculo das estatísticas de primeira e segunda ordem sobre o fecho convexo. De posse destas três implementações, será possível avaliar a eficácia de cada uma.

#### 4.1.2.3 Caixa Mínima Aproximada

O último algoritmo de construção de uma OBB diverge dos demais por não utilizar uma análise estatística dos vértices a serem envolvidos. Ao invés disso, procura-se encontrar uma aproximação satisfatória da caixa orientada mínima [24].

O algoritmo utiliza um procedimento que encontra uma aproximação do diâmetro de um conjunto de pontos em  $d$  dimensões. O diâmetro de um conjunto de pontos é definido como o par de pontos de maior distância entre si. Esta aproximação de  $(1 - \epsilon)$  do diâmetro, juntamente com seu par de pontos correspondente, pode ser calculada em tempo  $O(n + (1/\epsilon^{3/2}) \lg(1/\epsilon))$ .

A idéia central do algoritmo é utilizar a aproximação do diâmetro do conjunto de pontos para aproximar o fecho convexo por um poliedro convexo de baixa complexidade.

Finalmente, calcula-se a caixa mínima exata através do algoritmo mencionado anteriormente [21]. O poliedro que aproxima o fecho convexo é escolhido de forma que a caixa mínima encontrada seja uma aproximação de  $(1 + \epsilon)$  da caixa mínima do conjunto de vértices original. O tempo total de execução deste algoritmo é  $O(n + 1/\epsilon^{4.5})$ .

Infelizmente, esta proposta é complexa para ser implementada. Uma alternativa mais simples também é apresentada. Esta alternativa utiliza um *grid*, construído a partir dos vértices de entrada, para testar combinações de caixas orientadas e obter uma aproximação da caixa de menor volume, de forma semelhante ao algoritmo já descrito. Esta alternativa possui complexidade  $O(n \lg n + n/\epsilon^3)$ . Vale destacar que ambas as abordagens possuem uma constante razoavelmente grande “escondida” na notação assintótica utilizada.

Utilizou-se uma implementação fornecida pelos autores da técnica que provê três heurísticas para o cálculo da caixa orientada mínima aproximada, todas baseadas no segundo algoritmo mencionado. A primeira utiliza apenas um número fixo de aproximações do diâmetro para calcular a caixa mínima envolvente. Com isso, procura-se melhorar a caixa construída a cada nova iteração. Contudo, sabe-se que esta abordagem pode convergir para um mínimo local, e não global.

A segunda implementação utiliza uma amostragem nos vértices das geometrias a serem envolvidas. Com esta amostragem, constrói-se um *grid* e utiliza-se as aproximações do diâmetro deste *grid* para se obter a caixa envolvente. O processo de amostragem procura melhorar seu desempenho para bases de dados muito grandes.

Finalmente, a terceira implementação traduz exatamente o algoritmo enunciado, construindo-se um *grid* sobre todos os vértices das geometrias a serem envolvidas e testando-se todas as aproximações do diâmetro possíveis. Encontra-se, assim, a caixa mínima aproximada para o conjunto de vértices original.

## 4.2 Hierarquia de Agrupamento Espacial

De posse dos algoritmos para construção de volumes envolventes, o próximo passo é desenvolver técnicas para agrupar espacialmente estes volumes de modo a obter uma hierarquia. Esta estrutura será importante em três momentos: no correto agrupamento espacial dos objetos na cena, no uso de *Frustum Culling* dos volumes envolventes e durante o *Coherent Hierarchical Culling*.

O objetivo principal do estudo destes algoritmos de divisão espacial é identificar e relacionar as diferentes abordagens e seus impactos no desempenho da aplicação de visualização 3D. Todos os algoritmos de construção estudados possuem complexidade  $O(n \lg n)$  no caso médio, onde  $n$  é o número de objetos na cena. Exceções serão indicadas e explicadas conforme são encontradas.

#### 4.2.1 BSP-Tree

Iniciou-se o estudo por uma estrutura hierárquica conhecida por *Binary Space-Partitioning Tree*, ou BSP-Tree [1]. De certa forma, esta estrutura é uma definição geral de uma árvore binária de divisão espacial. Muitas estruturas semelhantes podem ser reduzidas a casos particulares desta mesma hierarquia.

Como o nome sugere, esta estrutura consiste em uma árvore estritamente binária, onde um nó pai é subdividido em dois nós filhos e assim sucessivamente. Esta divisão é representada por um plano em particular, chamado plano de corte, responsável por dividir o volume envolvente do nó pai em dois volumes envolventes menores. Dessa forma, divide-se o conjunto original de objetos contidos pelo nó pai em dois conjuntos: os que estão no semi-espacô positivo do plano (na direção da normal do plano) e os que estão no semi-espacô negativo do mesmo (contra a direção da normal do plano).

Todos os objetos que forem classificados como estando no mesmo lado do plano de corte são designados para um mesmo filho. A maneira mais simples de efetuar este particionamento é classificar as geometrias de acordo com seu centro geométrico.

Este algoritmo, como enunciado, é efetuado *top-down*<sup>13</sup> de forma recursiva, até que uma determinada condição de parada seja atingida. Por exemplo, o número mínimo de objetos em um nó ou então uma profundidade máxima da árvore. Neste momento, este nó é declarado uma folha. Contudo, outras abordagens existem para uma construção *bottom-top*<sup>14</sup>, onde os objetos isolados da cena vão sendo progressivamente agrupados em volumes cada vez maiores. Da mesma forma, existem heurísticas para determinar o início e término do algoritmo.

A característica principal de uma BSP-Tree é o posicionamento do chamado plano de corte. Nesta estrutura em particular, há total liberdade para que este plano seja

<sup>13</sup> Constrói-se primeiro a raiz da hierarquia e depois progressivamente seus filhos, gerando por último as folhas da árvore.

<sup>14</sup> Primeiro constrói-se as folhas da árvore e depois progressivamente seus nós ancestrais, até que por último seja construída da raiz da hierarquia.

escolhido em qualquer posição e em qualquer orientação, desde que ele seja capaz de subdividir o volume envolvente em duas partes menores.

Por conta disso, existem inúmeros algoritmos que visam determinar o posicionamento e a orientação deste plano de corte. O mais simples é partitionar sempre na mediana espacial do volume envolvente do nó pai. Porém outras propostas procuram equilibrar a quantidade de geometrias em ambos os lados do plano ou então minimizar a quantidade de geometrias que são interceptadas pelo mesmo. Eventualmente, toda a inteligência na construção de uma BSP-Tree recai no posicionamento do plano de corte de modo a satisfazer certas heurísticas.

É importante identificar e estudar para cada aplicação quais são as características de uma “boa” hierarquia. Devido a esta flexibilidade, o estudo de heurísticas para a BSP-Tree será um dos principais objetivos no desenvolvimento dos algoritmos de divisão espacial neste projeto.

#### 4.2.2 Octree

A idéia fundamental da Octree [1] é subdividir o volume envolvente do nó pai utilizando não um, mas dois planos de corte perpendiculares entre si. Além disso, estes planos são sempre posicionados na mediana espacial do volume e orientados segundo dois dos eixos coordenados  $x$ ,  $y$  ou  $z$ . Imaginando que o volume envolvente do nó pai seja uma AABB, a partição de uma Octree irá gerar 8 AABBs de tamanho exatamente igual a  $1/8$  do volume original. Portanto, cada nó interno de uma Octree possui não dois, mas oito filhos.

A analogia para o caso bidimensional é a Quadtree, onde um quadrado é subdividido em 4 quadrados menores de tamanho  $1/4$  do original. Neste caso, cada nó interno possui 4 filhos, definidos pelos quadrados menores. Tanto no caso 3D como no caso 2D, o partitionamento dos objetos para os filhos ocorre de maneira análoga à BSP-Tree.

Normalmente, os planos de corte são escolhidos de forma a alternar sua orientação em relação a cada um dos eixos coordenados. Ou seja, o primeiro plano possui sua normal na direção do eixo  $x$ , o segundo possui sua normal na direção do eixo  $y$ , o terceiro na direção do eixo  $z$ , o quarto na direção do eixo  $x$  novamente e assim sucessivamente. Dessa forma procura-se balancear a divisão espacial ao longo da construção da hierarquia.

Caso o número de subdivisões espaciais seja constante, a Octree enunciada utilizando a AABB como volume envolvente nada mais é do que uma representação hierárquica de um *grid* regular. Este *grid* partitiona a cena inteira em células de mesmo

tamanho. Cada célula constitui um nó folha da Octree, cada conjunto de oito células representa um nó interno da hierarquia e assim sucessivamente.

Uma das desvantagens da Octree é sua regularidade. Todos os nós de um mesmo nível da hierarquia possuem um volume envolvente de mesmo tamanho. Dependendo da distribuição da geometria da cena, poderão ocorrer nós com muitas geometrias e nós quase vazios. Além disso, fixar a orientação dos planos de acordo com os eixos coordenados pode prejudicar a divisão espacial no caso de volumes envolventes não alinhados, como a OBB.

No final das contas, pode-se construir um caso particular de uma BSP-Tree que represente a mesma divisão espacial do que uma Octree correspondente. A árvore assim gerada terá apenas uma altura maior do que a Octree original. De qualquer forma, na literatura não há estudos que favoreçam o uso de uma Octree juntamente a algoritmos de *Frustum Culling* ou *Occlusion Culling*.

Por conta destes fatores, optou-se por não desenvolver algoritmos específicos para uma Octree, mantendo o foco em uma estrutura mais flexível como é o caso da BSP-Tree.

#### 4.2.3 OBB-Tree

Esta estrutura surgiu juntamente com a pesquisa de OBBs como volumes envolventes para colisão de corpos em simulações físicas [19]. Na verdade, a OBB-Tree não é uma estrutura de divisão espacial, e sim uma hierarquia de volumes envolventes que procura decompor um mesmo objeto em partes menores. Seu principal uso é para acelerar algoritmos de colisão, como já mencionado.

Normalmente para se colidir dois corpos, seria necessário testar a interseção de todos os triângulos de ambos os objetos uns com os outros. Claramente algo longe de ser possível em ambientes de simulação em tempo real. Surgiu então a necessidade de uma hierarquia que pudesse identificar rapidamente se dois corpos estavam colidindo ou não, e identificar seu ponto de contato sem necessidade de muito processamento. Neste contexto, foi proposta uma hierarquia que utilizava OBBs para partitionar um mesmo corpo em diferentes partes. Dessa forma, a colisão entre dois corpos se dá pela colisão entre suas OBB-Trees<sup>15</sup>.

---

<sup>15</sup> O algoritmo de colisão utilizando OBB-Trees foge ao escopo deste projeto, podendo ser encontrado nas referências mencionadas.

O algoritmo de construção desta hierarquia propõe o posicionamento do plano de corte na média dos centros dos objetos de um nó, alinhado com o eixo mais longo da OBB. Dessa forma, procura-se equilibrar a complexidade geométrica dos filhos a serem gerados. Caso se opte por construir as OBBs a partir do fecho convexo das geometrias envolvidas, a complexidade final do algoritmo de construção aumenta para  $O(n \lg^2 n)$ .

De qualquer forma, pode-se definir a OBB-Tree como um caso de BSP-Tree em que o plano de corte é alinhado com o eixo de maior extensão da OBB e posicionado na média dos centros dos objetos envolvidos.

#### 4.2.4 kd-Tree

A definição de uma kd-Tree [1] talvez seja tão genérica quanto a de uma BSP-Tree: *k-dimensional tree*. Apesar do nome, a kd-Tree clássica possui algumas particularidades que a torna, mais uma vez, um caso particular de uma árvore BSP. A kd-Tree é utilizada nas mais diversas áreas da Computação, auxiliando algoritmos de *clusterização* e dando apoio a problemas de busca espacial (como o problema de encontrar um ou mais vizinhos mais próximos de um elemento).

A kd-Tree clássica é definida como uma árvore estritamente binária onde cada nó contém um plano de corte. Este plano é sempre orientado de acordo com um dos eixos coordenados mas, ao contrário de uma Octree, o plano de corte pode ser posicionado em qualquer ponto ao longo da extensão do nó. Isto significa que a estrutura possui maior liberdade para balancear as geometrias contidas em cada nó e, por conta disso, existem diferentes heurísticas para gerar árvores otimizadas para cada tipo de aplicação. Devido à orientação fixa dos planos de corte, kd-Trees normalmente são associadas ao uso da AABB como volume envolvente.

Kd-Trees ganharam notoriedade mais recentemente graças a estudos que a identificaram como estruturas eficientes para *ray-tracing* [25]. Neste caso, utiliza-se AABBs em torno dos triângulos de uma cena e constrói-se a hierarquia de forma *top-down*, escolhendo-se a posição e orientação do plano de corte de modo a minimizar uma função de custo conhecida por SAH ou *Surface Area Heuristic*. Esta heurística de construção provou ser capaz de gerar hierarquias mais eficientes para *ray-tracing* em todos os casos estudados.

A heurística de SAH merece destaque pois modela a construção da hierarquia por um algoritmo guloso que procura encontrar uma aproximação da árvore ótima segundo este critério. Às vezes, o algoritmo converge para um mínimo local, mas as hierarquias construídas provaram ser mais eficientes do que todas as propostas anteriores. Ao minimizar a função de custo a seguir para ambos os filhos a serem

gerados, considerando todos os planos  $p$  possíveis, a técnica minimiza o custo total de caminhamento e interseção de raios da árvore.

$$\mathcal{C}_V(p) = \mathcal{K}_T + \mathcal{P}_{[V_l|V]} \mathcal{C}(V_l) + \mathcal{P}_{[V_r|V]} \mathcal{C}(V_r).$$

Esta fórmula modela a escolha gulosa do algoritmo para um dado plano  $p$ . As grandezas  $P_{[vx|v]}$  representam a probabilidade do volume  $V_x$  ser interceptado por um raio qualquer. Esta probabilidade é dada como a razão entre a área de superfície deste volume menor e a área de superfície do volume do nó pai.

A idéia básica de SAH decorre da modelagem do custo de caminhar na hierarquia de acordo com o custo de se calcular a interseção de um raio com os objetos contidos em um determinado nó. Este custo é o fator determinante no desempenho de uma aplicação de *ray-tracing*.

$$\mathcal{C}(T) = \sum_{n \in nodes} \frac{\mathcal{SA}(V_n)}{\mathcal{SA}(V_s)} \mathcal{K}_T + \sum_{l \in leaves} \frac{\mathcal{SA}(V_l)}{\mathcal{SA}(V_s)} \mathcal{K}_I,$$

Nesta equação,  $C(T)$  é o custo de uma hierarquia.  $SA$  representa uma função que calcula a área de superfície de um volume envolvente.  $V_s$  corresponde ao volume da cena inteira.

Como no caso deste projeto é interessante a construção de volumes envolventes bem ajustados, pode-se considerar o uso de uma heurística similar ao SAH. Tal abordagem e seus resultados será analisada em detalhes mais adiante.

#### 4.2.5 Heurísticas de Construção

O estudo das hierarquias mencionadas indicou o uso de uma BSP-Tree como forma genérica de estrutura de divisão espacial. Diferentes algoritmos de construção e heurísticas para este caso são capazes de gerar hierarquias equivalentes às outras estruturas estudadas.

Como forma de simplificar as heurísticas desenvolvidas, por padrão todas elas procuram orientar o plano de corte de acordo com o eixo de maior extensão da caixa envolvente, seja ela uma AABB ou OBB. O fator de decisão, portanto, é o posicionamento do plano de corte ao longo do eixo definido. Caso o plano de corte escolhido não seja capaz de particionar as geometrias de um nó, este é declarado uma folha e a recursão retorna.

Os critérios de parada para declarar um nó como folha são o princípio o número mínimo de objetos (normalmente um) e a altura máxima da árvore, definida por uma equação encontrada nos estudos de kd-Trees para *ray-tracing*:

$$\text{MaxTreeDepth} = 1.2 * \log_2(\text{Total geometries}) + 2.0$$

Adicionalmente, existe a opção do usuário de determinar um número mínimo de vértices a serem agrupados em uma mesma folha. Caso a subdivisão de um dado nó gere um filho com complexidade menor do que a indicada, o dado nó é declarado indivisível e, portanto, nenhum filho é gerado.

Diferentes cenas possuem melhor desempenho para diferentes valores de número mínimo de vértices nas folhas. Uma discussão mais detalhada destas e outras heurísticas relacionando-as à performance da aplicação se encontra no capítulo 6.

#### 4.2.5.1 Mediana dos Objetos

Esta simples heurística posiciona o plano de corte na mediana dos objetos ao longo do eixo de maior extensão da caixa envolvente. Para isso é necessário ordenar as geometrias de acordo com seus centros ao longo do eixo escolhido. Esta ordenação é local, ou seja, apenas sobre os objetos contidos no nó corrente. Assim, a complexidade final do algoritmo de construção da hierarquia aumenta para  $O(n \lg^2 n)$ .

Caso o número de objetos seja par, existem duas medianas. O plano de corte é então posicionado no ponto intermediário entre os centros das medianas. Caso o número de objetos seja ímpar, optou-se por posicionar o plano de corte no ponto intermediário entre os centros dos objetos imediatamente antes e imediatamente após a única mediana existente. Dessa forma, o lado do plano em que a mediana estará não está definido, dependendo da distribuição espacial dos objetos.

Esta heurística garante a construção de uma árvore binária cheia e completa. Ou seja, se uma cena possuir  $k$  objetos e cada folha da árvore conter apenas um objeto, a árvore construída terá  $k$  folhas e  $2k - 1$  nós internos. Vale ressaltar que este tipo de balanceamento não necessariamente é o desejado para os algoritmos de renderização. Por exemplo, uma partição na mediana dos objetos pode gerar filhos desbalanceados em relação ao tamanho de seus volumes envolventes ou em relação à complexidade de suas geometrias.

#### 4.2.5.2 Mediana Espacial

Uma solução parecida com a anterior é posicionar o plano de corte na mediana espacial ao longo do maior eixo da caixa envolvente. Para isso, utiliza-se o próprio centro do volume envolvente. Enquanto a solução anterior visa equilibrar a quantidade de objetos em cada nó filho subseqüente, esta heurística visa equilibrar o volume dos nós filhos a serem gerados.

