PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

Eduardo Telles Carlos

Frustum Culling Híbrido Utilizando CPU e GPU

Dissertação de Mestrado

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

Orientador : Prof. Alberto Barbosa Raposo
Co–Orientador: Prof. Marcelo Gattass

Rio de Janeiro
Abril de 2009

Pontifícia Universidade Católica
do Rio de Janeiro

Eduardo Telles Carlos

Frustum Culling Híbrido Utilizando CPU e GPU

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

Prof. Alberto Barbosa Raposo
Orientador
Departamento de Informática — PUC–Rio

Prof. Marcelo Gattass
Co–Orientador
Departamento de Informática — PUC–Rio

Prof. Waldemar Celes Filho
Departamento de Informática — PUC-Rio

D.Sc. Luciano Soares
Departamento de Informática — PUC-Rio

Ph.D. Rodrigo de Toledo
Cenpes— Petrobras

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

Rio de Janeiro, 02 de Abril de 2009

Todos os direitos reservados. É proibida a reprodução total ou parcial do trabalho sem autorização da universidade, do autor e do orientador.

Eduardo Telles Carlos

Graduou-se em Engenharia da computação na PUC-Rio em 2006. Desde 2002 trabalha no Grupo de Tecnologia em Computação Gráfica da universidade (TecGraf) desenvolvendo sistemas de realidade virtual e visualização científica.

Ficha Catalográfica

Carlos, Eduardo Telles

Frustum Culling Híbrido Utilizando CPU e GPU / Eduardo Telles Carlos; orientador: Alberto Barbosa Raposo; co–orientador: Marcelo Gattass. — 2009.
99 f: il. (color.); 30 cm

Dissertação (Mestrado em Informática) - Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2009.
Inclui bibliografia

1. Informática – Teses. 2. Algoritmos de visualização. 3. Descarte de volumes envolventes. 4. GPGPU. 5. Primitivas GPU. I. Raposo, Alberto Barbosa. II. Gattass, Marcelo. III. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. IV. Título.

CDD: 004

À minha mãe Nilce Telles Carlos, ao meu pai William Braga Carlos e à minha irmã Érica Telles Carlos. Sem o carinho, amor e dedicação deles, este trabalho não seria possível.

Agradecimentos

Ao meu orientador Alberto Raposo, pela dedicação, confiança e conselhos dados ao longo de toda a execução deste trabalho.

Ao meu co-orientador Marcelo Gattass pelas sugestões e confiança no meu trabalho.

Aos professores Marcelo Gattass e Waldemar Celes Filho pelas excelentes aulas na graduação e no mestrado.

A meus pais e a minha irmã por tudo que sou hoje.

A minha avó, tios, tias, primos, primas e a minha madrinha pela ajuda, conselhos e amor.

A Pontifícia Universidade Católica do Rio de Janeiro pela excelente formação a mim dada.

Ao Tecgraf pela oportunidade de aprendizado e auxílios concedidos.

A todos os integrantes das equipes do SiVIEP, Environ e demais dentro do Tecgraf pela compreensão, aprendizado e momentos de descontração.

Aos grandes amigos André Luis Pinto Ferreira e Rafael Brito de Farias pelas amizades verdadeiras e apoio nos momentos difíceis.

A Rodrigo Toledo por toda a ajuda dada antes mesmo da definição da área de pesquisa a ser seguida e ajuda no framework das GPU primitives.

A Paulo Ivson Netto pela ajuda nos trabalhos ao longo da graduação, do mestrado e no desenvolvimento deste trabalho.

A equipe de suporte do Tecgraf por terem me ajudado com os problemas de hardware.

A paciência dos integrantes do futebol de domingo pela queda no meu rendimento.

Resumo

Carlos, Eduardo Telles; Raposo, Alberto Barbosa; Gattass, Marcelo. **Frustum Culling Híbrido Utilizando CPU e GPU**. Rio de Janeiro, 2009. 99p. Dissertação de Mestrado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Um dos problemas mais antigos da computação gráfica tem sido a determinação de visibilidade. Vários algoritmos têm sido desenvolvidos para viabilizar modelos cada vez maiores e detalhados. Dentre estes algoritmos, destaca-se o *frustum culling*, cujo papel é remover objetos que não sejam visíveis ao observador. Esse algoritmo, muito comum em várias aplicações, vem sofrendo melhorias ao longo dos anos, a fim de acelerar ainda mais a sua execução. Apesar de ser tratado como um problema bem resolvido na computação gráfica, alguns pontos ainda podem ser aperfeiçoados, e novas formas de descarte desenvolvidas. No que se refere aos modelos massivos, necessita-se de algoritmos de alta performance, pois a quantidade de cálculos aumenta significativamente. Este trabalho objetiva avaliar o algoritmo de *frustum culling* e suas otimizações, com o propósito de obter o melhor algoritmo possível implementado em CPU, além de analisar a influência de cada uma de suas partes em modelos massivos. Com base nessa análise, novas técnicas de *frustum culling* serão desenvolvidas, utilizando o poder computacional da GPU (Graphics Processing Unit), e comparadas com o resultado obtido apenas pela CPU. Como resultado, será proposta uma forma de *frustum culling* híbrido, que tentará aproveitar o melhor da CPU e da GPU.

Palavras–chave

Algoritmos de visualização. Descarte de volumes envolventes. GPGPU. Primitivas GPU.

Abstract

Carlos, Eduardo Telles; Raposo, Alberto Barbosa; Gattass, Marcelo. **Hybrid Frustum Culling Using CPU and GPU**. Rio de Janeiro, 2009. 99p. MSc Dissertation — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

The definition of visibility is a classical problem in Computer Graphics. Several algorithms have been developed to enable the visualization of huge and complex models. Among these algorithms, the frustum culling, which plays an important role in this area, is used to remove invisible objects by the observer. Besides being very usual in applications, this algorithm has been improved in order to accelerate its execution. Although being treated as a well-solved problem in Computer Graphics, some points can be enhanced yet, and new forms of culling may be disclosed as well. In massive models, for example, algorithms of high performance are required, since the calculus arises considerably. This work analyses the frustum culling algorithm and its optimizations, aiming to obtain the state-of-the-art algorithm implemented in CPU, as well as explains the influence of each of its steps in massive models. Based on this analysis, new GPU (Graphics Processing Unit) based frustum culling techniques will be developed and compared with the ones using only CPU. As a result, a hybrid frustum culling will be proposed, in order to achieve the best of CPU and GPU processing.

Keywords

Visualization algorithms. Frustum culling. GPGPU. GPU Primitives.

Sumário

1 Introdução **13**
1.1 Motivação 13
1.2 Objetivo da dissertação 15
1.3 Trabalhos relacionados 15
1.4 Organização da dissertação 16

2 Algoritmos de Visibilidade **17**
2.1 Determinação de visibilidade 17
2.2 Classificação dos algoritmos de visibilidade 20
2.3 Pipeline dos algoritmos de visualização 22

3 Frustum culling **26**
3.1 Implementação clássica 27
3.2 Estado da arte 28
3.3 Memória utilizada em CPU 42
3.4 Ambiente de benchmark 43

4 Implementação em CPU **48**
4.1 Renderização 48
4.2 Radar X Planos 48
4.3 Hierarquia 51
4.4 Percurso 53
4.5 Otimizações 56
4.6 SIMD 58
4.7 Multiprocessamento 60
4.8 Pipeline final em CPU 64

5 Frustum culling em GPU **65**
5.1 Timeline gpu 65
5.2 Gpu primitives 67
5.3 Algoritmos de frustum culling em GPU 71
5.4 Memória utilizada em GPU 74
5.5 Percurso sem pilha em GPU 75
5.6 Implementação 76

6 Frustum culling híbrido **79**
6.1 Heurísticas 79
6.2 Implementação 85

7 Conclusão e Trabalhos futuros **89**
7.1 Conclusão 89
7.2 Trabalhos futuros 90

Referências Bibliográficas **92**

Lista de figuras

1.1 Modelos Massivos. 13

2.1 Problema com o algoritmo do pintor. 18
2.2 Técnicas de culling. 19
2.3 Pipeline do OpenGL. 22
2.4 Frustum culling em CPU. 23
2.5 Frustum culling em GPU. 24

3.1 Tipos de Projeção. 26
3.2 Pseudo-código de descarte de pontos. 28
3.3 Volumes envolventes, imagem divulgada por Ericson *et al.* [23] 29
3.4 Teste de descarte de AABB. 30
3.5 Classificação de um ponto P contra o frustum. 31
3.6 Teste de pontos contra radar. 31
3.7 Teste de AABB contra radar. 32
3.8 Hierarquia de volumes envolventes. 33
3.9 Classificação dos vértices n e p. 35
3.10 Volume envolvente do frustum. 36
3.11 Teste de descarte com apenas quatro planos. 37
3.12 Escape index. 38
3.13 SISD X SIMD. 40
3.14 Dados utilizados no nó da hierarquia. 43
3.15 Aplicação desenvolvida para os testes. 44
3.16 Modelos com informações paramétricas. 45
3.17 Caminhos de câmera pelas plataformas. 46
3.18 Caminho de câmera pelo Boeing (73 segundos). 47

4.1 Caminhos de câmera sem hierarquia nas plataformas. 50
4.2 Caminhos de câmera sem hierarquia no Boeing. 51
4.3 Caminhos de câmera com hierarquias onde média foi melhor. 53
4.4 Caminho de câmera com hierarquias onde mediana foi melhor. 54
4.5 Caminho de câmera sem pilha nas plataformas. 55
4.6 Caminho de câmera sem pilha no Boeing. 55
4.7 Otimizações sugeridas por Assarsoon nas plataformas. 57
4.8 Otimizações sugeridas por Assarsoon no Boeing. 57
4.9 Caminhos de câmera pelas plataformas com otimizações. 58
4.10 Caminhos de câmera pelo Boeing com otimizações. 59
4.11 Resultados obtidos por Assarsson em duas cenas. 61
4.12 Percurso com *multithread*. 61
4.13 Multiprocessamento sem troca de tarefas entre threads. 62
4.14 Multiprocessamento sem troca de tarefas entre threads no Boeing. 62
4.15 Multiprocessamento com troca de tarefas entre threads nas plataformas. 63
4.16 Multiprocessamento com troca de tarefas entre threads no Boeing. 64
4.17 Pipeline final em CPU. 64

5.1 Evolução das placas gráficas até a quarta geração. 66
5.2 Evolução das placas gráficas a partir da quarta geração. 66
5.3 Pipeline das placas gráficas modernas. 67
5.4 Variáveis do vertex shader das gpu primitives (extraída de [70]). 68
5.5 Variáveis do pixel shader das gpu primitives (extraída de [70]). 68
5.6 Cilindro Gpu primitives. 69
5.7 Cone GPU primitives. 70
5.8 Torus slice GPU primitives. 70
5.9 Frustum culling junto com GPU primitives. 71
5.10 Frustum culling em shader separado. 72
5.11 Frustum culling em modelos genéricos. 74
5.12 Frustum culling em modelos genéricos. 74
5.13 Memória utilizada no percurso em GPU. 76
5.14 Frustum culling nas GPU primitives. 77
5.15 Frustum culling em GPU para modelos genéricos. 77

6.1 Melhores caminhos de câmera em CPU e GPU nas plataformas. 80
6.2 Melhores caminhos de câmera em CPU e GPU no Boeing. 81
6.3 Esquema do algoritmo de frustum culling híbrido. 83
6.4 Possíveis estados do frustum culling híbrido. 84
6.5 Resultados do frustum culling híbrido nas plataformas. 86
6.6 Resultados do frustum culling híbrido no Boeing. 87
6.7 Resultados do frustum culling híbrido com render ligado. 88

Lista de tabelas

2.1 Tabela comparativa de alguns algoritmos de visibilidade. 25

3.1 Modelos paramétricos. 45
3.2 Modelos com malhas triangulares. 46

4.1 Radar X Planos. 49
4.2 Radar sem interseção X Radar com interseção para o Boeing. 50
4.3 Hierarquias utilizando mediana. 52
4.4 Hierarquias utilizando média. 52
4.5 Detalhes dos caminhos com hierarquias de média e mediana. 54
4.6 Detalhes dos caminhos com hierarquias de média e mediana. 54
4.7 Resultados obtidos no trabalho de Assarsson. 56
4.8 Melhor combinação de otimizações utilizadas em cada modelo. 59
4.9 SIMD nos cálculos de descarte da P-43. 60

5.1 Memória necessária para os volumes envolventes. 75

6.1 Comparação dos resultados entre melhor algoritmo em CPU e Híbrido. 87

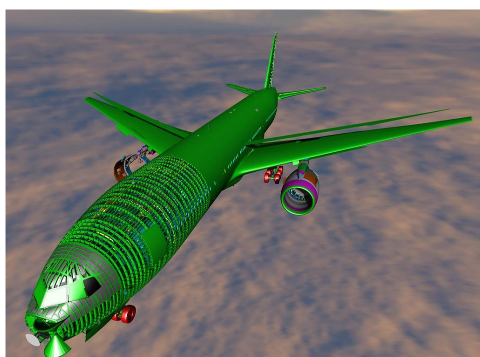
"A verdadeira felicidade está na própria casa,
entre as alegrias da família."