Entretanto, o algoritmo de mediana espacial por si só não garante o equilíbrio do volume dos nós filhos, devido ao fato da partição dos objetos se dar de acordo com o centro geométrico dos mesmos. Existem casos em que um nó não consegue ser mais subdividido e existem casos em que uma divisão bem-sucedida não equilibra os volumes dos filhos gerados.

#### 4.2.5.3 Mediana dos Centros

A mediana dos centros surge como forma de efetuar uma divisão mais equilibrada do que as propostas anteriores e, ao mesmo tempo, garantir que um nó com mais de uma geometria sempre será particionado com sucesso. Nesta heurística, posiciona-se o plano de corte no ponto médio entre o centro da primeira e da última geometrias.

#### 4.2.5.4 Média dos Centros

Análoga ao algoritmo de construção proposto pela OBB-Tree, esta heurística calcula a média dos centros de todos os objetos e posiciona o plano de corte neste ponto. Dessa forma, se um dos extremos da caixa possuir uma maior complexidade geométrica, o plano de corte estará mais próximo do mesmo.

#### 4.2.5.5 Equilibrar Complexidade Geométrica

As heurísticas anteriores procuram de alguma forma equilibrar ou o número de objetos a serem designados para os filhos de um nó, ou então seus volumes envolventes. Contudo, equilibrar o número de objetos em cada filho não garante o equilíbrio de sua complexidade geométrica, já que diferentes objetos possuem diferentes quantidades de vértices.

Como forma de traduzir explicitamente um equilíbrio na complexidade geométrica, foi implementada uma heurística que posiciona o plano de corte de modo a partitionar da forma mais equilibrada possível o número de vértices dos objetos a serem

particionados. É necessário ordenar os objetos ao longo do eixo escolhido, aumentando a complexidade da construção da árvore para  $O(n \lg^2 n)$ .

Percorre-se as geometrias em ordem, acumulando-se a quantidade de vértices até o momento. Quando esta quantidade se torna maior do que a metade do total de vértices de todos os objetos, identifica-se um ponto de quebra para posicionamento do plano de corte. É necessário apenas verificar se a última geometria incluída em um dos lados do plano, se alocada no outro lado, permitirá um melhor balanceamento da complexidade final dos filhos.

#### 4.2.5.6 Minimizar SAH

Devido ao grande sucesso da heurística de SAH mencionada anteriormente para o caso de *ray-tracing*, experimentou-se o uso de uma heurística semelhante para a biblioteca desenvolvida. Na função de custo implementada, utiliza-se a complexidade geométrica dos filhos (número de vértices) como forma de ponderar a qualidade da subdivisão. A equação original utilizava apenas a quantidade de objetos para este mesmo fim<sup>16</sup>.

No caso da heurística original proposta, inúmeras otimizações foram possíveis de modo a manter a complexidade da construção da hierarquia ainda em  $O(n \lg n)$ . Isto se deve ao uso de uma kd-Tree como uma representação de volumes envolventes (AABBS) de forma implícita. Além disso, bastava manter apenas o número de geometrias em cada lado dos potenciais planos de corte.

No caso deste projeto, existe a necessidade de se determinar exatamente o volume dos filhos para se classificar corretamente a qualidade da subdivisão. O algoritmo desenvolvido desta forma necessita computar explicitamente as caixas envolventes dos filhos para cada posição e orientação possíveis do plano de corte. E mais ainda, estas caixas devem ser construídas com o mesmo algoritmo que o utilizado para o restante da hierarquia, para manter consistente a avaliação da função de custo. O algoritmo então implementado, para fins de teste, possui complexidade quadrática. Isto faz com que a construção da hierarquia leve um tempo muito maior do que as abordagens anteriores. Otimizar tal implementação pode se tornar um importante assunto de pesquisa futura.

Uma outra vantagem de se utilizar esta heurística é um critério adicional de parada na recursão, que procura avaliar a qualidade da subdivisão que será efetuada:

<sup>16</sup> Na verdade, em seu contexto original, o SAH foi proposto para uma árvore de *ray-tracing* de triângulos. Neste caso, para a hierarquia cada triângulo é um objeto. Portanto, ponderar pelo seu número de “objetos” é equivalente a ponderar pela complexidade geométrica dos objetos na cena.

caso a melhor subdivisão segundo o SAH ainda gere pelo menos um filho com volume envolvente quase tão grande quanto o pai, é preferível que o nó nem seja subdividido e nenhum filho seja gerado. Utiliza-se então o teste a seguir:

```
if largest child area > k * current box area
    return indivisible
```

O valor da constante  $k$  sugerido por [25] era de 0,75. Durante os testes neste projeto, observou-se que muitos nós eram declarados como indivisíveis desnecessariamente. Portanto, optou-se por um valor de 0,85 para a constante  $k$ . De maneira nenhuma este valor garante um bom desempenho na construção da árvore ou em seu uso durante a renderização de uma cena. Outros valores poderão ser mais adequados para diferentes cenas.

## 4.3 *Frustum Culling*

A técnica de *Frustum Culling* implementada utiliza a hierarquia de volumes envolventes, construída segundo uma das heurísticas anteriores, para descartar rapidamente os objetos que se encontram totalmente fora do volume de visão.

### 4.3.1 Extração dos Planos do *Frustum*

A extração dos planos do *frustum* segue um algoritmo específico para OpenGL<sup>17</sup>, capaz de fazê-lo de forma limpa e eficiente, sem necessidade de inversão de matrizes [26]. Basta que seja fornecida uma matriz de transformação de coordenadas espaciais. Caso a matriz fornecida seja apenas a *Projection*, os planos extraídos se encontrarão no espaço da câmera. Caso a matriz seja o produto entre a *View* e a *Projection*, os planos serão extraídos no espaço global.

Como todos os objetos da cena e todos os volumes envolventes da hierarquia se encontram já definidos no espaço global, optou-se por extrair os planos utilizando a *View* combinada à *Projection*. Dessa forma, não há necessidade de transformar a todo momento cada volume envolvente para o espaço onde está definido o *frustum*.

---

<sup>17</sup> Um algoritmo análogo também é apresentado para Direct3D.

### 4.3.2 Algoritmo Hierárquico

O algoritmo base é simples: conforme se percorre a hierarquia em pré-ordem<sup>18</sup>, testa-se cada nó visitado contra o volume de visão. Foi convencionado que os planos do *frustum* possuem suas respectivas normais apontando para dentro do volume de visão.

O teste dos planos procura seguir um esquema *early-out*, ou seja, classifica-se um volume como fora do *frustum* assim que o mesmo se encontre no lado negativo de um dos planos testados. De forma análoga, para um volume ser classificado como dentro do *frustum*, é necessário que ele seja encontrado no lado positivo de todos os planos do volume de visão.

Caso um nó se encontre fora do volume de visão, ele é descartado juntamente com suas sub-árvore à esquerda e à direita. O caminhamento continua em um irmão do nó visitado, caso exista, ou no próximo ancestral ainda não visitado na árvore.

Como as aplicações de modelos massivos geralmente consistem no caminhamento e inspeção destes modelos, é comum que o observador se encontre imerso dentre os objetos da cena. Assumindo esta hipótese, os planos do *frustum* são testados na seguinte ordem: *near, left, right, top, bottom e far*. Assim, procura-se eliminar rapidamente os volumes mais prováveis de estarem fora do *frustum* de visão.

### 4.3.3 Melhorias

Foram implementadas duas das otimizações estudadas para a técnica de *Frustum Culling*. Uma delas é a utilização do último plano a descartar cada nó da hierarquia. Esta técnica procura utilizar uma coerência temporal entre os quadros.

A outra otimização implementada é a utilização de uma máscara dos planos, para evitar que os filhos de um nó sejam testados, desnecessariamente, contra os planos que o nó pai já foi encontrado como estando totalmente do lado positivo.

---

<sup>18</sup> Primeiro visita-se o nó corrente, depois seus filhos.

## 4.4 Occlusion Culling

Apesar de todos as técnicas e otimizações mencionadas até o momento, ainda não seria possível a visualização interativa de modelos massivos. Como fruto dos estudos de técnicas de otimização específicas para este problema, optou-se pelo aprofundamento em uma proposta conhecida por *Coherent Hierarchical Culling*.

Como o nome já descreve, esta técnica utiliza uma hierarquia de volumes envolventes para determinar rapidamente a visibilidade dos nós e, portanto, dos objetos em uma cena. Nesta hierarquia, assume-se que apenas as folhas contém geometrias, como uma simplificação do algoritmo. Para os testes de visibilidade, utiliza-se uma extensão do hardware gráfico conhecido por *occlusion queries*.

### 4.4.1 Occlusion Queries

O processo de efetuar uma *occlusion query* é simples: antes de se renderizar uma geometria, o hardware gráfico é avisado que deseja-se efetuar uma *occlusion query*, enviando-se também um número identificador desta *query*. Este número representa uma estrutura conceitual chamada de *query object*. Em seguida, efetua-se a renderização da geometria da forma usual. No final da renderização, avisa-se o hardware de que a *query* terminou.

```
ID = generateQueryObject()
beginOcclusionQuery( ID )
renderGeometry()
endOcclusionQuery()
```

Neste momento, o processador gráfico irá transformar a informação geométrica enviada em fragmentos. Estes fragmentos passam por uma série de testes de acordo com o *pipeline gráfico* até que possam ser confirmados como visíveis, quando são promovidos a *pixels* e passam a compor a imagem final da cena. Com o uso de uma *query*, a placa gráfica armazena para cada *query object* o número de *pixels* que foi gerado com sucesso durante sua última renderização. Define-se um objeto como visível caso pelo menos um *pixel* seja gerado durante seu processo de renderização. Caso deseje-se uma aproximação da visibilidade, pode-se utilizar um valor mínimo de *pixels* que deve ser gerado antes de se considerar uma geometria como visível.

Como enunciado, o uso de uma *occlusion query* para determinar a visibilidade de um objeto contradiz seu objetivo principal: evitar o envio e processamento de geometrias não-visíveis para a placa gráfica. Para este fim, efetua-se uma *query* não sobre a geometria do objeto a ser testado, mas sobre seu volume envolvente.

Claramente, esta troca será benéfica somente se o custo de renderizar o volume envolvente for muito menor do que o custo de renderizar a geometria de um objeto.

Além disso, vale destacar o fato de que o teste utilizando o volume envolvente poderá acusar falsa visibilidade caso este volume não seja bem-ajustado à geometria. Nestes casos, serão renderizadas geometrias determinadas visíveis que na verdade não irão contribuir para a imagem final da cena. Por conta destes fatores dedicou-se uma parcela deste projeto ao estudo de volumes envolventes simples e bem-ajustados a geometrias em geral. Por conta destes fatores, escolhe-se a utilização de caixas como volumes envolventes, já que possuem apenas oito vértices e podem ser renderizadas de forma eficiente.

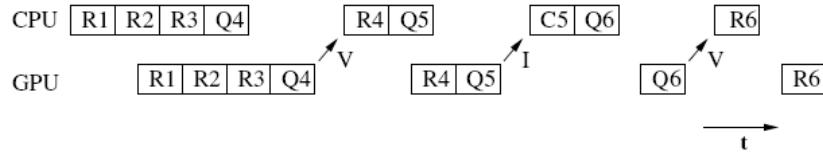
Outra questão importante a ser ressaltada é que resultados falsos de visibilidade podem ser obtidos também devido à ordem em que os objetos são renderizados. Como o resultado de uma *occlusion query* depende do conteúdo corrente do *framebuffer*, é necessário garantir que a geometrias já renderizadas constituem potenciais oclusores para as próximas geometrias a serem testadas. Para isto, renderiza-se as geometrias mais próximas do observador primeiro, e as demais em ordem crescente de distância do mesmo.

Este comportamento pode ser obtido ao se caminhar na hierarquia de uma cena em *front-to-back*. Existem diferentes mecanismos para efetuar este caminhamento específicos para diferentes tipos de hierarquia. No caso deste projeto, como cada nó da hierarquia possui um plano de corte, para caminhar em *front-to-back* bastaria indagar em qual lado deste plano se encontra o observador. Caso ele se encontre no lado negativo do plano, a sub-árvore à esquerda do nó estará mais próxima do observador do que a sub-árvore à direita do nó e vice-versa.

Entretanto, para garantir um algoritmo *front-to-back* geral, utilizou-se uma fila de prioridades para determinar a ordem em que os nós devem ser visitados durante o caminhamento na hierarquia. Neste caso, um nó de alta prioridade encontra-se próximo ao observador e um nó de baixa prioridade encontra-se distante do mesmo. É necessário, portanto, armazenar a informação da distância de cada nó até o observador e atualizar este valor conforme os nós são visitados. Além da vantagem de tornar o algoritmo independente da hierarquia utilizada, esta abordagem possui a vantagem de garantir uma ordenação mais precisa dos nós visitados, processando-os em camadas de profundidade cada vez maior.

Por motivos óbvios, o resultado de uma *occlusion query* só está disponível para a aplicação após o término da renderização da geometria sendo testada. Caso a CPU procure obter o resultado de uma *query* antes que a mesma tenha terminado, ela será obrigada a esperar a placa gráfica terminar de efetuar a renderização corrente. Este fenômeno é conhecido por *CPU Stall*. Este atraso da CPU diminui o volume de dados

transferidos para a GPU. Esta diminuição pode fazer com que a GPU opere por vários ciclos sem ter nenhum dado a ser processado. Este desperdício de processamento da placa gráfica é conhecido por *GPU Starvation*.

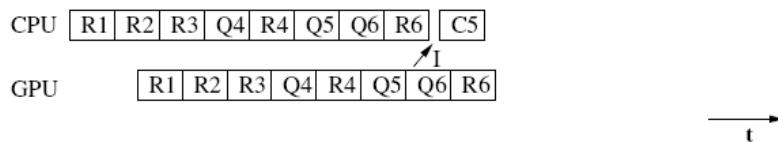


**Figura 4.4.1.1 :**  $R_n$ ,  $Q_n$  e  $C_n$  denotam renderização, envio de *query* e *culling* do objeto  $n$ .  
Nota-se os intervalos em que a CPU ou a GPU não efetuam nenhum processamento.

No modelo de computação utilizado, a CPU e a GPU operam de forma paralela. Enquanto a CPU processa a cena e envia as geometrias para a GPU, esta se encarrega de efetuar a renderização dos objetos recebidos. A única garantia que existe é que os dados serão processados na placa gráfica na ordem em que foram enviados pela CPU. Por conta disso, uma aplicação de visualização conseguirá desempenho ótimo quando a CPU for capaz de prover constantemente dados para serem processados pela GPU, ao mesmo tempo em que não sobrecarregue tanto o barramento de transferência entre ambas quanto a capacidade de processamento do hardware gráfico.

O maior desafio no uso de *occlusion queries*, portanto, é evitar *CPU Stalls* e minimizar a ocorrência de *GPU Starvation*. Para este fim, deve-se ocupar a CPU com outras tarefas enquanto espera-se o resultado de uma *query*. Felizmente, a API gráfica possui um mecanismo para indagar, sem *overhead* significativo, se uma dada *occlusion query* já terminou ou se ainda está sendo processada.

No caso de uso desta extensão para determinar a visibilidade, propõe-se que logo após o envio de uma *occlusion query* a CPU envie para serem renderizados objetos visíveis ou então que sejam efetuadas *queries* independentes para outros objetos. Dessa forma, a CPU não trava esperando o resultado da primeira *query* e mantém ocupada a GPU sem desperdiçar de seu processamento.



**Figura 4.4.1.2 :** Envio mais eficiente de *occlusion queries*, evitando *Stalls* e *Starvation*.

#### 4.4.2 Algoritmo Hierárquico

De forma similar ao uso de uma hierarquia no caso do *Frustum Culling*, utiliza-se uma árvore de volumes envolventes para determinar rapidamente se um conjunto de objetos encontra-se visível ou oculto por detrás de outros.

Caso a *query* do volume envolvente de um nó retorne um número *pixels* gerados maior do que um certo mínimo (normalmente zero), considera-se o nó visível. Neste caso, deve-se renderizar suas geometrias, se houver alguma, e continuar a caminhar na hierarquia visitando seus filhos. Caso a *query* do volume envolvente de um nó resulte em um número *pixels* gerados menor do que o mínimo determinado, considera-se o nó como não-visível e pode-se descartar não somente o nó em questão mas também suas sub-árvores à esquerda e à direita. O caminhamento na árvore continua em um irmão do nó em questão, se houver, ou no próximo ancestral ainda não visitado.

Como destacado anteriormente, o resultado de uma *query* não está prontamente disponível logo após seu envio e a CPU deve se ocupar com outras tarefas (e ocupar a GPU também) para evitar uma perda de desempenho por parte da aplicação de visualização. Pode-se perceber, também, que dependendo do agrupamento espacial gerado diferentes objetos de diferentes complexidades geométricas serão testados para visibilidade. Encontrar um equilíbrio entre o particionamento espacial e uma divisão eficiente para o uso de *occlusion queries* é uma dificuldade que este projeto pretende combater.

Por conta destes fatores, o algoritmo de *Occlusion Culling* proposto procura gerenciar de forma inteligente três passos: o envio de *occlusion queries* para nós recentemente visitados, o teste de disponibilidade das *queries* já enviadas e a obtenção dos resultados das mesmas. Neste momento, pode-se renderizar um objeto visível e continuar a percorrer a árvore a partir do nó testado. Para que isto seja efetuado da forma mais eficiente possível, é explorada a coerência temporal entre os resultados de visibilidade de um quadro anterior para o seguinte.

#### 4.4.3 Coerência Temporal

As informações de visibilidade do quadro anterior são aproveitadas no quadro corrente através de três decisões principais:

1. Somente efetua-se *occlusion queries* para os nós da hierarquia onde o caminhamento terminou no quadro anterior.
2. Assume-se que uma folha anteriormente visível continuará visível no quadro atual, podendo ser renderizada sem que se espere o resultado de sua *occlusion query*.
3. As *queries* efetuadas no quadro corrente são armazenadas em uma fila, enquanto são processadas pela GPU.