Leon Tolstoi, *Leon Tolstoi.*

1
Introdução

1.1
Motivação

As constantes buscas pela visualização de modelos maiores e com maior nível de detalhes têm motivado as indústrias de *software* e *hardware* a desenvolver novas soluções que os viabilizem. Mesmo não tendo um valor preciso, a literatura descreve que a interação em um sistema de visualização deve prover no mínimo 30 (*fps*)[1]. Quando estamos tratando de modelos massivos como o Boeing 777 1.1(a) (extraída de [20]) e Sunflowers 1.1(b) (extraída de [73]), que possuem uma enorme quantidade de polígonos e texturas, manter esta taxa de renderização se torna um desafio.



1.1(a): 350 milhões de triângulos.

1.1(b): 1 bilhão de triângulos.

Figura 1.1: Modelos Massivos.

Um dos grandes problemas é que a complexidade dos modelos cresce na mesma velocidade ou acima da *Lei de Moore*[2]. Outro fator que deve ser levado em conta é que mesmo que o poder de processamento computacional tente acompanhar a Lei de Moore, outros fatores influenciam no desempenho do

[1]fps - *frames per second* ou quadros por segundo mede a quantidade de imagens que são atualizadas no *display* no intervalo de 1 segundo.

[2]A Lei de Moore surgiu a partir de uma observação feita por Gordon Moore, co-fundador da Intel, em 1965, que o número de transistores por polegada quadrada estava dobrando a cada 18 meses desde a introdução dos circuitos integrados. Ele previu que esta tendência iria continuar nas próximas décadas, o que acabou tornando-se verdade.[64]

sistema, como tempo de acesso ao disco rígido e à memória, que não seguem a mesma evolução que os microprocessadores.

Diferentemente das CPUs (*Central Processing Units*), as GPUs (*Graphics Processing Units*) são processadores dedicados para a parte gráfica e são muito paralelizados. Nos últimos anos o poder de processamento das placas gráficas sofreu grande aumento, superando a lei de Moore. A cada nova geração de placas, novos recursos e algoritmos são incorporados aumentando o seu desempenho e tornando viável a renderização de cenas cada vez maiores.

Enquanto as CPUs atualmente podem trabalhar com oito núcleos simultaneamente, as GPUs mais modernas chegam a ter 320 *shaders processors* em paralelo. Uma rápida comparação no número de *flops* [3] das CPUs e GPU já dá uma boa ideia do poder de processamento das GPUS. Enquanto uma CPU Core 2 Extreme QX9650 chega a 48 *gigaflops* [4], uma GPU GeForce GTX 280 consegue 933 *gigaflops*.

A partir de 2001 as GPUs sofreram uma grande mudança, tornando alguns de seus estágios programáveis (antes tais estágios eram implementados diretamente em *hardware* sem nenhum tipo de acesso externo). Com isso, vários algoritmos novos foram desenvolvidos em diversas áreas. Uma área que surgiu com o acesso aos estágios do *pipeline* da placa gráfica aliada ao seu poder de processamento foi o *GPGPU* (*General-purpose computing on graphics processing units*).

Mesmo com a constante evolução das placas gráficas, a demanda por modelos com geometrias mais complexas, iluminação global e resolução de *display* cada vez mais alta ainda superam essa evolução. Por essas razões, muitos algoritmos têm sido desenvolvidos juntamente com a evolução do *hardware*. Alguns algoritmos tentam descartar objetos que estejam muito longe do observador utilizando técnicas como *fog* ou até mesmo eliminando-os, outros aumentam o nível de detalhes dos objetos gradativamente à medida que os objetos ficam mais próximos do observador (LOD-*level of detail*). Apesar de válidos em muitas aplicações, estas técnicas influenciam, mesmo que de forma mínima, na imagem final, o que não é desejado em várias situações.

Este trabalho vai focar em um dos algoritmos de visualização conhecido como *frustum culling*, tentando aplicar as técnicas de *GPGPU* para acelerar o descarte de objetos que não estejam visíveis.

[3]flops - *Floating point Operations Per Second* é uma medida de performance na computação, especialmente na área de cálculos científicos que utiliza cálculos em ponto flutuante.

[4]gigaflops- $10^9 flops$.

1.2 Objetivo da dissertação

Um dos primeiros objetivos deste trabalho é fazer uma investigação do estado da arte do algoritmo de *frustum culling*. A partir desta investigação, pretende-se acoplar a ele técnicas de aceleração a fim de obter o melhor algoritmo possível em CPU.

Como atualmente só se conhece implementação deste tipo de algoritmo em CPU e as GPUs estão se tornando cada vez mais poderosas e acessíveis para programação, um segundo objetivo deste trabalho é fazer uma implementação desse algoritmo em GPU e compará-la com o que foi obtido em CPU, analisando os pontos positivos e negativos.

Outro objetivo é implementar o descarte de objetos paramétricos diretamente em GPU. Tais objetos são conhecidos como *gpu primitives*. Diferentemente dos modelos naturais onde um objeto é representado por um conjunto de vértices, as *gpu primitives* são representadas por informações paramétricas que permitem sua renderização diretamente na placa gráfica. Aliado ao fato de que as *gpu primitives* são renderizadas diretamente na GPU, o cálculo de descarte será inserido neste *pipeline* de três maneiras diferentes e comparado com a implementação da CPU.