O algoritmo de caminhamento na hierarquia pode terminar em dois momentos: em nós folhas ou em nós internos classificados como invisíveis. Estes nós são então

classificados como *termination nodes*. Os nós internos classificados como visíveis são chamados então de *opened nodes*. O algoritmo utilizado evita o envio de *occlusion queries* para todos os *opened nodes*. Quando o algoritmo encontra um *termination node*, pode ocorrer uma das duas situações: ou ele é uma folha ou ele é um nó interno previamente invisível.

No caso do *termination node* ser um nó previamente invisível, envia-se uma *query* do volume envolvente deste nó, que é então armazenada na fila de *queries*. O caminhamento continua em um irmão do nó visitado, se houver, ou no próximo ancestral do mesmo.

No caso do *termination node* ser uma folha previamente visível, normalmente seria efetuada a *query* de seu volume envolvente e, como otimização, a geometria seria renderizada logo em seguida, sem que se espere o resultado de seu teste de visibilidade.

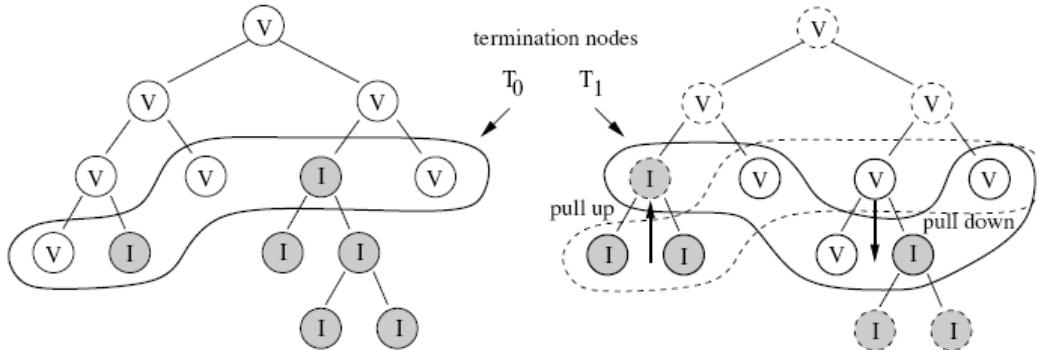
Porém, foi implementada uma otimização sugerida pelos autores da técnica: ao invés de se enviar primeiro o teste do volume envolvente para imediatamente depois renderizar uma geometria, efetua-se a *query* em cima da própria renderização da geometria. Isto traz consigo duas vantagens: evita-se a troca de estados e o custo adicional de renderização do volume envolvente e obtém-se um resultado de visibilidade mais preciso do que se fosse utilizado a caixa envolvente. Esta abordagem mais conservativa pode fazer com que mais geometrias do que o necessário sejam renderizadas.

Na forma enunciada, o algoritmo caminha a hierarquia inteira e toma as seguintes decisões para cada nó encontrado:

- Caso seja um nó interno previamente visível: Simplesmente continua a visitar suas sub-árvores. Declara o nó como invisível por *default*.
- Caso seja um nó previamente invisível: Envia-se uma *occlusion query* para seu volume envolvente, armazena-se a *query* em uma fila e continua-se o caminhamento em um irmão do nó visitado, se houver, ou no próximo ancestral do mesmo.
- Caso seja uma folha previamente visível: Envia-se uma *occlusion query* em cima da renderização da própria geometria e armazena-se a *query* em uma fila. Declara o nó como invisível por *default*.

Nota-se que conforme a hierarquia foi visitada, os nós foram sendo declarados como invisíveis por *default*. Isto é efetuado por um motivo essencial: um nó só pode ser declarado visível se pelo menos um de seus filhos for visível. Porém, caminha-se na hierarquia em pré-ordem e, portanto, não se sabe de antemão o resultado de visibilidade no quadro atual dos filhos de cada nó.

Desse modo, toda vez que um nó é encontrado como visível, deve-se atualizar não somente a sua visibilidade, mas a visibilidade de todos os seus ancestrais. Esta técnica é chamada de *pull-up visibility*.



**Figura 4.4.3.1:** Instâncias da hierarquia no início e término de um quadro. Nota-se os *termination nodes* e o processo de *pull-up visibility* para atualizar a visibilidade dos nós internos para o quadro seguinte.

Em algum momento, as *queries* enviadas e armazenadas na fila devem ser processadas para que seus resultados sejam obtidos da GPU. Caso o nó seja invisível, nenhum processamento adicional precisa ser feito. Caso o nó seja visível, deve-se atualizar sua visibilidade, renderizar suas geometrias e continuar o caminhamento em suas sub-árvores.

Utiliza-se o seguinte algoritmo para processamento das *queries* já enviadas: entre a visita consecutiva de dois nós na hierarquia, verifica-se se a *occlusion query* enviada há mais tempo já está pronta. Caso ela ainda não tenha seu resultado disponível, continua-se com o caminhamento na hierarquia. Caso a *query* esteja pronta, interrompe-se temporariamente o caminhamento na hierarquia para que o resultado da *query* seja processado.

Neste momento, já se verifica se a próxima *query* da fila também está pronta. Se for o caso, continua-se processando as *queries* já enviadas até que a próxima a ser processada ainda não esteja disponível. Nesse momento, retorna-se para o caminhamento normal na hierarquia.

Pode-se dizer, então, que o algoritmo alterna o caminhamento na hierarquia com o processamento das *queries* já enviadas. Durante o caminhamento, são renderizadas geometrias previamente visíveis e efetuadas *queries* para os *termination nodes*. Esta alternância garante uma renderização das geometrias próximo a um *front-to-back* exato e elimina quase que completamente *CPU Stalls* e *GPU Starvation*. O pseudo-código a seguir ilustra uma implementação do algoritmo proposto.

```

while( !distanceQueue.empty() || !queryQueue.empty() )
{
    // Parte 1: processar queries pendentes para o quadro atual
    while( !queryQueue.empty() &&
        ( frontResultAvailable( queryQueue ) || !distanceQueue.empty() ) )
    {
        Node = queryQueue.front()

        // Obter resultado da occlusion query
        visiblePixels = getQueryResultFromOpenGL( Node )

        // Testar se esta visivel
        if( visiblePixels >= threshold )
        {
            pullUpVisibility( Node )

            // Evitar renderizar nós já renderizados neste frame uma segunda vez
            if( Node.lastRendered < framstamp )
            {
                Node.lastRendered = framstamp
                renderAndTraverse( Node )
            }
        }
    }
    // Parte 2: caminhar na hierarquia
    if( distanceQueue.empty() )
        continue

    // Obter próximo nó a ser visitado
    Node = distanceQueue.popTop()

    // Frustum Culling
    if( !insideViewFrustum( Node ) )
        continue

    // Interceptar o plano near pode causar resultados incorretos de visibilidade
    if( intersectsNearPlane( Node ) )
    {
        Node.lastVisited = framstamp
        Node.lastRendered = framstamp
        pullUpVisibility( Node )
        renderAndTraverse( Node )
    }
    else
    {
        wasVisible = Node.visible && ( Node.lastVisited == framstamp - 1 )
        Node.visible = false
        Node.lastVisited = framstamp

        if( wasVisible )
        {
            Node.lastRendered = framstamp

            if( !isLeaf( Node ) )
            {
                // Opened node ( nó interno previamente visível )
                renderAndTraverse( Node )
            }
            else
            {
                // Termination node ( folha previamente visivel )
                renderWithQuery( Node )
                queryQueue.push( Node )
            }
        }
        else
        {
            // Termination node ( nó previamente invisível )
            issueQueryForBoundingVolume( Node )
            queryQueue.push( Node )
        }
    }
}
}

```

#### 4.4.4 Melhorias

Mesmo levando-se em conta a inteligência do algoritmo proposto e suas inúmeras otimizações, ainda existem alguns aspectos a serem aprimorados.

Primeiramente, podem ocorrer situações em que o caminhamento na hierarquia já terminou mas ainda restam *queries* a serem processadas. Neste caso, não há como evitar *CPU Stalls*, pois a CPU deve esperar o resultado de todas as *queries* do quadro atual antes de poder prosseguir para o próximo. Caso estas situações estejam afetando a performance da aplicação, pode-se renderizar especulativamente as *queries* restantes.

Uma segunda proposta de melhoria procura resolver uma questão ainda significativa: a quantidade de *occlusion queries* efetuadas. Mesmo com o uso da hierarquia para diminuir este valor, dependendo da cena e da divisão espacial gerada o número de *queries* efetuadas pode ser tão grande ao ponto de degradar a performance da aplicação.

Para contornar este fato, propõe-se estimar a visibilidade no quadro atual de um nó previamente visível. Ou seja, assume-se que alguns nós previamente visíveis continuam de fato visíveis. Neste caso, apenas renderiza-se suas geometrias sem que sejam enviadas *queries* para a mesma. Uma implementação simples desta heurística envia *queries* para um mesmo nó a cada  $n$  quadros, onde  $n$  é uma constante especificada pelo usuário. Uma abordagem mais sofisticada pode levar em conta o histórico de visibilidade do nó em questão, o tamanho de sua projeção na tela, o movimento efetuado da câmera, dentre outros fatores.

Neste projeto, foram exploradas três abordagens para tentar evitar um número excessivo de *occlusion queries*.

##### 4.4.4.1 Evitar Testes Consecutivos

Esta heurística procura reduzir o número total de *queries* efetuadas ao designar para cada nó um contador. Quando o nó corrente era visível no quadro anterior, simplesmente incrementa-se este contador e renderiza-se as geometrias do nó sem efetuar-se uma *query*. Quando este contador atingir um valor especificado, renderiza-se as geometrias do nó efetuando-se também uma *occlusion query*. Atualiza-se a visibilidade do nó apenas de tempos em tempos.

#### 4.4.4.2 Probabilístico

Como normalmente os nós que estão visíveis variam pouco de um quadro para o próximo, a heurística anterior pode fazer com que de tempos em tempos ocorra um quadro com muitas *queries* sendo efetuadas. Para contornar este fato, ao invés de um contador, utiliza-se um gerador de números aleatórios para “adivinar” se um nó previamente visível continua visível. Assim, distribui-se as *queries* enviadas ao longo de vários quadros.

#### 4.4.4.3 Proximidade do Observador

Esta última heurística procura explorar o caminhamento na hierarquia em *front-to-back*. Em uma cena comum, a maioria dos objetos visíveis se encontram próximos do observador, enquanto os objetos invisíveis estão ocultos por detrás destes. Levando-se em conta este fato, pode-se assumir que as primeiras folhas a serem visitadas corresponderão a geometrias que estarão de fato visíveis na cena.

De maneira simples, a heurística consiste em evitar o envio de *occlusion queries* para as  $n$  primeiras folhas previamente visíveis que são visitadas pelo algoritmo de caminhamento na árvore. Esta constante é especificada pelo usuário e depende fortemente da cena a ser visualizada.

## 5 Especificação do Sistema

Como já mencionado anteriormente, o segundo objetivo deste projeto é o desenvolvimento de uma biblioteca com o intuito de implementar a técnica de otimização escolhida, juntamente com seus estudos relacionados. Esta biblioteca deve ser capaz de ser integrada facilmente à aplicações de visualização já existentes.

Além disso, é fundamental que esta biblioteca seja desenvolvida com a maior coesão e o menor acoplamento possível. Por exemplo, uma aplicação pode estar interessada apenas no uso dos algoritmos de construção de volumes envolventes, ou então apenas utilizar os módulos de construção da hierarquia para obter um agrupamento espacial de seus objetos.

O código foi inteiramente desenvolvido em C++, fazendo uso de técnicas de programação modular e orientação a objetos. Quando possível, foram utilizados *design patterns* [27] como *abstract factories* para centralizar a implementação de certas partes do código e facilitar sua posterior extensão. Desse modo, procurou-se preparar a biblioteca desenvolvida para servir como base para futuros estudos e pesquisas de novas propostas de algoritmos.

A API gráfica utilizada foi o OpenGL, padrão da indústria e das diferentes áreas de pesquisa em Computação Gráfica. O código foi desenvolvido em ambiente Windows XP Professional e compilado com o Microsoft Visual Studio .NET 2003. Além disso, foi dada atenção para se ater ao padrão ISO C++ mais recente, ISO/IEC 14882:2003.

Por não possuir nenhuma dependência específica com o sistema operacional, acredita-se que o código possa ser portado e compilado em outras plataformas, como Linux, sem maiores dificuldades. Basta obter suas bibliotecas dependentes também na plataforma desejada.

### 5.1 Arquitetura

A biblioteca desenvolvida pode ser subdividida nos componentes lógicos a seguir. Cada componente é composto em média por dois módulos responsáveis por prover as funcionalidades enunciadas.

- Obtenção de geometrias da aplicação

Responsável por definir uma API com a aplicação cliente de modo a traduzir as geometrias da cena para uma representação interna da biblioteca. Esta representação é utilizada pelos algoritmos de construção de hierarquia e de renderização da cena.

- Construção de caixas envolventes

Implementa os algoritmos de construção de caixas envolventes desenvolvidos. Inclui algoritmos de construção de um fecho convexo de um conjunto de vértices. Utiliza *abstract factories* para que novos algoritmos de construção de caixas sejam facilmente adicionados à biblioteca.

- Construção da hierarquia de divisão espacial

Implementa os algoritmos de construção de uma hierarquia, customizáveis através de classes de heurísticas e de algoritmos de particionamento espacial implementadas de forma extensível.

- Algoritmos de análise e cálculo geométricos

Possui módulos independentes de cálculo de distância e interseção entre entidades geométricas, além de algoritmos de análise estatística de um conjunto de vértices e cálculo da área de superfície de caixas.

- Iteradores e caminhamento simples da hierarquia

Foram desenvolvidos iteradores para facilitar o caminhamento na hierarquia utilizada. As classes de caminhamento utilizam estes iteradores para prover uma API baseada em *callbacks* para simplificar a visita aos nós da hierarquia.

- Suporte à renderização da hierarquia

Para os testes efetuados, foram desenvolvidos dois módulos. O primeiro é responsável por gerar o agrupamento de renderização de cada nó da hierarquia, através da construção de *Display Lists*. O segundo sintetiza e centraliza todos os algoritmos de renderização e otimização implementados, de modo que estes possam ser escolhidos e configurados facilmente pela aplicação cliente.

- Algoritmos de *Frustum Culling*

Implementa classes para representar o volume de visão e computar os testes de visibilidade de um nó, através da interseção de seu volume envolvente com os planos do *frustum*. Além disso, disponibiliza o algoritmo de caminhamento em uma hierarquia implementando o *Frustum Culling* hierárquico.

- Algoritmos de *Occlusion Culling*

Implementa o algoritmo de otimização descrito anteriormente, tendo suporte à integração do *Frustum Culling* ou outro critério para que um nó possa ser visitado através do uso de *callbacks*.

Os componentes que efetuam caminhamento na hierarquia e implementam os algoritmos de *Frustum Culling* e *Occlusion Culling* foram desenvolvidos de forma a receber da aplicação cliente *callbacks* que são chamadas para cada nó visitado. Estas *callbacks* podem ser tanto ponteiros para funções quanto *functors* e são responsáveis, no caso da visualização, pela renderização cada um dos nós visitados. Além disso, esta implementação possibilita o uso destes algoritmos para coleta de nós para posterior processamento por parte da aplicação cliente, dando maior flexibilidade para seu uso em diferentes situações.

## 5.2 API Desenvolvida

Existem duas etapas principais para utilização desta biblioteca: a etapa de pré-processamento e a etapa de visualização.

Caso a aplicação deseje apenas utilizar a biblioteca para visualização, tirando proveito dos algoritmos e otimizações implementados, basta utilizar o módulo chamado *SceneBuilder*. Esta classe torna transparente para a aplicação as etapas de processamento internas da biblioteca. Ela provê dois objetos: uma instância da classe *SceneData* e uma instância da classe *BspTreeBuilder*.

A aplicação utiliza a classe *SceneData* para informar à biblioteca as geometrias que compõem a cena a ser visualizada. Estes dados são fornecidos utilizando-se tipos básicos da linguagem, para facilitar a adaptação da biblioteca às mais variadas aplicações. Cada tipo de geometria possui sua respectiva interface de renderização, fornecida pela aplicação. Para isso, basta implementar uma classe de interface abstrata chamada *IRender*.

```
// Reset input for new geometries
void initGeometryInput();

///////////////////////////////
// Begin new geometry
void beginGeometry( unsigned int id, IRender* renderInterface );

void addVertex( const float* vertex );
void addVertex( const double* vertex );

void addVertices( const float* vertices, unsigned int size );
void addVertices( const double* vertices, unsigned int size );

void transformVertices( const double* matrix );

const VertexContainer& currentVertices();

void endGeometry();
// End current geometry
/////////////////////////////
```

API principal da classe *SceneData*.

A aplicação também pode utilizar a classe *BspTreeBuilder* caso deseje configurar as heurísticas de construção da hierarquia. Além disso, o desenvolvedor da aplicação possui a opção de especializar uma classe de heurísticas, chamada *BspHeuristics*, de modo a implementar seus próprios algoritmos de divisão espacial e construção da hierarquia. Assim, pode-se estender a biblioteca para utilização dos mais variados tipos de divisão espacial.

```
void setHeuristics( BspHeuristics* heuristics );
BspHeuristics* heuristics();

Node* createHierarchy( ConstructionNode* root );

const Statistics& statistics() const;
```

API principal da classe *BspTreeBuilder*.

Finalmente, podem ser adicionadas *factories* para implementar algoritmos de construção de caixas envolventes. Para isto, utiliza-se a classe *BoxFactory*.

```
// Use active box type factory, if available.
static Box* CreateBox( const double* vertices, unsigned int size );

// Use specific box type factory, if available.
static Box* CreateBox( const double* vertices,
                      unsigned int size, unsigned int boxType );

// Box type to use either when no factory is found or
// when input vertices may be degenerated.
static void SetDefaultBoxType( unsigned int boxType );
static unsigned int DefaultBoxType();

// Box type to use when none provided.
static void SetActiveBoxType( unsigned int boxType );

// Next available box type that can be registered.
static unsigned int NextAvailableBoxType();

// Returns true if a factory for given boxType already exists.
static bool HaveFactory( unsigned int boxType );

// Register box factory prototype.
// Warning: will overwrite any existing prototypes for given type!
static void RegisterPrototype( AbstractBoxFactory* prototype,
                               unsigned int boxType );
```

API principal da classe *BoxFactory*.

Após informar as geometrias através da instância da classe *SceneData* e configurar as heurísticas de construção da hierarquia de divisão espacial, através da instância da classe *BspTreeBuilder*, a aplicação fornece à classe *SceneBuilder* uma instância da classe *SceneRenderer*.

Esta classe é responsável por implementar todos os algoritmos de otimização de visualização de forma transparente para o desenvolvedor. A classe *SceneBuilder* utiliza as informações fornecidas pela aplicação para configurar corretamente a instância da classe *SceneRenderer*.

Neste momento, será construída a hierarquia de divisão espacial desejada, utilizando-se as heurísticas especificadas e os algoritmos de construção de caixas envolventes selecionados. Por padrão, as estatísticas de construção da hierarquia poderão ser obtidas na saída padrão do sistema (*stdout*). Esta chamada marca o fim da etapa de pré-processamento.

```
// Use this to input geometry render interfaces and vertex data.
SceneData* sceneData();

// Use this to configure hierarchy construction heuristics.
BspTreeBuilder* bspTreeBuilder();

// Should be called when ready to build scene.
// Should have a valid OpenGL context. This ensures stability and correct behavior.
// Creates hierarchy based on geometry input gathered by SceneData,
// Using heuristics set in BspTreeBuilder.
// From this hierarchy, creates clusters (displayLists)
// for rendering groups of geometries in leaf nodes.
// Sets the hierarchy and corresponding clusters to given SceneRenderer.
// Finally, initializes any OpenGL dependencies left.
// SceneRenderer is ready to be used.
bool setupSceneRenderer( SceneRenderer* sceneRenderer );
```

#### API principal da classe *SceneBuilder*.

Pode-se então iniciar a etapa de visualização: a aplicação utiliza a classe *SceneRenderer* para facilmente visualizar a cena. Para isso, basta chamar seu método *render*. Além de fornecer estatísticas de renderização, a classe *SceneRenderer* provê uma interface simples para seleção dos algoritmos de renderização que devem ser utilizados.

## 6 Testes e Resultados

Para validar os estudos efetuados e as técnicas implementadas, foram efetuados uma série de testes sistemáticos. Estes tem por objetivo avaliar a qualidade e o desempenho das propostas desenvolvidas, bem como analisar as relações entre cada uma e a performance da aplicação de visualização.

Através destes testes, será possível concluir o estudo iniciado por este projeto e abrir caminho para novas propostas e melhorias futuras.

Todos os testes foram efetuados em um PC desktop com processador Athlon 64 XP 3800+ de 2.4 Ghz, 2 Gb de RAM DDR operando em *Dual-Channel* a 800Mhz. A placa de vídeo utilizada foi uma nVidia GeForce 7800 GT 256Mb em barramento PCI-Express.

### 6.1 Cenas de Teste

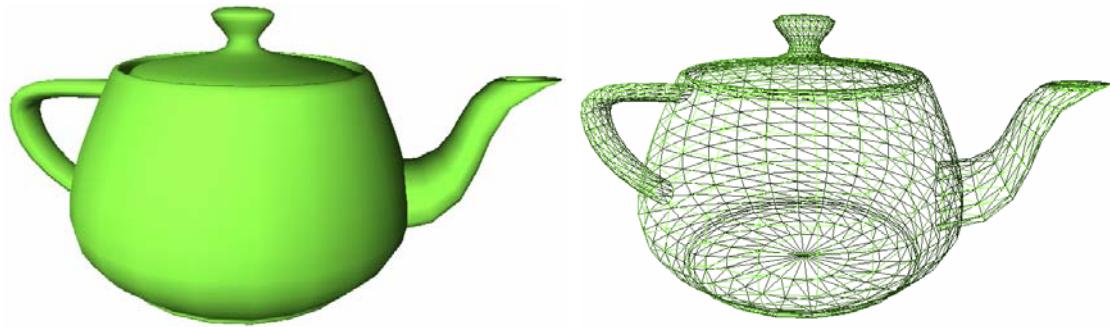
A biblioteca desenvolvida foi avaliada segundo duas vertentes principais: uma que procura analisar diretamente seus algoritmos implementados, e outra que visa analisar um caso de uso da implementação em uma aplicação real.

Para cada uma das cenas testadas, foi construído um caminho a ser percorrido pelo observador da cena de forma automática pela aplicação. Estes percursos procuraram variar a quantidade de geometrias visualizadas a cada ponto de vista, bem como seu grau de oclusão.

No primeiro caso, foi desenvolvida uma aplicação capaz de gerar uma cena de teste artificial. Esta cena consiste em um número pré-determinado de *Teapots*<sup>19</sup>, posicionados e rotacionados aleatoriamente em um volume no espaço. Este volume possui a forma de um paralelepípedo alinhado com os eixos e de dimensões fixas. Cada *Teapot* é composto essencialmente por 1364 vértices, mas para correta visualização devem ser enviados para a placa gráfica 2781 vértices utilizando-se índices.

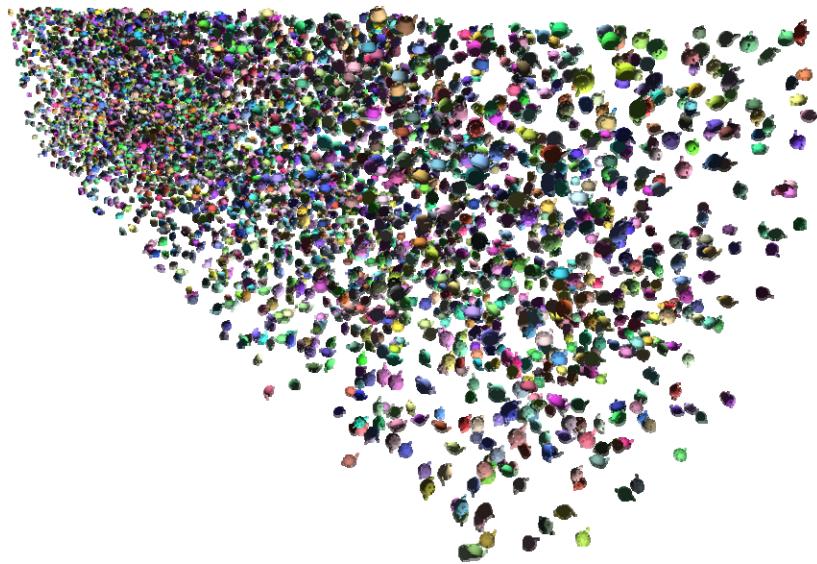
---

<sup>19</sup> O uso de *Teapots* como *benchmark* remete aos primórdios da Computação Gráfica. O leitor é convidado a pesquisar suas origens.



**Figura 6.1.1:** *Teapot* utilizado nos testes.

Pode-se variar não somente a quantidade de *Teapots* a serem visualizados, mas também sua escala individual. Com isso, pode-se obter uma cena com objetos grandes ou pequenos, alterando-se assim seu grau de oclusão. Esta cena de teste será referida nas próximas seções como **Teapots(*n,s*)**, onde ***n*** é o número de objetos que compõem a cena e ***s*** é o fator de escala aplicado a cada um.

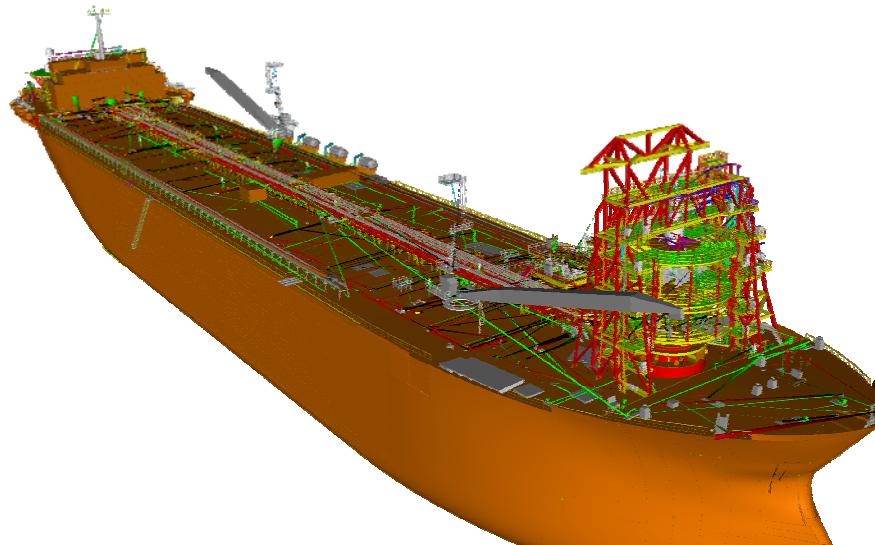


**Figura 6.1.2:** Cena de testes dos *Teapots*.

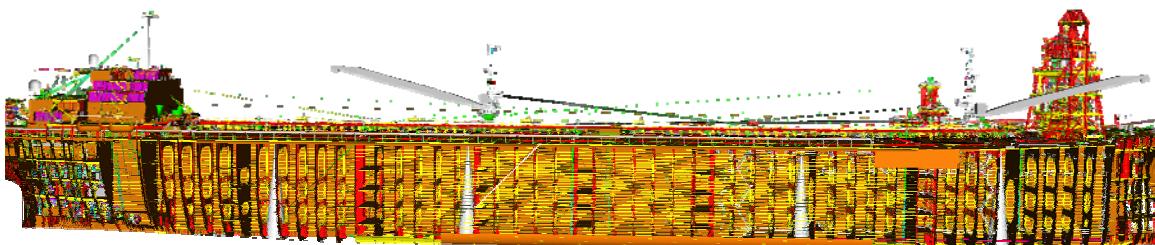
O segundo tipo de cena de teste foi desenvolvido em cima de uma biblioteca de visualização conhecida por *OpenSceneGraph* [27]. Esta consiste em um grafo de cena *open-source* com inúmeras aplicações, capaz de visualizar diferentes tipos de modelos. Existem vários produtos comerciais e de pesquisa que utilizam esta biblioteca como *engine* de renderização, tendo provado não somente sua flexibilidade mas também sua performance de visualização.

A adaptação de uma cena do *OpenSceneGraph* para o uso com a biblioteca desenvolvida foi simples: implementou-se uma interface de renderização para seu tipo específico de geometria e estas foram coletadas e repassadas para a biblioteca deste projeto. O uso do *OpenSceneGraph* traz consigo a vantagem de se poder utilizar modelos CAD reais, fornecidos por intermédio do Laboratório de Tecnologia em Computação Gráfica da PUC-Rio, ou Tecgraf. Estes modelos são oriundos de uma parceria do laboratório com a empresa Petróleo Brasileiro S/A, ou Petrobras.

Para os testes neste projeto, foi utilizado o modelo CAD de uma plataforma de petróleo chamada **P-38**. Este modelo possui 90.867 objetos, representados por um total de 4.717.749 vértices.



**Figura 6.1.3:** Cena de teste da plataforma de petróleo P-38.



**Figura 6.1.4:** Observa-se a complexidade geométrica no interior do modelo.

As análises dos volumes envolventes tem como propósito quantificar o desempenho e a qualidade de seus respectivos algoritmos. Esta avaliação não irá considerar o impacto destes algoritmos no desempenho da aplicação de visualização. Mais adiante, quando forem analisados os algoritmos de *Frustum Culling* e *Occlusion Culling* serão obtidas medidas de performance da aplicação final.

## 6.2 Volumes Envolventes

Cada volume envolvente foi avaliado segundo os seguintes parâmetros: seu tempo de construção e sua área de superfície. O primeiro procura identificar a escalabilidade dos algoritmos e o segundo é uma medida do ajuste das caixas construídas. Os valores de tempo são a média em **segundos** de três execuções de cada algoritmo. As áreas não possuem unidade específica, sendo expressadas em unidades da cena.

Foram desenvolvidos, ao todo, nove algoritmos de construção de caixas envolventes. As propostas implementadas seguem a seguinte nomenclatura:

- **AABB:** Algoritmo padrão de construção de uma AABB.
- **Covar:** Utiliza somente a análise de média e covariância sobre os vértices das geometrias para construção de uma OBB.
- **Hull\_Covar:** Primeiro, constrói-se o fecho convexo das geometrias a serem envolvidas e, depois, utiliza-se diretamente a média e covariância sobre os vértices do fecho, como se o mesmo fosse uma geometria da cena.
- **Hull\_Unif\_1:** Primeira proposta de melhoria utilizando o fecho convexo e uma amostragem uniforme sobre sua superfície para calcular-se a média e covariância.
- **Hull\_Unif\_2:** Segunda proposta de melhoria utilizando o fecho convexo e uma amostragem uniforme sobre sua superfície para calcular-se a média e covariância.
- **Hull\_Unif\_3:** Terceira proposta de melhoria utilizando o fecho convexo e uma amostragem uniforme sobre sua superfície para calcular-se a média e covariância.
- **Min\_Aprox:** Aproximação simples da caixa mínima utilizando um algoritmo iterativo de aproximações do diâmetro dos vértices.

- **Min\_Aprox\_GA:** Aproximação da caixa mínima utilizando um *grid* a partir de uma amostragem dos vértices das geometrias a serem envolvidas.
- **Min\_Aprox\_G:** Aproximação da caixa mínima utilizando um *grid* a partir de todos os vértices a serem envolvidos.

	Teapots(1,10)		Teapots(5,10)		Teapots(500,10)		Teapots(5000,10)	
	tempo	área	tempo	área	tempo	área	tempo	área
<b>AABB</b>	0,00	27,88	0,00	6.306,58	0,15	10.623,51	1,59	10.835,00
<b>Covar</b>	0,00	27,48	0,00	3.304,90	0,17	13.117,38	1,78	13.928,76
<b>Hull_Covar</b>	0,02	26,97	0,12	3.657,13	17,92	13.413,87	164,91	15.149,93
<b>Hull_Unif_1</b>	0,02	27,65	0,13	5.652,73	17,99	11.927,99	164,78	13.253,46
<b>Hull_Unif_2</b>	0,02	27,41	0,12	3.426,37	17,88	13.298,16	166,89	12.786,44
<b>Hull_Unif_3</b>	0,02	25,71	0,12	13.897,29	17,85	31.878,56	164,98	36.239,65
<b>Min_Aprox</b>	0,02	22,84	0,10	3.069,29	14,68	10.581,23	206,01	10.802,03
<b>Min_Aprox_GA</b>	2,19	22,71	2,70	3.033,52	102,51	10.581,15	1.351,43	10.802,03
<b>Min_Aprox_G</b>	7,35	22,70	42,47	3.030,00	6.886,68	10.581,23	-	-

Tabela 6.2.1: Algoritmos de construção de caixas envolventes.

As primeiras duas cenas, **Teapots(1,10)** e **Teapots(5,10)** tem como objetivo principal avaliar o ajuste das diferentes caixas envolventes construídas. Já as duas últimas cenas, **Teapots(500,10)** e **Teapots(5000,10)**, procuram medir principalmente a escalabilidade das soluções desenvolvidas.

Analizando-se primeiramente o tempo de construção das caixas, conclui-se que a implementação **Covar** é praticamente equivalente à **AABB**. Em seguida, surge a proposta **Min\_Aprox**, que somente é superada pelas baseadas em covariância na cena **Teapots(5000,10)**. Em todas as cenas, as propostas **Min\_Aprox\_GA** e **Min\_Aprox\_G** se provaram as mais lentas. Inclusive, não foi possível obter-se as medidas desta última referentes à cena **Teapots(5000,10)**.

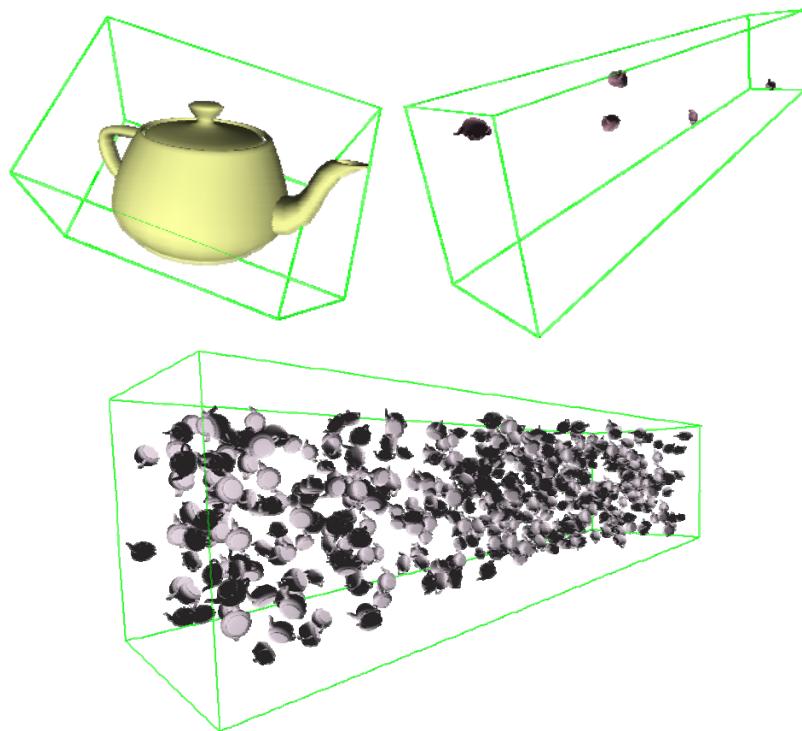
O interesse maior na pesquisa destes algoritmos é analisar o ajuste das caixas construídas. Como sua construção se dá em uma etapa de pré-processamento, pode se dar ao luxo de optar por um algoritmo mais lento, contanto que suas caixas sejam significativamente mais bem-ajustadas.

Como as geometrias são posicionadas aleatoriamente no interior de um paralelepípedo alinhado com os eixos coordenados, observa-se que uma elevada quantidade de geometrias na cena favorece uma distribuição ao longo dos eixos. Este fato é explorado pela **AABB** por sua definição, obtendo assim um ajuste excelente para as duas últimas cenas. Somente o algoritmos baseados na aproximação da caixa mínima são capazes de superá-la. As demais propostas baseadas em covariância, são todas afetadas negativamente pela distribuição espacial dos vértices envolvidos.