O último objetivo deste trabalho é implementar um *frustum culling* híbrido utilizando a GPU em momentos oportunos para determinar quais os objetos estão visíveis pela câmera. Tais gargalos podem diminuir consideravelmente o desempenho da aplicação dependendo da cena quando tratados apenas em CPU. Esta parte do trabalho tentará identificar os momentos oportunos para uso da GPU e melhorar o desempenho para que a interação não seja afetada.

1.3 Trabalhos relacionados

Os principais trabalhos relacionados a esta dissertação podem ser separados em dois grandes grupos. O primeiro engloba os estudos feitos em cima do algoritmo de *frustum culling* e técnicas de aceleração a ele aplicadas. Dentre eles, pode-se destacar [3], que desenvolveu algoritmos que aceleram os cálculos de descarte. Outro trabalho desenvolvido também por Assarsson *et al.* [4] tenta aplicar as técnicas de *frustum culling* em máquinas multiprocessadas. [48] removeu a necessidade da utilização de uma pilha para realizar o percurso da hierarquia. Mais recentemente [53] implementou uma nova forma de fazer descarte de objetos sem a necessidade de fazer extração de matrizes e cálculo de planos a cada *frame*. Além dos trabalhos citados, existem outros

trabalhos relacionados a esta dissertação que serão citados e discutidos com mais detalhes no capítulo 3.

O segundo grupo de trabalhos está relacionado aos algoritmos desenvolvidos para GPU. Como não é conhecida nenhuma implementação do algoritmo de *frustum culling* em GPU, os trabalhos relacionados deste grupo foram implementados para outros fins, porém a ideia básica foi aproveitada. Um exemplo é o trabalho de Thrane e Simonsen *et al.* [48], já citado anteriormente, que desenvolveu um percurso de hierarquia em GPU a priori para acelerar algoritmos de *ray tracing*. Um dos trabalhos que serviram de base para uma parte deste trabalho foi a tese de doutorado de Toledo [70], onde são apresentadas as *gpu primitives*. O Capítulo 5 explicará com mais detalhes esses trabalhos.

1.4 Organização da dissertação

Esta dissertação está dividida em sete capítulos. O próximo capítulo faz uma breve revisão sobre os algoritmos de determinação de visibilidade, dando destaque para os algoritmos de *culling*. Além desta breve revisão sobre os principais algoritmos, eles serão classificados segundo suas características.

O Capítulo 3 inicia apresentando o algoritmo de *frustum culling* básico. Este capítulo analisa algumas técnicas de aceleração utilizadas para melhorar o desempenho do algoritmo.

O Capítulo 4 tem por finalidade avaliar a implementação dos algoritmos e técnicas de aceleração levantadas no Capítulo 3 e ao final utilizar as que obtiveram melhor desempenho para servir de base de comparação com os algoritmos desenvolvidos em GPU.

O Capítulo 5 analisa a evolução do poder de processamento das placas gráficas ao longo dos tempos. Depois são propostos e implementados três algoritmos de *frustum culling* em GPU. São feitas análises e comparações em cima do algoritmo conseguido em CPU no capítulo anterior.

O Capítulo 6 explora a combinação entre o melhor dos mundos conseguido em CPU e em GPU, utilizando cada qual nos momentos onde conseguem melhor desempenho. Os resultados desta abordagem são comparados e analisados com as outras abordagens implementadas.

O Capítulo 7 analisa os resultados obtidos em todas as abordagens e comenta se os objetivos do trabalho foram alcançados. Também são sugeridos trabalhos futuros na área e as contribuições deixadas com este.

2
Algoritmos de Visibilidade

Este capítulo vai fazer uma rápida revisão em cima dos principais algoritmos de determinação de visibilidade, classificando-os segundo [14]. Também será identificada no *pipeline* a localização dos algoritmos de *frustum culling*, desenvolvidos neste trabalho, que serão discutidos com maiores detalhes nos Capítulos 3 e 5.

2.1
Determinação de visibilidade

Determinação de visibilidade é uma importante área da computação gráfica, que tem proporcionado muitos trabalhos nas últimas décadas. Sua função é determinar quais objetos estão visíveis pelo observador em uma cena 3-D. Os primeiros estudos nesta área foram os algoritmos de *hidden line removal*(HLR) [59] e posteriormente *hidden surface removal* (HSR). Com a evolução, os algoritmos de determinação de visibilidade começaram a tratar os objetos não visíveis pelo observador. As próximas subseções explicam com maiores detalhes estes dois grupos de algoritmos.

2.1.1
Hidden Surface Removal

Um dos algoritmos mais simples para solucionar o problema de visibilidade é conhecido como algoritmo do pintor. Este algoritmo tem esse nome por usar uma das técnicas utilizadas por pintores. A ideia básica é ordenar os polígonos na cena de acordo com a sua profundidade e renderizá-los do mais longe para o mais próximo do observador. O grande problema deste algoritmo é que ele pode falhar quando temos sobreposição de polígonos como está indicado na Figura 2.1. Um dos algoritmos mais conhecidos que trata a ordenação dos polígonos é conhecido como partição binária do espaço (BSP - *binary space partitioning*) [25] que divide a cena recursivamente utilizando planos.

O principal representante dos algoritmos de HSR é o *Z-Buffer*. Desenvolvido por [12], este algoritmo consiste em rasterizar os polígonos em uma ordem qualquer verificando o valor de profundidade de cada *pixel*. Caso o

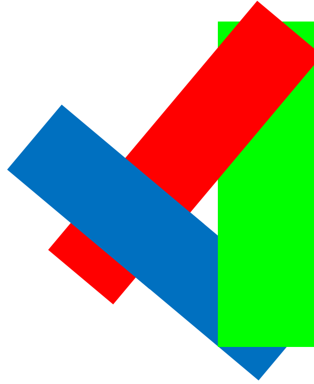


Figura 2.1: Problema com o algoritmo do pintor.

valor corrente seja menor que o valor já existente o *pixel* é descartado, caso contrário ele é mantido e seu valor é atualizado em um *buffer* chamado de *Z-Buffer*. Por ser simples de ser implementado em *hardware* e facilmente paralelizável, o *z-buffer* tornou-se o mais utilizado, atualmente presente em todas as placas gráficas.

O algoritmo de *ray tracing* soluciona o problema de visibilidade através do traçado de raios a partir do observador. A ideia do algoritmo é traçar raios a partir da câmera e o objeto mais próximo a ser interceptado pelo raio irá determinar a cor do *pixel*. O algoritmo de *ray tracing* foi desenvolvido por [2] e a partir de então recebeu várias contribuições visando melhorias de desempenho e qualidade.