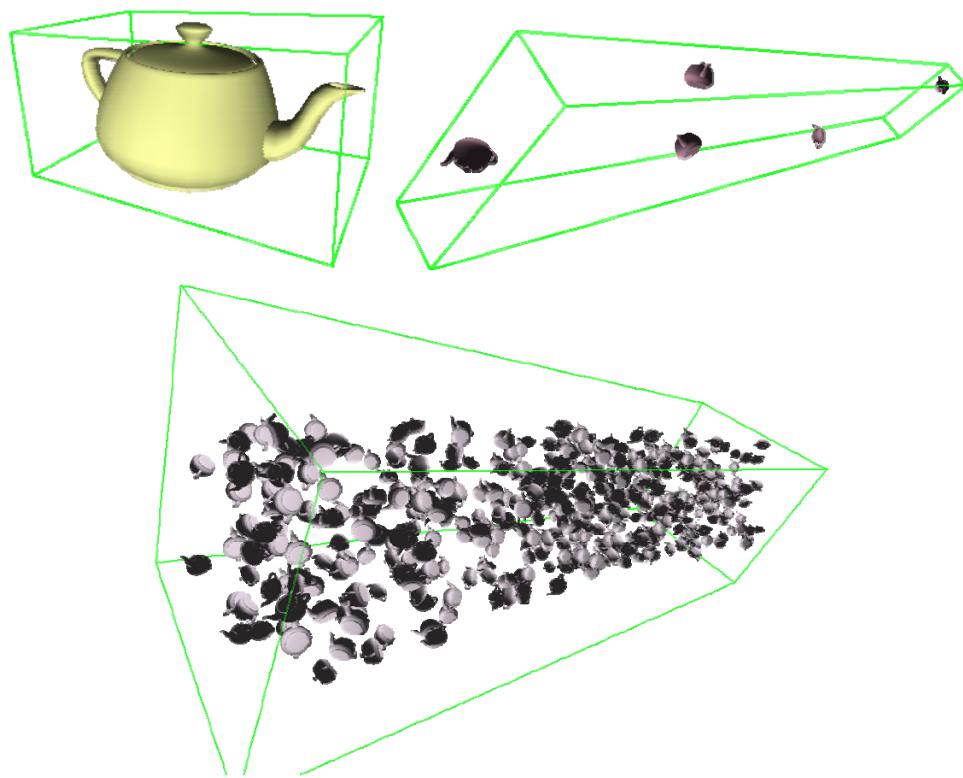
Segundo um critério de custo x benefício, o algoritmo **Min\_Approx** possui tempo de construção equivalente às propostas de fecho convexo, mas conseguiu obter um ajuste muito superior do que quase todas as outras propostas. Esta técnica, portanto, se provou superior a todas as demais.

As duas últimas linhas correspondem a algoritmos de aproximação da caixa mínima que, embora consigam fornecer um ajuste ainda melhor do que os outros, apresentam uma relação custo x benefício muito além das outras propostas.

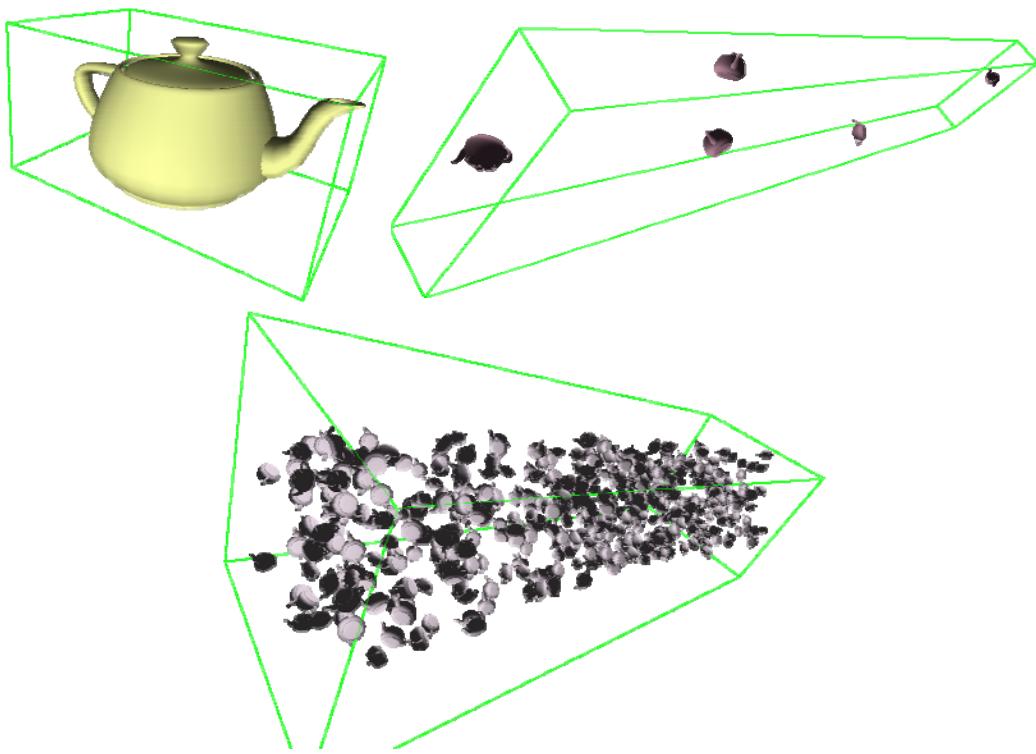
As imagens a seguir ilustram as caixas envolventes obtidas para as cenas **Teapots(1,10)**, **Teapots(5,10)** e **Teapots(500,10)**. A imagem de cada cena foi obtida sob o mesmo ponto de vista. Assim, pode-se observar as diferentes caixas encontradas por cada algoritmo.



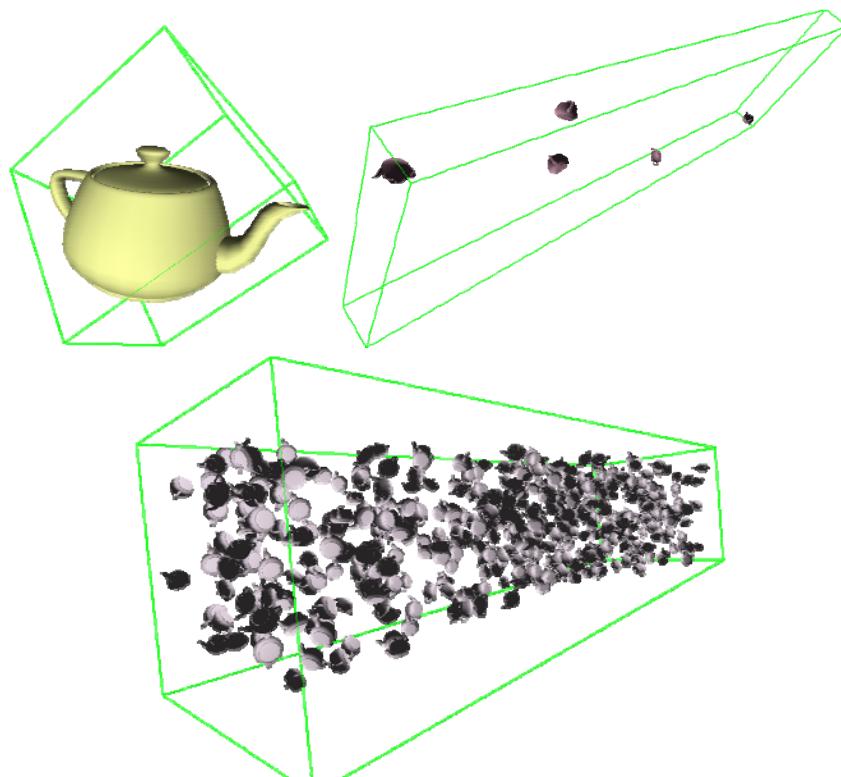
**Figura 6.2.1:** Algoritmo **AABB**. Um ajuste ruim particularmente na segunda cena.



**Figura 6.2.2:** Algoritmo **Covar**. Muitos objetos influenciam negativamente a covariância.



**Figura 6.2.3:** Algoritmo **Hull\_Covar**. Ainda há influência negativa de muitos objetos.



**Figura 6.2.4:** Algoritmo **Hull\_Unif\_1**. A amostragem do fecho convexo melhorou o ajuste na terceira cena.

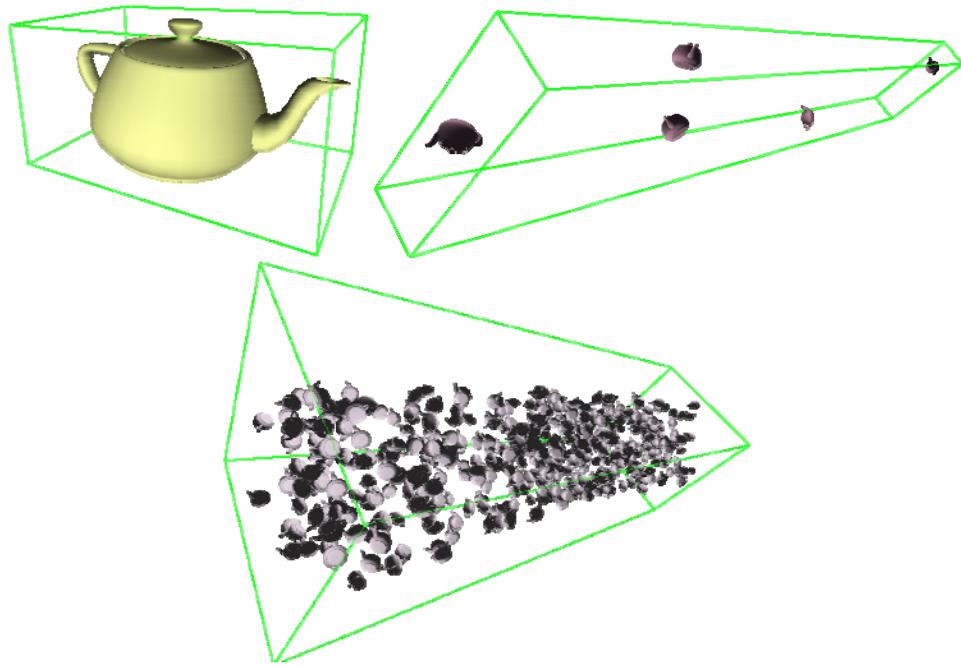


Figura 6.2.5: Algoritmo **Hull\_Unif\_2**. Ainda afetado pela distribuição dos objetos no espaço.

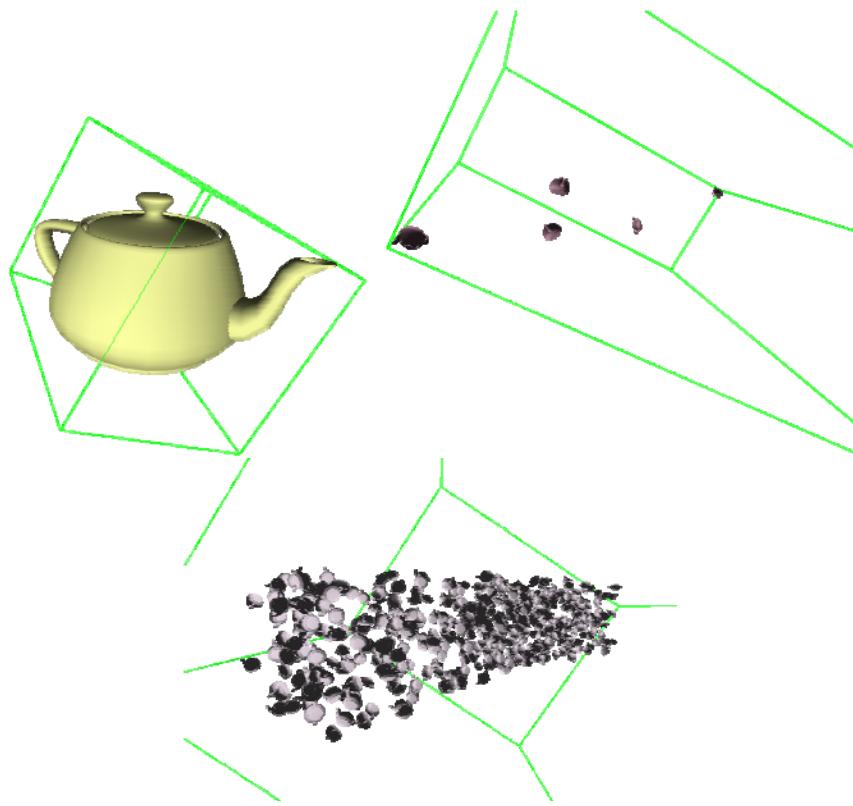
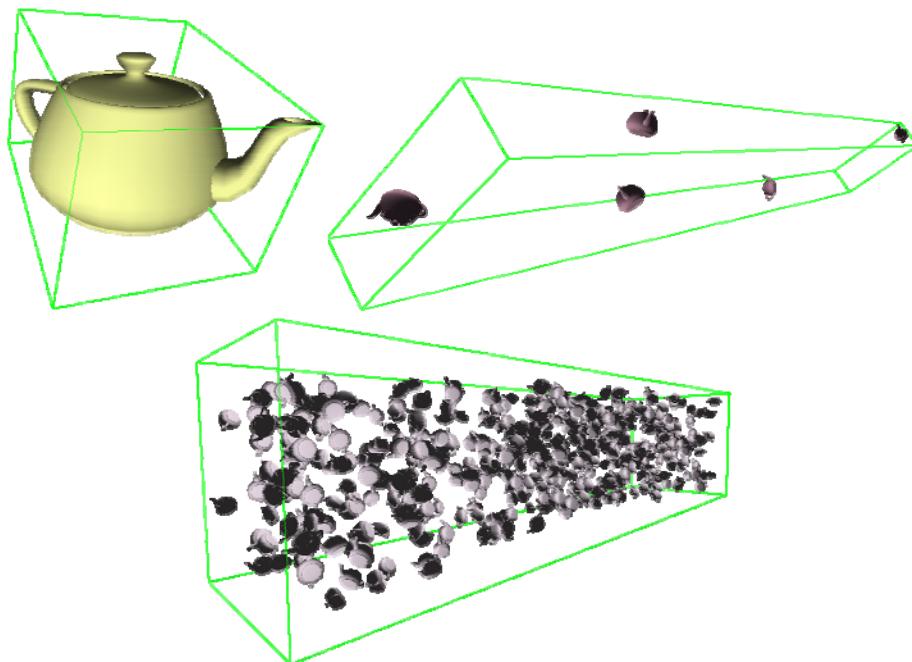
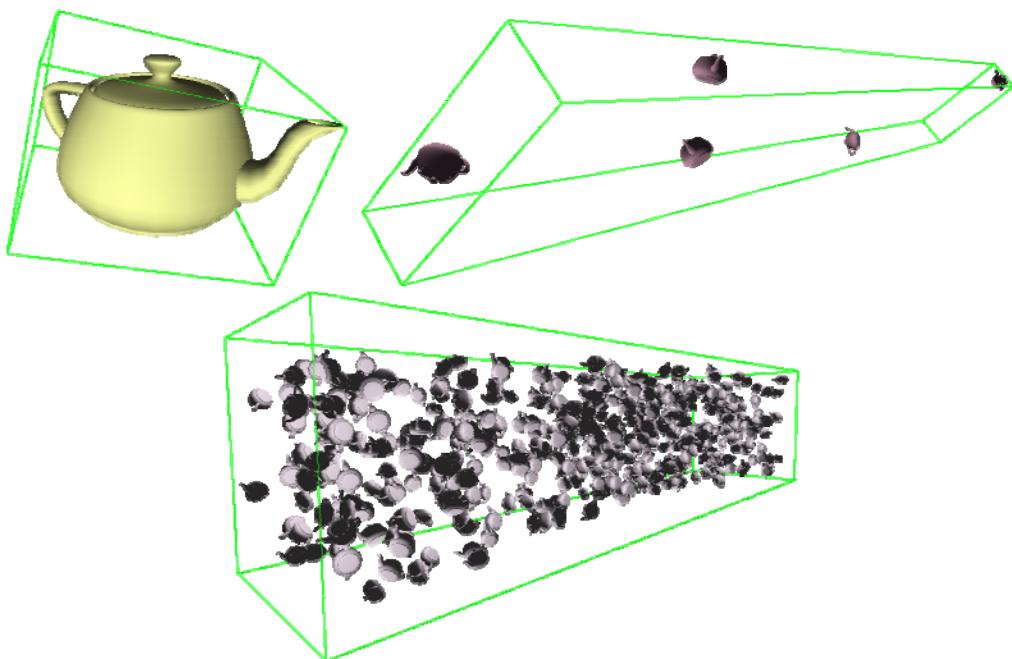


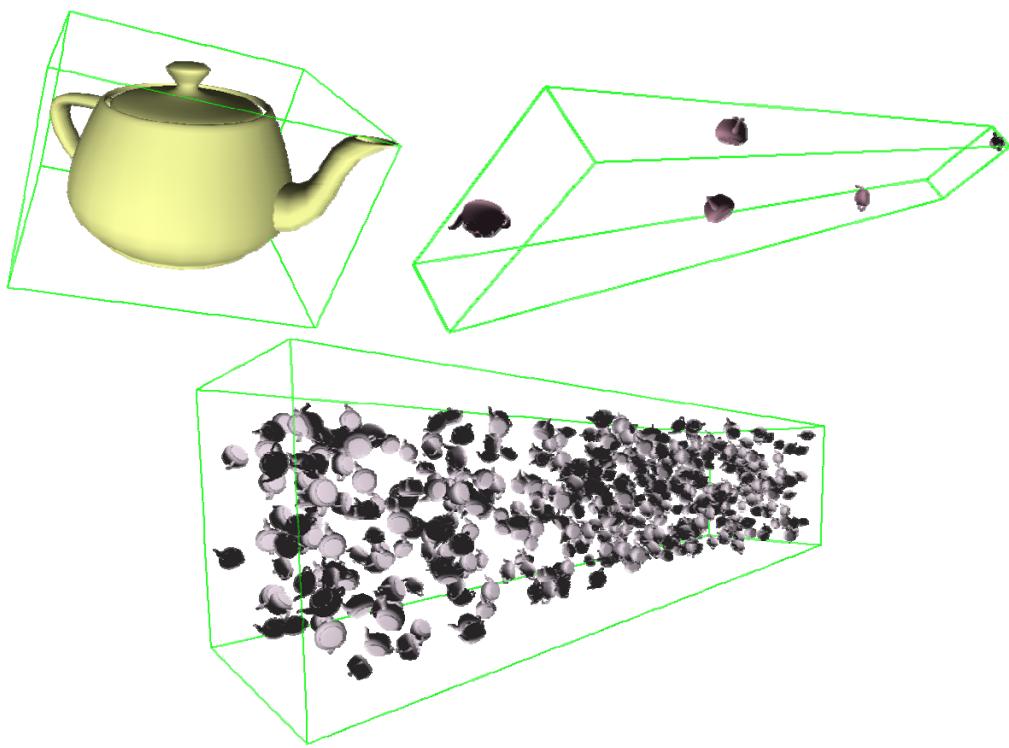
Figura 6.2.6: Algoritmo **Hull\_Unif\_3**. Caixas tão ruins que nem aparecem totalmente em cada imagem.



**Figura 6.2.7:** Algoritmo Min\_Approx. Observa-se um ajuste diferente, e melhor, na primeira cena.



**Figura 6.2.8:** Algoritmo Min\_Approx\_GA. Um ajuste ainda melhor nas cenas.



**Figura 6.2.9:** Algoritmo **Min\_Approx\_G**. Grande aumento no tempo de processamento em troca de uma pequena melhoria em um ajuste que já era bom.

### 6.3 Occlusion Queries

Antes que a proposta de *Coherent Hierarchical Culling* fosse desenvolvida em detalhes, efetuou-se um estudo mais aprofundado na extensão de *occlusion queries*. Neste, foi analisado o impacto do uso de *queries* no desempenho de uma aplicação de visualização simples, sem uso de hierarquia. Todos os tempos estão em **milissegundos** e seus valores são a média de 1000 amostragens.

O primeiro programa de testes desenvolvido simplesmente efetua uma única *query* para a cena inteira. Após o envio deste teste, foram efetuados diferentes procedimentos. Variando-se a quantidade de objetos, pode-se analisar como a complexidade da geometria a ser testada para oclusão influencia o desempenho de uma *occlusion query*. A tabela 6.3.1 a seguir apresenta os resultados obtidos. As colunas seguem a seguinte nomenclatura:

- **tTesteDisp:** Tempo consumido por apenas um teste de disponibilidade da *query* efetuada naquele mesmo quadro.
- **tDisp:** Tempo para que a *query* efetuada se torne disponível, medido evitando-se que a aplicação inicie a renderização do quadro seguinte.
- **tDispObt:** Tempo anterior somado ao tempo de obtenção do resultado de visibilidade, ou seja, do número de *pixels* gerados.
- **nEsperas:** Número de iterações apenas testando-se se o resultado da *query* efetuada já se encontrava disponível.

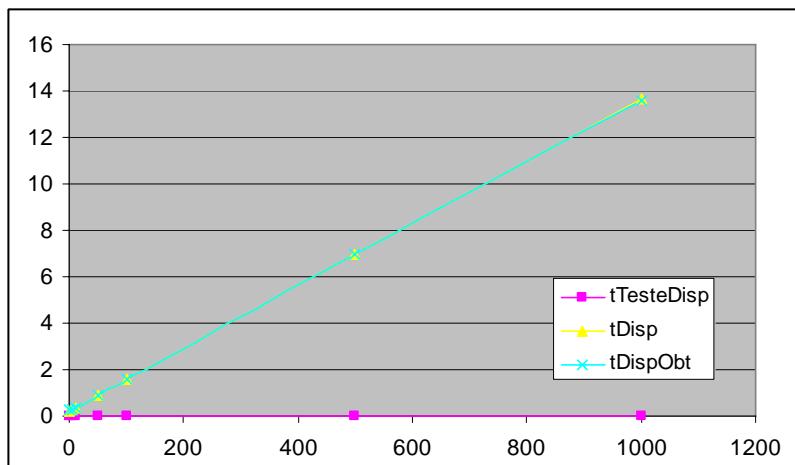
No. Teapots	tTesteDisp	tDisp	tDispObt	nEsperas
1	0,00176	0,24	0,24	495
5	0,00181	0,29	0,29	598
10	0,00183	0,36	0,36	735
50	0,00180	0,88	0,88	1.820
100	0,00175	1,56	1,56	3.214
500	0,00176	6,99	6,99	14.452
1000	0,00177	13,67	13,62	28.196

Tabela 6.3.1: Uma única *query* para a cena inteira.

Observando-se a coluna **tTesteDisp**, pode-se concluir que o custo do teste para se determinar se uma *occlusion query* já terminou e possui resultado disponível é extremamente baixo, sendo aproximadamente  $1.8 \times 10^{-6}$  segundos. Normalmente, uma aplicação irá efetuar um número de testes próximo ao número de *queries* efetivamente enviadas. Isto porque com o passar do tempo, as *queries* efetuadas anteriormente tornam-se disponíveis. Portanto, o custo de testar se uma *query* está disponível pode ser considerado desprezível.

A última coluna **nEsperas** ilustra a quantidade de processamento que poderia ser feito no intervalo de tempo entre o envio de uma *query* e seu resultado encontrar-se disponível. Mais uma vez, comprova-se a necessidade de se ocupar a *CPU* entre uma *query* e a seguinte.

Comparando-se as colunas **tDisp** e **tDispObt**, conclui-se que não há overhead em obter-se o resultado de uma *occlusion query* que já esteja pronta. Além disso, pode-se observar que o tempo para cada teste é diretamente proporcional à complexidade da geometria sendo testada. Isto faz sentido já que para se determinar a visibilidade deve-se efetivamente renderizar a geometria em questão. O gráfico 6.3.1 construído a partir dos valores da tabela 6.3.1 indica que esta relação é de fato linear.



**Gráfico 6.3.1:** O tempo para uma *occlusion query* se tornar disponível é linearmente proporcional à complexidade da geometria sendo testada.

O segundo programa de testes procura avaliar o custo em se efetuar uma *occlusion query* comparado ao custo de somente renderizar-se uma dada geometria. Para isso, foram obtidas as seguintes medidas:

- **tRender:** Tempo gasto para renderização de todas as geometrias sem nenhum teste de visibilidade.
- **tUmaQuery:** Tempo gasto para renderização de todas as geometrias efetuando-se apenas um teste de visibilidade para todas.
- **tVariasQueries:** Tempo gasto para renderização de todas as geometrias, efetuando-se uma *occlusion query* para cada uma.

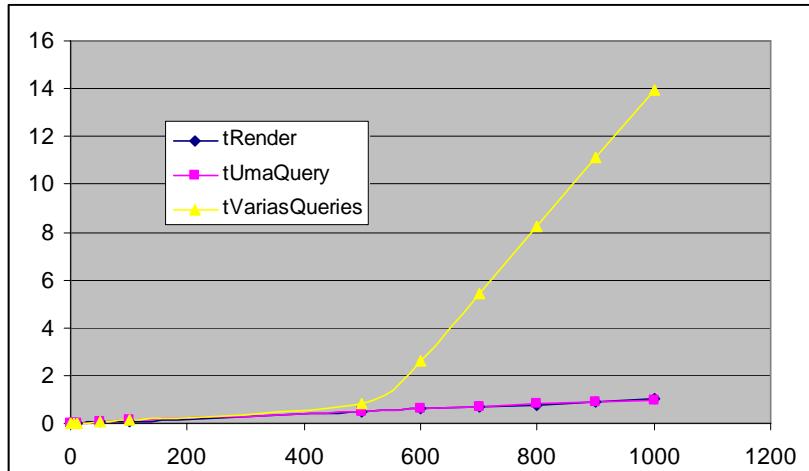
No. Teapots	tRender	tUmaQuery	tVariasQueries
1	0,003	0,005	0,005
5	0,008	0,009	0,011
10	0,013	0,014	0,019
50	0,052	0,055	0,085
100	0,103	0,106	0,163
500	0,503	0,508	0,808
600	0,598	0,610	2,606
700	0,698	0,703	5,415
800	0,789	0,791	8,270
900	0,910	0,898	11,092
1000	0,997	0,995	13,915

Tabela 6.3.2: Overhead para envio de uma ou mais *occlusion queries*.

Os valores das colunas **tRender** e **tUmaQuery** indicam que o custo adicional de efetuar-se uma *query* ao mesmo tempo em que se renderiza uma dada geometria é muito pequeno, em torno de  $2 \times 10^{-6}$  segundos. Porém, não se pode concluir prontamente que este *overhead* seja fixo de acordo com a quantidade de *queries* efetuadas por quadro.

Para comprovar este fato, basta observar a coluna **tVariasQueries**. Seus valores indicam que quanto mais *occlusion queries* são efetuadas em um mesmo quadro, maior será o custo equivalente de cada uma. Por exemplo, enquanto efetuar-se uma única *query* para 500 *Teapots* custa 0.508 milissegundos, se cada um dos 500 *Teapots* for testado separadamente o custo total das *queries* efetuadas aumenta para 0.808 milissegundos, cerca de 37% maior. Isto confirma a estratégia de agrupamento de geometrias como forma eficiente de utilização das *occlusion queries*.

Deve-se atentar para as linhas correspondentes a 500 e 600 *Teapots*. Comparando-se os valores correspondentes à coluna **tVariasQueries**, observa-se um súbito aumento no custo de efetuar-se a renderização juntamente com uma *occlusion query*. Este fato é claramente evidenciado no gráfico 6.3.2 correspondente.

Grafico 6.3.2: A partir de cerca de 500 *occlusion queries* por quadro, seu *overhead* torna-se cada vez mais significativo.

Chama-se a atenção para que uma aplicação executando a 30 fps gasta em média 33ms em cada quadro. Estes valores destacados na tabela podem, potencialmente, prejudicar o desempenho da visualização. Comprova-se, então, a necessidade de controlar o número de *queries* enviadas por quadro. No caso ideal, não se deve exceder o patamar entre 500 e 600 *queries* a cada quadro.

Uma observação final sobre o uso de *occlusion queries* é o fato de que existe um limite para a quantidade de *query objects* construídos e utilizados. Na aplicação de teste anterior, experimentou-se a criação de um novo *query object* para cada nova *query* a ser efetuada. Com isso, observou-se uma redução drástica no desempenho da visualização com o uso de cerca de 260.000 *query objects*. Neste momento, o número de quadros por segundo diminuiu rapidamente de 150 fps para menos do que 1 fps.

## 6.4 Visualização

Nesta etapa, será analisado o desempenho tanto dos algoritmos de construção de hierarquia quanto das técnicas de otimização para visualização implementadas. Como já mencionado, a análise dos algoritmos de renderização será efetuada através de caminhos pelos quais o observador percorrerá a cena. Os valores de tempo são a média em **segundos** de três execuções de cada algoritmo.

Cada técnica de otimização terá como base de comparação um algoritmo de renderização que sempre percorre toda a hierarquia, sem nenhum tipo de otimização. Como normalmente o gargalo da aplicação se encontra na GPU, a diferença no tempo de caminhamento em diferentes hierarquias não deve causar nenhum impacto significativo na performance deste algoritmo.

Todos os testes de *Occlusion Culling* utilizaram também o algoritmo de *Frustum Culling* para efetuar um primeiro descarte de geometrias fora do volume de visão. Assim, pode-se avaliar a contribuição exata para a performance de renderização do algoritmo de descarte por oclusão. O algoritmo de *Frustum Culling* utilizará a notação **vfc** enquanto o algoritmo de *Coherent Hierarchical Culling* será referido por **chc**.

Cada heurística de construção de hierarquia foi analisada segundo os seguintes critérios: tempo de construção e sua de área de superfície. O primeiro avalia a complexidade de cada algoritmo, enquanto o segundo é uma medida da qualidade da divisão espacial obtida. As áreas não possuem unidade específica, sendo expressadas em unidades da cena.

A área de superfície de uma hierarquia é definida como a soma das áreas de superfície de todos os seus volumes envolventes. Dessa forma, procura-se avaliar de forma geral o ajuste obtido para todos os nós da árvore. Contudo, este não pode ser o único parâmetro a ser analisado, pois uma árvore com poucos nós irá, necessariamente, possuir uma menor área de superfície.

Deve-se atentar, também, ao balanceamento da hierarquia que pode não ser o melhor possível. Neste caso, é necessário analisar a altura da árvore obtida, bem como seu total de nós e de folhas. Assim, pode-se ter uma comparação mais adequada para diferentes hierarquias. As heurísticas de construção da hierarquia seguem a nomenclatura a seguir:

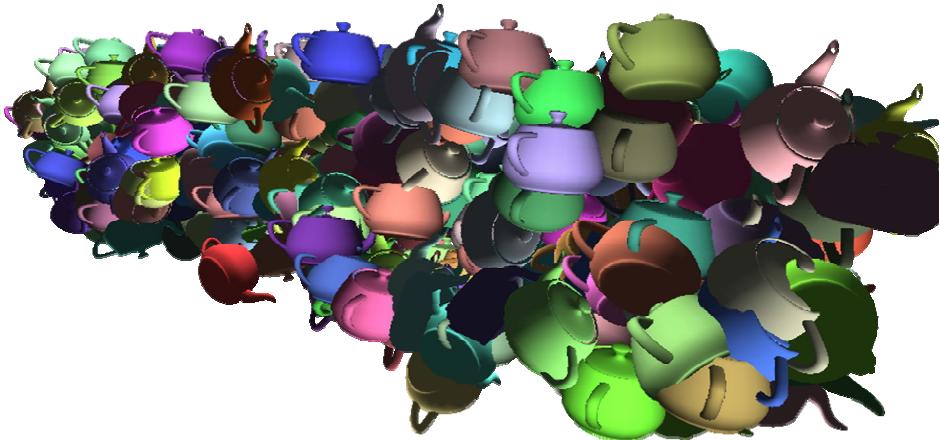
- **Med\_Obj**: Plano de corte posicionado na mediana dos objetos no nó corrente.
- **Med\_Esp**: Utiliza-se a medina espacial (centro da caixa envolvente) como posição do plano de corte.
- **Med\_Cent**: Obtém-se o ponto intermediário entre o centro da primeira e da última geometrias como posição do plano de corte.
- **Avg\_Cent**: Primeiro calcula-se a média geométrica de todos os centros de todas as geometrias. O plano de corte é posicionado neste ponto.
- **Eq\_Comp**: Procura equilibrar a complexidade geométrica em ambos os lados dos nós filhos a serem gerados.
- **SAH**: Posiciona o plano de corte de modo a minimizar uma função de custo que depende das áreas dos filhos a serem gerados e a complexidade das geometrias em cada um.

Segundo a análise anterior dos volumes envolventes, optou-se por avaliar a influência de três deles na construção de hierarquias e renderização das mesmas: **AABB**, **Covar** e **Approx\_Min**. Além disso, optou-se pelo limite inferior de 1000 vértices em cada folha da hierarquia. Com isso, evita-se árvores excessivamente profundas devido à grande quantidade de geometrias pouco complexas em uma cena.

Para os testes efetuados nesta etapa, foram escolhidas quatro cenas diferentes: **Teapots(500,30)**, **Teapots(5000,3)**, **Teapots(5000,20)** e **P-38**. Cada volume envolvente será analisado em cada uma destas cenas. É importante ressaltar que o uso de *Display Lists* possibilita ao hardware gráfico otimizar a visualização das geometrias na cena. Como observado nos testes, a placa gráfica foi capaz de melhorar o desempenho da visualização quando havia poucas geometrias dentro do volume de visão. Neste sentido, uma espécie de *Frustum Culling* em hardware já estava sendo efetuado.

### 6.4.1 Teapots(500,30)

Contendo uma pequena quantidade de geometrias muito grandes, seu objetivo é testar os algoritmos em uma situação simples onde há poucos objetos e muita oclusão. Espera-se medir a eficiência de *Occlusion Culling* para cenas que já possuem boa performance.



**Figura 6.4.1:** Observa-se a baixa complexidade geométrica aliada a muita oclusão na cena.

#### 6.4.1.1 AABB

Os testes com a caixa envolvente definida pelo algoritmo de construção de uma AABB obtiveram os seguintes resultados.

	Hierarquia		Visualização		
	tempo	area	simples	vfc	chc
<b>Med_Obj</b>	1,31	570.884,5	3,134	2,982	1,892
<b>Med_Esp</b>	1,36	548.270,2	3,165	2,984	1,917
<b>Med_Cent</b>	1,35	564.459,2	3,168	2,980	1,890
<b>Avg_Cent</b>	1,31	561.398,3	3,169	3,010	1,888
<b>Eq_Comp</b>	1,30	573.621,8	3,185	2,982	1,892
<b>SAH</b>	216,28	515.468,2	3,133	2,983	1,924

**Tabela 6.4.1.1.1:** Resultados da construção da hierarquia e performance de visualização.

	Med_Obj	Med_Esp	Med_Cent	Avg_Cent	Eq_Comp	SAH
Altura	9	11	12	11	9	11
No. Nós	999	929	999	999	999	935
No. Folhas	500	465	500	500	500	468

**Tabela 6.4.1.1.2:** Características das hierarquias construídas.

Primeiramente, observa-se o tempo de construção muito elevado para a heurística de minimização da função de **SAH**. Além disso, pode-se concluir que as demais heurísticas possuem tempo de construção similar para esta cena.

A coluna **simples** indica o tempo para percorrer o caminho durante a renderização, utilizando-se um algoritmo de visualização padrão. Nota-se que as técnicas de *Frustum Culling* e *Occlusion Culling* são capazes de reduzir o tempo para percorrer o caminho em cerca de 5% e 40%, respectivamente.

Na tabela 6.4.1.1.2, observa-se que as diferentes heurísticas de construção de hierarquia obtiveram árvores com diferentes alturas e total de nós e folhas. Mesmo assim, não houve diferença significativa no desempenho da aplicação para as diferentes hierarquias construídas.

#### 6.4.1.2 OBB por Covariância

Utilizando a caixa orientada dada pela covariância dos vértices, foram obtidos os seguintes resultados.