2.1.2 Culling

A ideia dos algoritmos de *culling* é determinar a visibilidade dos objetos antes que eles sejam enviados para a placa gráfica e sejam processados desnecessariamente. Esses algoritmos não substituem os algoritmos de HSR, sendo o ideal que os dois tipos trabalhem em conjunto, onde os primeiros testes devem validar como visíveis os objetos e posteriormente os testes mais refinados trabalham no nível de *pixel*. Já com um número reduzido de objetos cabe ressaltar que os algoritmos de *culling* podem ser tratados no nível de triângulos como é o caso do *back-face culling* [33, 9], uma das primeiras técnicas desenvolvidas para na área de *culling*. Como a maioria dos objetos em uma cena 3-D é volumétrica e nem sempre é desejado visualizar todas as suas faces, essa técnica remove as faces voltadas para trás do objeto pela câmera. Atualmente essa técnica é implementada na maioria das placas gráficas.

Outra técnica de *culling* bem conhecida é chamada de *view-frustum*

culling que elimina os objetos que não estão dentro do *frustum*. Esse algoritmo será explicado com detalhes no Capítulo 3.

Em uma cena 3-D é muito comum que um objeto esteja na frente de outro a partir de um determinado ponto de vista. Esse problema pode ser tratado utilizando a técnica de *Z-Buffer*, porém terá que passar por todo o *pipeline* (Seção 2.3) até ser descartado na rasterização. Quando um objeto está totalmente encoberto por outro é interessante descartá-lo o quanto antes, pois ele não contribuirá em nada para imagem final. Quem trata esses casos é outro algoritmo de *culling* conhecido como *occlusion culling*. Mais detalhes sobre as diferentes técnicas de *occlusion culling* podem ser encontrados em [32] e [15].

Diferente das cenas *outdoor*, onde o algoritmo de *occlusion culling* proporciona bons resultados, o algoritmo de *portal culling* [40] traz otimizações em cenas *indoor*. Sua ideia básica é dividir a cena em setores e identificar a ligação entre eles chamadas de portais. Todos os objetos não visíveis entre setores diferentes a partir do ponto de vista do portal serão descartados.

O algoritmo de *detail culling*, também conhecido como *contribution culling*, tenta explorar o melhor equilíbrio entre qualidade e desempenho. Este algoritmo descarta os objetos que possuem volumes envolventes que ocupam uma pequena parcela da área da tela. Normalmente estes objetos estão muito longe da câmera e irão contribuir muito pouco para a imagem final. Métodos para estimar eficientemente a área ocupada pelo volume envolvente no espaço da tela podem ser encontrados em [62, 45].

A Figura 2.2 ilustra casos típicos dos cinco algoritmos de *culling* apresentados, onde a parte vermelha dos objetos pode ser descartada.

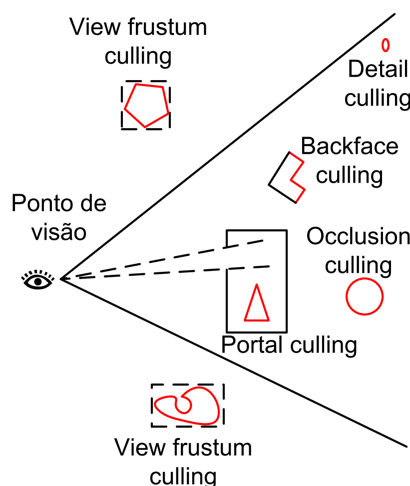


Figura 2.2: Técnicas de culling.

A próxima seção discute sobre a divisão dos algoritmos de visibilidade em classes feita por [14].

2.2 Classificação dos algoritmos de visibilidade

Cohen *et al.* [14] fizeram uma pesquisa em cima dos principais algoritmos de determinação de visibilidade desenvolvidos nos últimos anos, classificando-os segundo vários critérios. A seguir tais critérios são levantados juntamente com a descrição de cada uma das classes. Na Tabela 2.1, os algoritmos de *occlusion culling* mais conhecidos são comparados de acordo com as suas principais características. Em particular, neste trabalho, essa tabela foi modificada e estendida, com referências mais recentes.

Exato X Conservativo X Aproximado

- Exato - nesta classe de algoritmos a determinação de quais os polígonos estão visíveis é feita de forma precisa. Apesar de prover um resultado exato, este tipo de algoritmo é caro e não é o mais comum em aplicações interativas 3-D. Esta classe não está citada no trabalho de [14], porém houve a necessidade de incluí-la para categorizar, por exemplo, os algoritmos de *ray tracing* e *Z-Buffer*;
- Conservativo - bem mais tolerantes e mais rápidos que os algoritmos exatos, estes algoritmos removem uma grande parte dos polígonos não visíveis, porém nem todos. Os polígonos enviados erroneamente são tratados pela GPU posteriormente. Esta classe constitui grande parte dos algoritmos de visibilidade. A maioria dos algoritmos de *frustum culling* encontram-se nesta classe;
- Aproximado - diferentemente dos algoritmos exatos e conservativos, estes algoritmos podem remover polígonos que estejam visíveis, produzindo imagens erradas. Dependendo do tipo de aplicação pode se tornar interessante a utilização de algoritmos aproximados, mesmo não produzindo imagens certas, pois eles tem um melhor desempenho quando comparados com as outras duas classes. Como representante desta classe de algoritmos podemos citar [63] que utiliza probabilidade para determinação de visibilidade.

Oclusores totais X Oclusores parciais Esta classe de algoritmo separa as técnicas de *occlusion culling* em dois grupos. A primeira trata todos os objetos da cena como possíveis oclusores, enquanto o segundo grupo trata apenas uma parte dos objetos seguindo algum critério.

Oclusores individuais X Oclusores colapsados Este critério separa os algoritmos de *occlusion culling* que utilizam objetos individuais como oclusores dos que colapsam diversos objetos para formar um oclusor.

Pré-processados X Online Enquanto os algoritmos *online* fazem todos os cálculos de visibilidade a cada *frame*, os pré-processados fazem cálculos e guardam estruturas auxiliares antes da execução do algoritmo, a fim de aumentar o desempenho.

2-D X 3-D Enquanto alguns algoritmos utilizam o espaço da imagem, onde os cálculos são feitos em uma projeção 2-D de um ambiente 3-D, outros utilizam o espaço do objeto para processar os cálculos de visibilidade quando a projeção no espaço 2-D é feita após os cálculos.

Software X Hardware Muitos algoritmos estão tirando proveito do poder de processamento das placas gráficas e estão portando parcialmente ou totalmente seus algoritmos para serem processados em *hardware*. Outros ainda são implementados puramente em *software* por não ter algoritmo eficiente em *hardware*.

Cenas Dinâmicas X Cenas Estáticas

- Dinâmicos - algoritmos que tratam cenas onde os objetos se movimentam em relação aos outros. Esses algoritmos têm a dificuldade extra na determinação de visibilidade, uma vez que normalmente envolve atualização de dados necessários nos cálculos;
- Estáticos - classe de algoritmos que não envolvem animação na cena. Esses algoritmos normalmente utilizam estruturas de dados auxiliares construídas em pré-processamento que não sofrem mudanças, acelerando assim os cálculos.

O algoritmo de *frustum culling*, utilizado neste trabalho, classificado segundo os critérios citados por [14], quanto à acurácia é conservativo, pois alguns polígonos não visíveis são enviados para a GPU, porém nenhum visível é descartado. Quanto à estrutura de dados utilizada podemos classificar como pré-processado, à medida que é criada uma hierarquia para acelerar o algoritmo. A criação da hierarquia é feita no espaço do objeto e as cenas de testes são todas estáticas. Quanto à execução do algoritmo, temos as duas partes neste trabalho. A primeira totalmente em *software* referente ao Capítulo 3, a segunda utiliza apenas o *hardware* no Capítulo 5 e uma terceira abordagem que utiliza tanto o algoritmo em *software* quanto em *hardware*

descrito com mais detalhes no Capítulo 6. As classificações dos oclusores (parciais ou totais e individuais ou colapsados) não se aplicam ao *frustum culling*, pois são específicos da oclusão.

A próxima seção identificará as localizações do algoritmo de *frustum culling* desenvolvidos neste trabalho, assim como destacará os outros algoritmos que também serão utilizados nos programas de testes.

2.3 Pipeline dos algoritmos de visualização

A renderização das cenas utilizadas neste trabalho é feita através da API (*Application Programming Interface*) gráfica OpenGL. Desenvolvido no final da década de 80 por Kurt Akeley, a biblioteca OpenGL, escrita na linguagem C, provê uma série de funcionalidades para a criação de aplicações gráficas 2-D e 3-D, utilizando comunicação cliente-servidor com a placa gráfica. Neste trabalho estamos interessados na renderização de cenas 3-D. Para que um objeto seja visualizado na tela, ele deve ser geometricamente descrito por polígonos cujos vértices passam por vários estágios em uma fila conhecida como *pipeline*. A Figura 2.3 mostra o *pipeline* do OpenGL na transformação dos polígonos em *pixels*. Cada uma destas etapas envolve uma série de cálculos. Mais detalhes sobre o *pipeline* pode ser encontrado em [5].

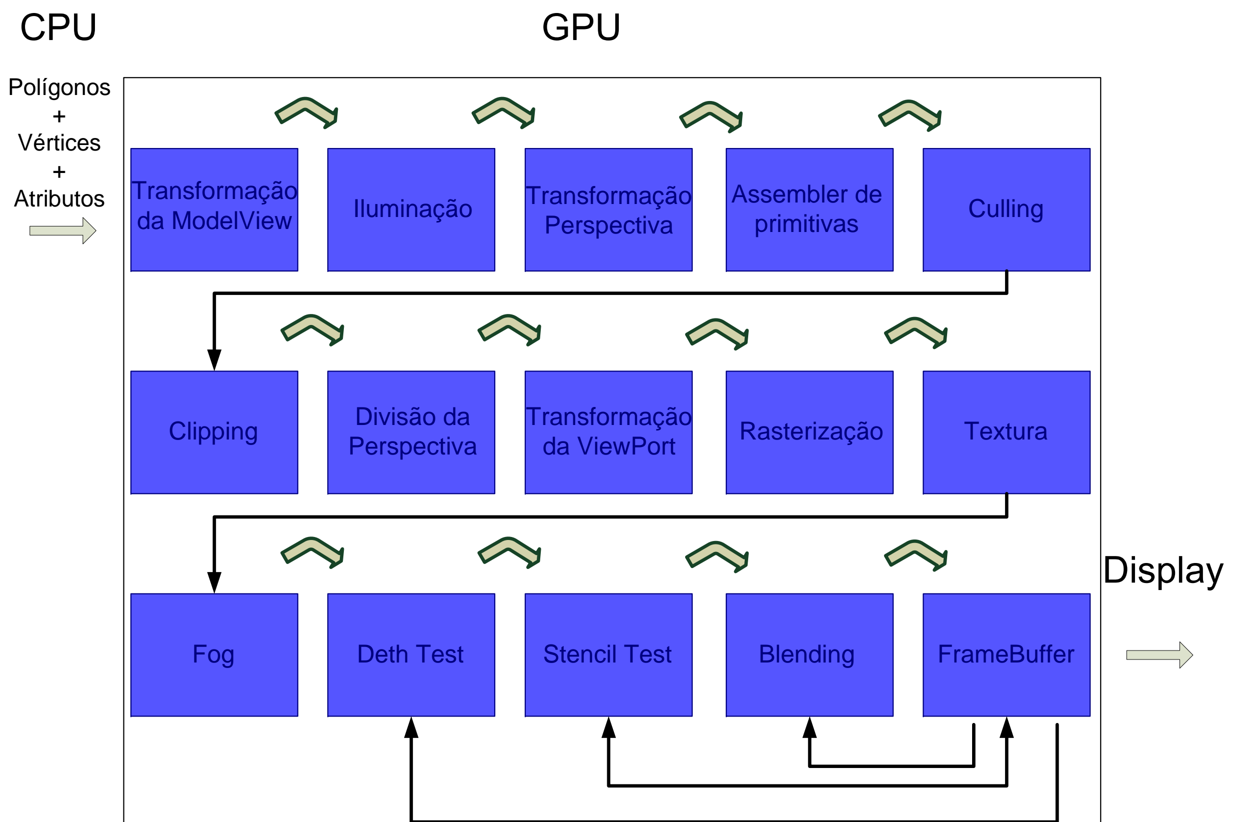


Figura 2.3: Pipeline do OpenGL.