	Hierarquia		Visualização		
	tempo	area	simples	vfc	chc
<b>Med_Obj</b>	1,42	588.440,0	3,162	2,979	1,919
<b>Med_Esp</b>	1,47	565.394,9	3,131	2,980	1,928
<b>Med_Cent</b>	1,48	577.345,2	3,132	2,978	1,924
<b>Avg_Cent</b>	1,44	579.327,5	3,138	2,978	1,919
<b>Eq_Comp</b>	1,42	589.520,4	3,160	2,977	1,923
<b>SAH</b>	260,31	517.034,2	4,855	4,633	2,867

Tabela 6.4.1.2.1: A heurística de SAH, embora com menor área, obteve o pior desempenho.

	<b>Med_Obj</b>	<b>Med_Esp</b>	<b>Med_Cent</b>	<b>Avg_Cent</b>	<b>Eq_Comp</b>	<b>SAH</b>
Altura	9	12	12	11	9	11
No. Nós	999	955	997	999	999	921
No. Folhas	500	478	499	500	500	461

Tabela 6.4.1.2.2: Características das hierarquias construídas.

Novamente, a heurística de **SAH** mostrou-se lenta para a construção da hierarquia. Observa-se que, mesmo com uma árvore de menor área de superfície, seu desempenho foi o pior dentre as heurísticas testadas. Mais uma vez, o algoritmo de *Frustum Culling* obteve uma melhoria de 5% em relação à renderização simples. O algoritmo de *Occlusion Culling* também manteve uma melhoria em torno de 40%.

Nota-se na tabela 6.4.1.2.2 que a árvore gerada pela heurística de **SAH** possui o menor número de nós e folhas, seguida pela heurística **Med\_Esp**. Observa-se que os

dois piores desempenhos no algoritmo de *Occlusion Culling* foram obtidos justamente com estas duas heurísticas de construção de hierarquias.

#### 6.4.1.3 OBB por Aproximação da Caixa Mínima

Utilizando o algoritmo de **Aprox\_Min** para a caixa orientada obteve os seguintes resultados.

	Hierarquia		Visualização		
	tempo	area	simples	vfc	chc
<b>Med_Obj</b>	113,54	493.661,1	3.138	2,965	1,831
<b>Med_Esp</b>	127,56	471.301,7	3.182	2,985	1,874
<b>Med_Cent</b>	131,51	480.400,8	3.180	2,987	1,880
<b>Avg_Cent</b>	125,92	483.034,3	3.186	2,991	1,869
<b>Eq_Comp</b>	126,10	494.672,8	3.179	2,988	1,870
<b>SAH</b>	-	-	-	-	-

Tabela 6.4.1.3.1: Mais uma vez, resultados similares de desempenho.

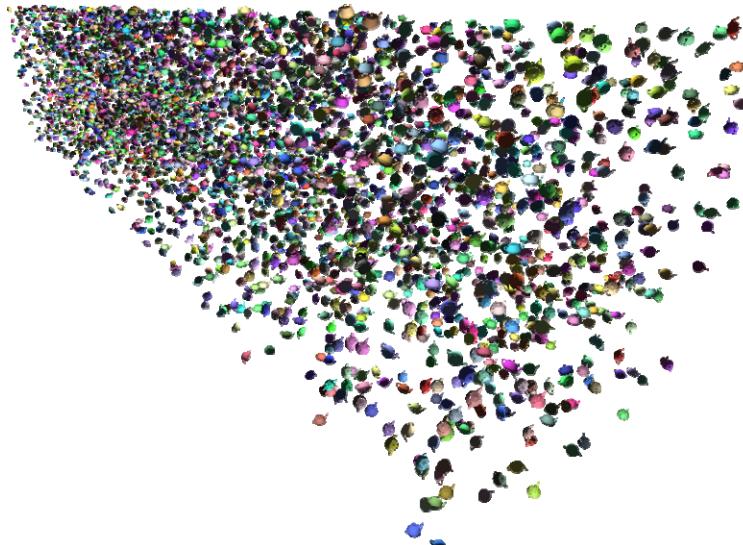
	Med_Obj	Med_Esp	Med_Cent	Avg_Cent	Eq_Comp	SAH
Altura	9	11	12	11	9	-
No. Nós	999	957	999	999	999	-
No. Folhas	500	479	500	500	500	-

Tabela 6.4.1.3.2: Características das hierarquias construídas.

Devido ao elevado tempo de processamento do algoritmo de **Aprox\_Min**, não foi possível obter-se uma medida para a heurística de **SAH**. Os ganhos de *Frustum Culling* e *Occlusion Culling* melhoraram um pouco para 7% e 42%, respectivamente. Mais uma vez, não se pode observar nenhum impacto significativo das diferentes hierarquias na performance da aplicação de visualização.

#### 6.4.2 Teapots(5000,3)

Nesta cena, procura-se testar os algoritmos para cenas complexas porém ainda com pouca oclusão. Será medido o *overhead* de testes de *Occlusion Culling* para uma grande quantidade de objetos visíveis.



**Figura 6.4.2:** Uma grande quantidade de objetos compõe a cena, porém há muito pouca oclusão.

#### 6.4.2.1 AABB

O algoritmo de **AABB** obteve os seguintes resultados para esta cena.

	Hierarquia		Visualização		
	tempo	area	simples	vfc	chc
<b>Med_Obj</b>	20,38	388.791,0	27,211	25,331	24,072
<b>Med_Esp</b>	20,45	372.349,8	27,668	25,787	24,489
<b>Med_Cent</b>	20,14	372.693,0	27,212	25,330	24,069
<b>Avg_Cent</b>	20,20	372.912,5	27,491	25,330	24,063
<b>Eq_Comp</b>	22,64	398.813,9	27,480	25,331	24,081
<b>SAH</b>	-	-	-	-	-

**Tabela 6.4.2.1.1:** A proposta **Med\_Esp** começa a obter os piores resultados.

	Med_Obj	Med_Esp	Med_Cent	Avg_Cent	Eq_Comp	SAH
Altura	13	16	16	16	13	-
No. Nós	9999	9985	9999	9999	9999	-
No. Folhas	5000	4993	5000	5000	5000	-

**Tabela 6.4.2.1.2:** Características das hierarquias construídas.

Como a cena **Teapots(5000,3)** possui um número elevado de objetos, não foi possível obter uma avaliação para a heurística de **SAH**. O ganho de *Frustum Culling* continua em torno de 8%. Observa-se que a pouca oclusão na cena fez com que o algoritmo de *Occlusion Culling* não fosse capaz de manter os mesmos resultados obtidos anteriormente. De qualquer forma, o uso desta técnica de otimização ainda trouxe uma melhoria em torno de 12% em relação à renderização simples. Isto mostra a eficácia do algoritmo, que se prova capaz de obter bons resultados mesmo em situações

desfavoráveis. Finalmente, observa-se que a heurística **Med\_Esp** dá indícios de uma hierarquia ruim para o algoritmo de *Occlusion Culling*.

#### 6.4.2.2 OBB por Covariância

O algoritmo de **OBB** por covariância obteve os seguintes resultados para esta cena.

	Hierarquia		Visualização		
	tempo	area	simples	vfc	chc
<b>Med_Obj</b>	21,84	454.606,7	27,167	25,327	24,368
<b>Med_Esp</b>	21,98	449.312,4	27,173	25,359	24,365
<b>Med_Cent</b>	21,70	454.704,2	27,190	25,330	24,366
<b>Avg_Cent</b>	21,58	448.239,2	27,497	25,639	24,656
<b>Eq_Comp</b>	21,94	457.389,4	27,175	25,330	24,374
<b>SAH</b>	-	-	-	-	-

Tabela 6.4.2.2.1: Nenhuma diferença significativa entre os algoritmos utilizados.

	Med_Obj	Med_Esp	Med_Cent	Avg_Cent	Eq_Comp	SAH
Altura	13	16	16	16	13	-
No. Nós	9999	9985	9999	9999	9999	-
No. Folhas	5000	4993	5000	5000	5000	-

Tabela 6.4.2.2.2: Características das árvores construídas.

Os resultados obtidos com o uso de uma **OBB** não se diferenciam dos demais para esta mesma cena. A melhoria no tempo para percorrer o caminho dos algoritmos de *Frustum Culling* e *Occlusion Culling* continua próximo aos anteriores, correspondendo a 7% e 10% respectivamente. Um observador mais rigoroso pode notar que os valores de desempenho obtidos com a **OBB** são ligeiramente maiores do que os obtidos com **AABB**.

#### 6.4.2.3 OBB por Aproximação da Caixa Mínima

Não foi possível obter muitos resultados para o uso da aproximação da caixa mínima, pois seu elevado tempo de processamento fez com que a construção da hierarquia demorasse quase 1h. Portanto, somente foi possível obter uma medida para a heurística **Med\_Obj**.

	Hierarquia			Visualização	
	tempo	area	simples	vfc	chc
Med_Obj	3.208,92	335.796,3	27.422	25.334	24.021
Med_Esp	-	-	-	-	-
Med_Cent	-	-	-	-	-
Avg_Cent	-	-	-	-	-
Eq_Comp	-	-	-	-	-
SAH	-	-	-	-	-

Tabela 6.4.2.3.1: Elevado tempo de construção da hierarquia inviabilizou os demais resultados.

	Med_Obj	Med_Esp	Med_Cent	Avg_Cent	Eq_Comp	SAH
Altura	13	-	-	-	-	-
No. Nós	9999	-	-	-	-	-
No. Folhas	5000	-	-	-	-	-

Tabela 6.4.2.3.2: A árvore construída não se difere das obtidas anteriormente.

Como o desempenho deste tipo de caixa orientada se mostrou similar ao uso da **AABB**, seu elevado tempo de pré-processamento o descarta como opção viável de uso.

### 6.4.3 Teapots(5000,20)

A diferença desta cena para a anterior é o maior tamanho dos objetos a serem visualizados. Isto possui duas consequências: a primeira é que não se altera o resultado de construção de hierarquias já que estes são baseados apenas nos centros dos objetos. Portanto, foram omitidos estes mesmos valores. A segunda consequência é um aumento no grau de oclusão dos objetos na cena. Esta mudança permite a análise do desempenho de *Occlusion Culling* em seu melhor cenário: uma cena complexa e com alto grau de oclusão.

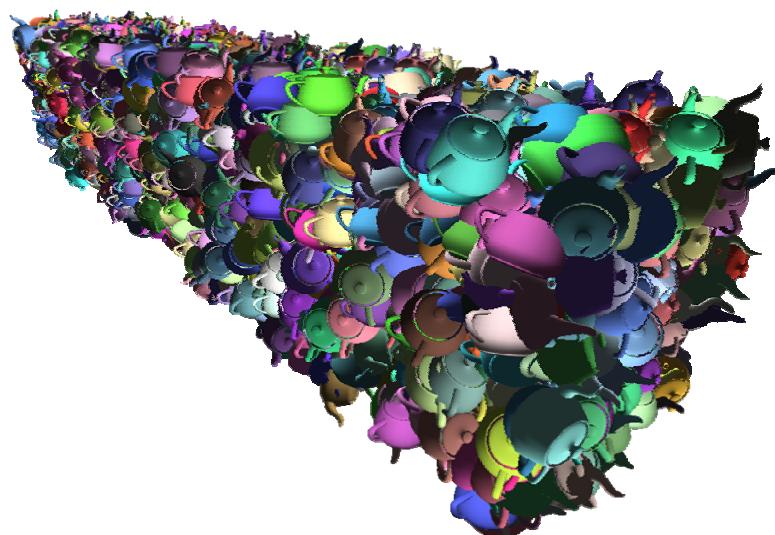


Figura 6.4.2: Muitos objetos grandes causam um elevado nível de oclusão na cena.

### 6.4.3.1 AABB

Nesta cena, a caixa construída como uma **AABB** obteve os seguintes resultados.

	Hierarquia		Visualização		
	tempo	area	simples	vfc	chc
<b>Med_Obj</b>	-	-	27,962	26,176	5,371
<b>Med_Esp</b>	-	-	27,896	26,204	5,550
<b>Med_Cent</b>	-	-	27,951	26,172	5,354
<b>Avg_Cent</b>	-	-	27,973	26,185	5,360
<b>Eq_Comp</b>	-	-	27,951	26,178	5,382
<b>SAH</b>	-	-	-	-	-

Tabela 6.4.3.1.1: Nota-se a melhoria excelente no desempenho por parte do *Occlusion Culling*.

A elevada complexidade geométrica na cena, aliada ao seu alto nível de oclusão favoreceu claramente o uso do algoritmo de *Occlusion Culling*. Esta técnica de otimização foi capaz de reduzir o tempo para percorrer o caminho em cerca 81%. Isto se traduz em um ganho de quase 6x na taxa média de quadros por segundo da aplicação de visualização.

### 6.4.3.2 OBB por Covariância

A caixa orientada por covariância produziu os resultados abaixo.

	Hierarquia		Visualização		
	tempo	area	simples	vfc	chc
<b>Med_Obj</b>	-	-	28,191	26,582	5,820
<b>Med_Esp</b>	-	-	27,941	26,172	5,847
<b>Med_Cent</b>	-	-	27,914	26,151	5,767
<b>Avg_Cent</b>	-	-	28,206	26,464	5,774
<b>Eq_Comp</b>	-	-	27,892	26,155	6,167
<b>SAH</b>	-	-	-	-	-

Tabela 6.4.3.2.1: Resultados similares ao uso da **AABB**, porém ligeiramente piores.

Os resultados com o uso deste algoritmo de caixa orientada foram similares ao uso de uma **AABB**. Mais uma vez, a técnica de *Occlusion Culling* comprova uma melhoria em torno de 80% no desempenho da aplicação. Uma análise mais rigorosa pode notar, mais uma vez, que os valores de tempo foram ligeiramente maiores do que os obtidos com a **AABB**.

### 6.4.3.3 OBB por Aproximação da Caixa Mínima

Os resultados da aproximação da caixa orientada mínima encontram-se na tabela a seguir. Mais uma vez, o elevado tempo de construção da hierarquia utilizando este algoritmo inviabilizou a obtenção dos demais resultados.

	Hierarquia		Visualização		
	tempo	area	simples	vfc	chc
<b>Med_Obj</b>	-	-	29,341	26,897	5,818
<b>Med_Esp</b>	-	-	-	-	-
<b>Med_Cent</b>	-	-	-	-	-
<b>Avg_Cent</b>	-	-	-	-	-
<b>Eq_Comp</b>	-	-	-	-	-
<b>SAH</b>	-	-	-	-	-

Tabela 6.4.3.3.1: Nenhuma melhoria em relação ao uso da caixa orientada por covariância.

Não se pode observar nenhuma melhoria em relação ao uso dos demais algoritmos de volumes envolventes. Mais uma vez, não se justifica o uso do algoritmo **Aprox\_Min** para construção das caixas envolventes.

### 6.4.4 P-38

Nesta cena, espera-se obter uma avaliação do desempenho dos algoritmos em um caso de uso real. Esta cena possui uma grande quantidade de geometrias, na sua maioria com pouca complexidade e distribuídas de forma irregular no espaço. Ao contrário das cenas anteriores, a construção de uma boa hierarquia será um fator determinante para o desempenho dos algoritmos de renderização.

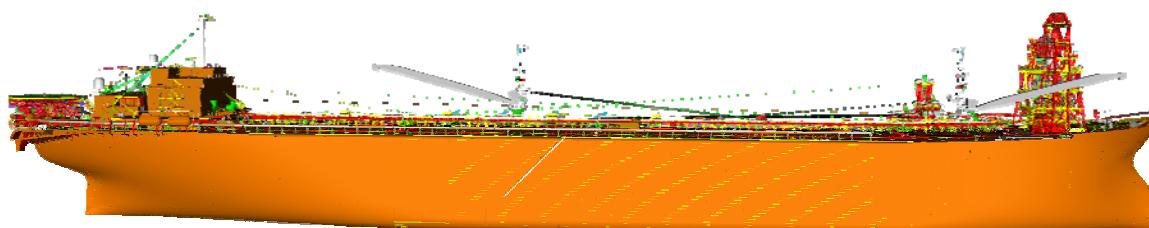


Figura 6.4.4: O modelo da plataforma de petróleo utilizado nos testes a seguir.

#### 6.4.4.1 AABB

O algoritmo de **ABB** obteve os seguintes resultados para esta cena.

	Hierarquia		Visualização		
	tempo	area	simples	vfc	chc
<b>Med_Obj</b>	19,37	10.120.705,5	78,717	74,827	34,974
<b>Med_Esp</b>	27,33	3.307.890,7	76,092	74,604	45,857
<b>Med_Cent</b>	35,08	9.222.618,9	77,705	92,528	44,766
<b>Avg_Cent</b>	20,91	9.682.044,9	76,119	74,709	33,964
<b>Eq_Comp</b>	16,98	10.818.353,7	78,814	75,015	33,384
<b>SAH</b>	-	-	-	-	-

Tabela 6.4.4.1.1: Mais uma vez, o algoritmo de *Occlusion Culling* prova sua eficácia.

	Med_Obj	Med_Esp	Med_Cent	Avg_Cent	Eq_Comp	SAH
Altura	14	21	17	17	13	-
No. Nós	4299	1345	3789	4365	5779	-
No. Folhas	2150	673	1895	2183	2890	-

Tabela 6.4.4.1.2: Características das árvores construídas.

Nesta cena, pode-se observar que a heurística **Med\_Esp** obteve o pior resultado, seguida pela hierarquia gerada segundo **Med\_Cent**. O tempo para percorrer o caminho de ambas é cerca de 24% pior do que as demais. Observa-se que estas duas heurísticas obtiveram as árvores mais desbalanceadas, com altura elevada e um número de nós e folhas menor do que as demais.

Os algoritmos de *Frustum Culling* e *Occlusion Culling* obtiveram uma melhoria no desempenho da aplicação em cerca de 5% e 57%, respectivamente. Pode-se concluir, portanto, que a técnica de otimização de *Coherent Hierarchical Culling* foi capaz de dobrar a taxa média de quadros por segundo da aplicação de visualização.

#### 6.4.4.2 OBB por Covariância

O uso da **OBB** por covariância obteve os seguintes resultados.