Antes da primeira etapa do processamento dos vértices ocorrer, eles são enviados para placa gráfica, o que já consome uma parcela do tempo. Posteriormente, ocorrem transformações em geral e cálculos de iluminação por vértices. Nas próximas etapas ocorrem as operações de *culling* e *clipping*, que eliminam os vértices que não serão visíveis na imagem final. A partir da etapa de rasterização todos os polígonos, linhas e pontos são convertidos em valores de *pixels* para serem enviados para o *display* posteriormente.

Além das funcionalidades básicas para a criação de aplicações gráficas, a biblioteca OpenGL também possui comandos destinados à otimização, como o *backface culling*. Este algoritmo normalmente é executado antes da realização do algoritmo de *clipping* por se tratar de uma operação rápida e irá afetar uma percentagem maior de triângulos que o algoritmo de *clipping*.

Essas duas técnicas, *clipping* e *backface culling*, normalmente melhoram o desempenho das aplicações à medida que são descartadas partes desnecessárias da cena num determinado momento. Porém existem situações em que ainda são feitos cálculos desnecessariamente como, por exemplo, quando um objeto está fora do volume de visão. Nesse caso, os vértices do objeto são enviados para a placa, ocorrem transformações e cálculos de iluminação para que apenas no estágio de *clipping* eles sejam descartados. Quando temos uma quantidade pequena de polígonos, passar por todas essas etapas não é perceptível, porém quando tratamos de modelos massivos, o desempenho pode ser afetado. Para isso foi criado o algoritmo de *frustum culling*, que é processado antes dos dados serem enviados para a placa gráfica. A Figura 2.4 ilustra a inserção do algoritmo de *frustum culling* no *pipeline* reduzido do OpenGL. Este *pipeline* se refere aos estágios programáveis da GPU que serão discutidos com mais detalhes no Capítulo 5.

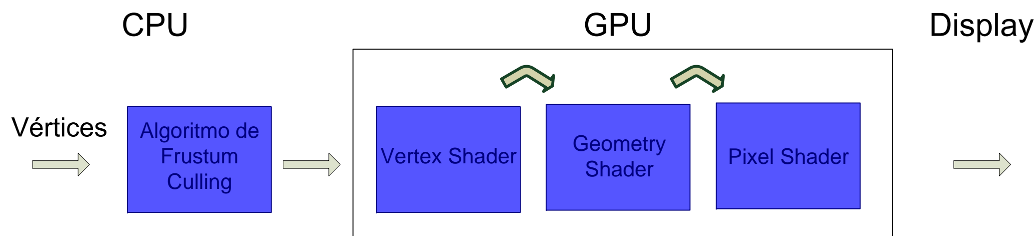


Figura 2.4: Frustum culling em CPU.

Além do algoritmo de *frustum culling* básico que será discutido com mais detalhes no próximo capítulo, este trabalho também apresenta três novas formas de *frustum culling*, onde o descarte é feito diretamente na GPU. Uma delas pode ser vista na Figura 2.5. Mais detalhes sobre a implementação do

frustum culling em GPU serão dados no Capítulo 5, onde serão discutidos as vantagens e desvantagens destas implementações.

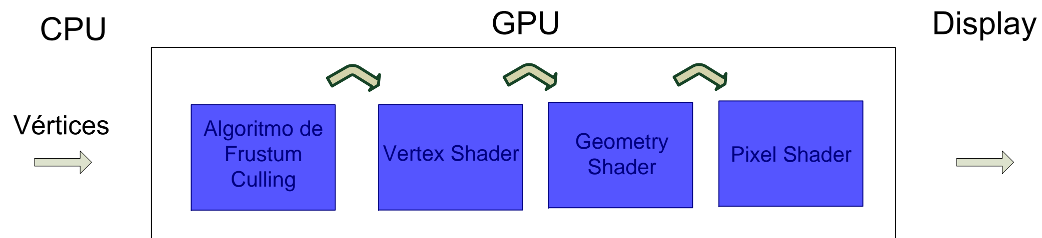


Figura 2.5: Frustum culling em GPU.