	Hierarquia		Visualização		
	tempo	area	simples	vfc	chc
<b>Med_Obj</b>	18,80	12.967.258,2	78,983	75,460	36,511
<b>Med_Esp</b>	22,50	6.790.755,5	76,653	74,459	42,337
<b>Med_Cent</b>	26,68	12.221.845,7	78,045	74,705	37,240
<b>Avg_Cent</b>	46,02	12.753.301,4	78,014	74,488	36,546
<b>Eq_Comp</b>	18,37	14.620.466,7	80,663	75,556	34,913
<b>SAH</b>	-	-	-	-	-

Tabela 6.4.4.2.1: Ganhos similares aos obtidos anteriormente.

	<b>Med_Obj</b>	<b>Med_Esp</b>	<b>Med_Cent</b>	<b>Avg_Cent</b>	<b>Eq_Comp</b>	<b>SAH</b>
Altura	15	19	16	15	12	-
No. Nós	4531	2149	3965	4523	6557	-
No. Folhas	2266	1075	1983	2262	3279	-

Tabela 6.4.4.2.2: Características das árvores construídas.

Mais uma vez, pode-se observar que a heurística **Med\_Esp** obteve o pior resultado. O tempo para percorrer o caminho de ambas é cerca de 14% pior do que as demais. Observa-se que esta heurística produziu novamente uma árvore desbalanceada, com a maior altura e o menor número de nós e folhas.

As técnicas de *Frustum Culling* e *Occlusion Culling* obtiveram novamente uma melhoria de cerca de 4,6% e 53%, respectivamente. Conforme uma tendência já observada nas cenas dos *Teapots*, os valores da **OBB** por covariância são um pouco maiores do que os obtidos com o uso de uma **AABB**.

#### 6.4.4.3 OBB por Aproximação da Caixa Mínima

A **OBB** obtida como aproximação da caixa mínima obteve os resultados a seguir.

	<b>Hierarquia</b>		<b>Visualização</b>		
	tempo	area	simples	vfc	chc
<b>Med_Obj</b>	1.032,08	9.354.155,7	73,735	69,834	31,130
<b>Med_Esp</b>	1.002,49	4.122.728,5	73,024	70,786	43,518
<b>Med_Cent</b>	-	-	-	-	-
<b>Avg_Cent</b>	-	-	-	-	-
<b>Eq_Comp</b>	-	-	-	-	-
<b>SAH</b>	-	-	-	-	-

Tabela 6.4.4.3.1: Mais uma vez, o algoritmo de *Occlusion Culling* prova sua eficácia.

	<b>Med_Obj</b>	<b>Med_Esp</b>	<b>Med_Cent</b>	<b>Avg_Cent</b>	<b>Eq_Comp</b>	<b>SAH</b>
Altura	14	19	-	-	-	-
No. Nós	4457	1791	-	-	-	-
No. Folhas	2229	896	-	-	-	-

Tabela 6.4.4.3.2: Características das árvores construídas.

Observa-se o tempo elevado de construção da hierarquia, aliado a um desempenho similar aos obtidos anteriormente. Pode-se considerar que o algoritmo de *Occlusion Culling* obteve uma melhoria de cerca de 10% em relação às demais caixas envolventes utilizadas. Mais uma vez, a heurística **Med\_Esp** obteve o pior resultado de desempenho aliado a uma hierarquia desbalanceada, de altura elevada e pouco número de nós.

Neste caso de teste, é importante destacar que o tempo de construção das hierarquias não foi o motivo pelo qual não foram obtidos os demais resultados. Na verdade, foram encontrados inúmeros problemas de estabilidade no algoritmo **Aprox\_Min** utilizado. A implementação fornecida pelos autores da proposta não foi capaz de tratar certas entradas singulares, interrompendo a execução do programa de testes.

## 6.5 Otimizações de *Occlusion Culling*

Para testar as otimizações de *Coherent Hierarchical Culling* implementadas, optou-se por utilizar a cena de teste da **P-38**, com a construção da hierarquia que obteve o melhor resultado: **Eq\_Comp**. Utilizou-se a **AABB** como caixa envolvente, por seu resultado ligeiramente melhor do que a **OBB** por covariância.

### 6.5.1 Evitar Testes Consecutivos

Para avaliar esta otimização, experimentou-se evitar diferentes quantidades de testes consecutivos de visibilidade para um mesmo nó. Os resultados são os seguintes:

	2 quadros	5 quadros	10 quadros	15 quadros
Test_Consec	33,083	33,036	33,208	33,335

Tabela 6.6.1: Resultados de desempenho.

Comparando-se estes resultados com a tabela 6.4.4.1.1, observa-se que a alternativa de evitar *queries* para os próximos 5 quadros obteve um ganho de cerca de 1%. Claramente, esta proposta não foi capaz de obter uma melhoria significativa no desempenho da aplicação.

### 6.5.2 Probabilístico

Nesta outra proposta, utilizou-se um esquema probabilístico para tentar “adivinar” se um dado nó previamente visível continua visível no frame atual. Os resultados são os seguintes:

	10%	20%	30%	50%	60%	70%	80%	90%
Probab	33,518	33,401	33,376	33,144	33,067	33,040	32,993	33,321

Tabela 6.6.2: Resultados de desempenho.

Observa-se que é necessário um valor alto de probabilidade para que se consiga algum ganho em relação ao algoritmo original. Mesmo assim, este ganho não passa de 1%.

### 6.5.3 Proximidade do Observador

Esta última proposta procura evitar o envio de queries para as folhas mais próximas do observador. Os testes utilizaram diferentes percentagens do total de folhas visíveis no quadro anterior como o número de testes consecutivos que serão evitados no quadro atual.

	10%	20%	30%	50%
Prox_Observ	33,959	33,726	33,741	33,837

Tabela 6.6.3: Resultados de desempenho.

Mais uma vez, não foi possível obter-se uma melhoria significativa no desempenho da aplicação de visualização.

## 6.6 Análise dos Resultados

Em todas as cenas de teste que utilizavam *Teapots*, não foi possível observar nenhuma diferença significativa no uso das heurísticas de construção de hierarquias. De forma similar, o uso de diferentes caixas envolventes não trouxe nenhum benefício significativo no desempenho da aplicação de visualização. Os valores de tempo para renderização simples, similares em todas as cenas, comprovam a hipótese de que o gargalo destas aplicações se encontra na GPU.

Mesmo na cena **Teapots(500,30)**, de pouca complexidade, o algoritmo de *Coherent Hierarchical Culling* foi capaz de obter um ganho de cerca de 40%. Isto significa que a performance da aplicação quase dobrou em relação à renderização simples. Na cena **Teapots(5000,3)**, onde há muitos objetos mas pouca oclusão, a técnica de *Occlusion Culling* implementada obteve uma melhoria de cerca de 10%. O uso da cena **Teapots(5000,20)**, com muita oclusão, favoreceu claramente a utilização desta técnica como forma de otimizar o desempenho de visualização. Seu ganho de cerca de 80% no tempo para percorrer o caminho traduz-se em uma melhoria de 5 a 6 vezes na taxa média de quadros por segundo da aplicação.

O caso de uso real, ilustrado pela cena **P-38**, ajudou a obter uma análise mais refinada dos algoritmos utilizados. Nos testes efetuados, foi possível observar que as heurísticas de construção de hierarquia **Med\_Esp** e **Med\_Cent** obtiveram os piores

resultados. Estas heurísticas proporcionaram um desempenho cerca de 20% abaixo das demais. Provavelmente, este fenômeno foi causado pelas suas árvores desbalanceadas, com altura elevada e um número pequeno de nós e folhas. Nos testes de **AABB** e **OBB\_Covar**, pode-se observar que a heurística de **Eq\_Comp** obteve resultados ligeiramente melhores do que todos os demais.

Considerando-se todas estas observações, pode-se identificar o seguinte padrão: as heurísticas de construção de hierarquia baseadas em uma divisão do espaço (**Med\_Esp** e **Med\_Cent**) obtiveram um resultado pior do que as heurísticas baseadas em uma divisão dos objetos. Em particular, uma divisão que procura equilibrar a complexidade geométrica dos nós foi a que obteve os melhores resultados de desempenho.

Pode-se conjecturar a seguinte proposição: ao contrário da hipótese inicial, a área de superfície das caixas envolventes não foi um fator determinante para um bom desempenho da aplicação. Os resultados obtidos com a heurística de **SAH** foram piores do que os demais. Observou-se também que as hierarquias não eram tão balanceadas.

Chega-se à seguinte conclusão: obter volumes envolventes que sejam bem-ajustados e os mais compactos possíveis não traz benefícios significativos para o uso de um algoritmo baseado no *Coherent Hierarchical Culling*. Neste caso, é mais favorável o uso de hierarquias balanceadas que procuram, de alguma forma, equilibrar a complexidade das geometrias em cada nó. Acreditava-se que o custo maior para efetuar-se uma *occlusion query* estava na rasterização das caixas envolventes utilizadas nos testes. Na verdade, os resultados obtidos indicam que o custo maior dos testes visibilidade se encontra na complexidade geométrica dos objetos sendo testados. Isto condiz com os testes de *occlusion queries* que indicaram uma relação diretamente proporcional com a complexidade das geometrias sendo testadas.

Finalmente, pode-se notar que todas as otimizações do algoritmo de *Occlusion Culling* implementadas não foram capazes de obter uma melhoria significativa no desempenho da aplicação. Isto indica a eficiência do algoritmo original, mas não pode levar à conclusão de que não há espaço para melhorias.

## 7 Conclusão

Infelizmente, o prazo de entrega do projeto impossibilitou alguns testes que poderiam ter sido efetuados para os algoritmos mais demorados. Além disso, caso houvesse mais tempo poderia ter sido investigada alguma técnica adicional de otimização. Particularmente, pode-se utilizar os resultados desta biblioteca para construção de um *PVS* e pré-processamento. Assim, evitaria-se o *overhead* de testes de visibilidade feitos em tempo de execução.

### 7.1 Contribuições do Projeto

Este projeto traz consigo duas contribuições principais: uma teórica e uma prática. A primeira consiste em um estudo comprehensivo das mais variadas técnicas de otimização para modelos massivos presentes no estado da arte. Com este estudo, fundamenta-se uma base de conhecimento para novas propostas de melhoria dos algoritmos já existentes ou mesmo para construção de novas técnicas que visam resolver o problema sendo tratado.

A segunda contribuição do projeto consiste na implementação de uma biblioteca que disponibiliza quatro funcionalidades essenciais para a visualização de modelos massivos. O desenvolvimento de módulos com alta coesão e baixo acoplamento possibilita a fácil integração desta biblioteca a diversas aplicações de visualização. As funcionalidades disponíveis podem ser resumidas a seguir:

- Algoritmos de construção de caixas envolventes.
- Algoritmos de construção de uma hierarquia de agrupamento espacial.
- Uma implementação do algoritmo de Frustum Culling, incluindo duas otimizações.
- Uma implementação do algoritmo de *Occlusion Culling* conhecido por *Coherent Hierarchical Culling*.

É importante destacar, também, que o estudo efetuado sobre as *occlusion queries* trouxe informações importantes a respeito do *overhead* de uso das mesmas, bem como suas vantagens e limitações. Finalmente, os resultados dos testes efetuados comprovam as seguintes proposições:

- O uso de uma hierarquia torna os algoritmos de *Frustum Culling* e *Occlusion Culling* escaláveis, sendo essencial para sua utilização com modelos massivos.

- Somente o algoritmo de *Frustum Culling* não é capaz de garantir um bom desempenho da aplicação de visualização.
- O algoritmo de *Coherent Hierarchical Culling* é de fato capaz de obter grandes melhorias na performance da visualização. No pior caso, quando há pouca oclusão, o algoritmo ainda é capaz de obter melhorias sem se tornar um *overhead* para a visualização. Em seu melhor caso, obtém-se ganhos de pelo menos 6x a performance original da aplicação.
- O agrupamento espacial utilizado no algoritmo de *Occlusion Culling* é capaz de influenciar positiva ou negativamente seu desempenho. Pelos resultados obtidos, ao invés de otimizar o ajuste dos volumes envolventes é na verdade mais importante procurar garantir uma hierarquia que equilibre a complexidade geométrica dos objetos ao longo da árvore.

## 7.2 Trabalhos Futuros

O algoritmo de caixa mínima aproximada provou ser capaz de construir caixas orientadas melhores do que quase todas as demais abordagens. Uma linha de pesquisa futura consiste no aprimoramento deste algoritmo, melhorando não somente seu tempo de execução mas também tornando-o robusto e capaz de lidar com quaisquer tipos de entrada.

Na tentativa de melhorar ainda mais o desempenho do algoritmo de *Coherent Hierarchical Culling*, não foram exploradas propostas para eliminar completamente *CPU Stalls* através da renderização especulativa de *queries* remanescentes. Além disso, podem ser desenvolvidas outras estratégias que visam diminuir o número de *occlusion queries* efetuadas a cada quadro. Uma análise mais detalhada das técnicas propostas neste trabalho pode indicar que uma delas pode trazer uma melhoria do desempenho em situações específicas. Com isso, um algoritmo inteligente poderia ser capaz de tirar o melhor proveito de diferentes propostas dependendo do estado corrente da visualização.

Finalmente, os resultados obtidos indicam para um estudo mais comprehensivo de diferentes hierarquias para utilização durante o *Occlusion Culling*. Ainda é necessário identificar exatamente quais são as características que definem uma “boa” hierarquia para o uso de *occlusion queries*. Além disso, pode-se considerar a modelagem da construção por uma estratégia gulosa, como foi feito na proposta de SAH. Contudo, torna-se necessário derivar uma outra função de custo mais apropriada para o uso com *Occlusion Culling*.

## 8 Referências Bibliográficas

- [1] Möller, Thomas & Haines, Eric. "Real-Time Rendering", A K Peters, 2002.
- [2] Luebke et al. "Level of Detail for 3D Graphics", Morgan Kaufmann, 2003.
- [3] Yoon. S., Salomon, B., Gayle, R. and Manocha, D. "Quick-VDR: Out-of-Core View-Dependent Rendering of Gigantic Models". IEEE Transactions on Visualization and Computer Graphics, July/August 2005, p369-382.
- [4] Cohen-Or D., Chrysanthou Y., Silva C. T. "A survey of visibility for walkthrough applications". Proc. of EUROGRAPHICS'00, course notes, 2000.
- [5] Airey, John. "Increasing Update Rates in the Building Walkthrough System with Automatic Model-Space Subdivision and Potentially Visible Set Calculations". Ph.D. thesis, UNC-CH CS Department TR #90-027 (July 1990).
- [6] Teller, Seth. "Visibility Computation in Densely Occluded Polyhedral Environments". Ph.D. thesis, UC Berkeley CS Department, TR #92/708 (1992).
- [7] David P. Luebke Chris Georges. "Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets". Proceedings of the 1995 symposium on Interactive 3D graphics.
- [8] Greene et al. "Hierarchical Z-Buffer Visibility", Proceedings of SIGGRAPH '93 (Anaheim, California 1993).
- [9] Hudson et al. "Accelerated occlusion culling using shadow frustums". In Proc. 13th Annu. ACM Sympos. Comput. Geom., pages 1–10, 1997.
- [10] I. Wald, P. Slusallek, and C. Benthin. "Interactive distributed ray tracing of highly complex models". In Rendering Techniques 2001 - Proceedings of the 12<sup>th</sup> EUROGRAPHICS Workshop on Rendering, pages 274–285, 2001.
- [11] Zhang et al. "Visibility culling using hierarchical occlusion maps". In Turner Whitted, editor, SIGGRAPH 97 Conference Proceedings, Annual Conference Series, pages 77–88. ACM SIGGRAPH, Addison Wesley, August 1997.
- [12] Bittner et al. "Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful". Institute of Computer Graphics and Algorithms, Vienna University of Technology, EUROGRAPHICS 2004.

- [13] 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.
- [14] Krüger, J. & Westermann, R. "Acceleration Techniques for GPU-based Volume Rendering". Computer Graphics and Visualization Group, Technical University Munich, 2003 IEEE Visualization.
- [15] Assarsson U., Moller T.. "Optimized view frustum culling algorithms for bounding boxes". Journal of Graphics Tools, 5(1):9--22, 2000.
- [16] Hofmam, M. "Tratamento Eficiente de Visibilidade Através de Árvores de Volumes Envolventes". Master thesis, Pontifícia Universidade Católica do Rio de Janeiro, 2002.
- [17] de Toledo, R. "Interactive Visualization of Massive Data Using Graphics Cards". Ph.D. thesis, Lab. LORIA/INRIA. Nancy, França, 2006.
- [18] Jeschke S., Wimmer M., Schumann H., Purgathofer W. "Automatic Impostor Placement for Guaranteed Frame Rates and Low Memory Requirements". ACM SIGGRAPH 2005 Proceedings, pages 103-110. April 2005.
- [19] Gottschalk S., Lin M., Manocha D. "OBB-Tree: A Hierarchical Structure for Rapid Interference Detection", Computer Graphics (SIGGRAPH 96 Proceedings), pp. 171–180, August 1996.
- [20] Duda R. O., Hart P. E., "Pattern Classification and Scene Analysis". John Wiley and Sons, 1973.
- [21] O'Rourke J. "Finding Minimal Enclosing Boxes", International J. Comput. Inform. Sci., 14:183-199, 1985.
- [22] Eberly, D. "3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics", 2nd edition. Morgan Kaufmann Publishers Inc, 2006.
- [23] Barber C., Dobkin D., Huhdanpaa H., "The Quickhull Algorithm for Convex Hull", Geometry Center Technical Report GCG53, Univ. of Minnesota, MN, 1993.
- [24] Barequet, G., Har-Peled, S. "Efficiently approximating the minimum-volume bounding box of a point set in three dimensions". Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms (SODA) (1999), pp. 82--91.

- [25] V. Havran. "Heuristic Ray Shooting Algorithms". Czech Technical University, Ph.D. thesis, 2001.
- [26] Gribb, G., Hartmann K., "Fast Extraction of Viewing Frustum Planes from the World-View-Projection Matrix".
- [27] Gamma, E., Helm, R., Johnson, R., Vlissides, J. "Design Patterns: Elements of Reusable Object-Oriented Software". Addison Wesley, 1995
- [28] Osfield R. "OpenSceneGraph". <http://www.openscenegraph.org>, 2003.