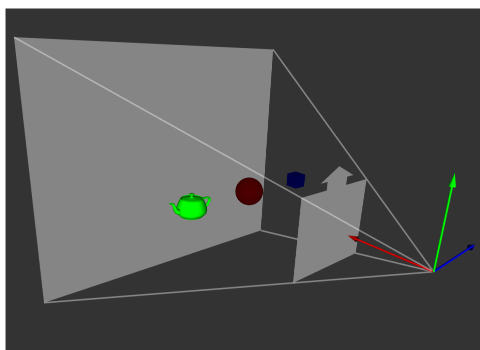
Método de occlusion culling	Espaço 2D/3D	Precisão do Métodos	Tipos de Obstáculos	Realiza Pré-processamento	Utiliza a GPU	Ambiente Dinâmico
Métodos baseados em regiões que exploram estruturas de células e portais						
Airey *et al.* [1]	2D/3D	pode ser conservativo	portais	PVS[1]	não	não
Teller *et al.* [68]	2D/3D	conservativo	portais	PVS	não	não
Métodos baseados em pontos e trabalham no espaço do objeto						
Luebke e Georges *et al.* [40]	3D	conservativo	portais	conetividade de células	não	atualiza a estrutura
Coorg e Teller *et al.* [15]	3D	conservativo	grandes conexos e agrupados	seleção de obstáculos	não	atualiza a estrutura
Hudson *et al.* [32]	3D	conservativo	grandes conexos e não agrupados	seleção de obstáculos	não	atualiza a estrutura
Brittner *et al.* [8]	3D	conservativo	grandes e agrupados	seleção de obstáculos	não	atualiza a estrutura
Klosowski e Silva *et al.* [34]	3D	aproximado	todos os objetos agrupados	volumétrico	não	atualiza a estrutura
Métodos baseados em pontos e trabalham no espaço da imagem						
Greene *et al.* [28]	3D	conservativo	todos os objetos e agrupados	não	*Z-Buffer*	sim
Zhang *et al.* [77]	3D	conservativo ou aproximado	grandes subconjuntos e agrupados	banco de dados de obstáculos	*Z-Buffer* e TM[2]	atualiza a estrutura
Bartz *et al.* [6]	3D	aproximado	todos os objetos e agrupados	não	*buffer* secundário	atualiza a estrutura
Wonka *et al.* [75]	2.5D	conservativo	grandes subconjuntos e agrupados	não	*Z-Buffer*	sim
Bernardini *et al.* [7]	3D	conservativo	pré-processado e agrupados	obstáculos	não	não
Klosowski *et al.* [35] e Silva	3D	conservativo	todos os objetos e agrupados	volumétrico	sim	atualiza a estrutura
Métodos baseados em regiões						
Schaufler *et al.* [61]	2D e 3D	conservativo	todos os objetos e agrupados	PVS	não	atualiza a estrutura
Durand *et al.* [22]	3D	conservativo	grandes subconjuntos e agrupados	PVS	sim	atualiza a estrutura
Koltun *et al.* [36]	2D e 2.5D	conservativo	grandes subconjuntos e agrupados	obstáculos virtuais	não	atualiza a estrutura
Wonka *et al.* [76]	2D	conservativo	grandes subconjuntos e agrupados	PVS	sim	atualiza a estrutura
Koltun *et al.* [37]	2.5D	conservativo	todos os objetos e agrupados	não	sim	atualiza a estrutura
Nirenstein *et al.* [49]	3D	conservativo	todos os objetos	PVS	não	não
Moreira *et al.* [47]	3D	conservativo	todos os objetos	PVS	não	atualiza a estrutura
Leyvand *et al.* [39]	2.5D	conservativo	todos os objetos	PVS	sim	atualiza a estrutura
Mora *et al.* [46]	3D	exato	todos os objetos	PVS	não	atualiza a estrutura
Haumont *et al.* [31]	2D e 3D	conservativo	todos os objetos	não	não	não
Laine *et al.* [38]	2D e 3D	aproximado e conservativo	todos os objetos	PVS	não	não

[1]PVS - *Potentially Visible Set* identifica a área potencialmente visível a partir de um referencial [2]TM - *Texture-Mapping*

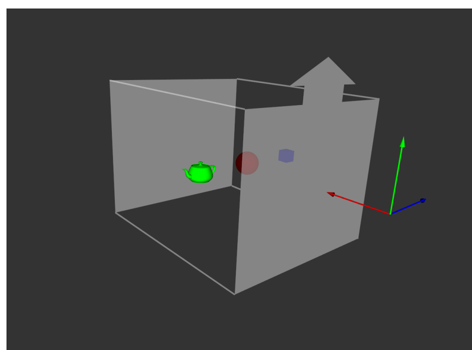
Tabela 2.1: Tabela comparativa de alguns algoritmos de visibilidade.

3
Frustum culling

Uma das áreas de pesquisa mais estudadas dentro da computação gráfica é a determinação de quais objetos estão visíveis dentro de uma cena 3-D. Nos jogos este problema é bem comum, pois o observador só está vendo uma parte do mundo virtual. Como foi descrito no capítulo anterior, a técnica de *frustum culling* faz parte da classe dos algoritmos de visibilidade. O ponto de visão em uma cena 3-D é modelado por câmeras que possuem atributos como posição, vetores *up*, *right* e *direction*, ângulo de abertura, entre outros. Na Figura 3.1, a posição da câmera está sendo representada pelo sistema de coordenadas e os vetores *up*, *right* e *direction* pelos eixos verde, azul e vermelho respectivamente.



3.1(a): Projeção perspectiva



3.1(b): Projeção paralela

Figura 3.1: Tipos de Projeção.

Quanto ao tipo de projeção, as câmeras podem ser classificadas como ortográficas, perspectivas ou não lineares. O volume de visão de uma projeção ortográfica (paralela) é um paralelepípedo. Ela é utilizada normalmente em aplicações de engenharia e arquitetura, pois os tamanhos e ângulos dos objetos são preservados. Já a projeção perspectiva não mantém tais propriedades, fazendo com que um objeto perto do observador torne-se maior que o mesmo objeto longe do observador. Seu volume de visão é representado por um *frustum* (tronco de pirâmide), sendo mais comum em aplicações que simulam imersão. Os planos formados pelas câmeras perspectivas são conhecidos como *near*, *far*, *left*, *right*, *top* e *bottom*. A projeção não linear, menos comum que as outras duas, recebe este nome por não poder ser representada por transformações

lineares. Este tipo de projeção é utilizada para criar efeitos especiais [60]. Neste trabalho utilizaremos apenas a projeção perspectiva, porém as ideias podem ser aplicadas também na projeção ortográfica. Mais detalhes sobre a projeção ortográfica podem ser encontrados em [74, 11]. A Figura 3.1 mostra os dois tipos de projeções mais comuns.

Este capítulo introduzirá a ideia básica do algoritmo de *frustum culling*, acrescentando as técnicas desenvolvidas ao longo do tempo para acelerar os testes. Ao final do capítulo esperamos ter o estado da arte do algoritmo a fim de poder compará-lo com as outras abordagens desenvolvidas neste trabalho.

3.1
Implementação clássica

A ideia do algoritmo de *frustum culling* é descartar a maior parte dos objetos que estão fora da região do *frustum*, evitando que primitivas geométricas localizadas fora do volume de visão sejam renderizadas. O teste de descarte pode resultar em três situações:

1. Objeto totalmente dentro do *frustum*
2. Objeto interceptando o *frustum*
3. Objeto totalmente fora do *frustum*

De acordo com [45], apenas as primitivas que estiverem totalmente ou parcialmente dentro do volume de visão precisam ser renderizadas, uma vez que as outras partes não irão contribuir para a imagem final, pois serão eliminadas pelo *pipeline* da placa gráfica. Como o *pipeline* tradicional não faz processamento em cima de objetos e malhas poligonais, o estágio de *clipping* processará cada um dos polígonos individualmente, tornando assim interessante descartar objetos não visíveis o mais cedo possível.

Para identificar se um polígono esta visível no *frustum* é necessário calcular os seis planos que o compõem (*near*, *far*, *left*, *right*, *top*, *down*) a cada *frame* que a câmera muda a posição ou orientação. Os cálculos para obtenção dos planos utilizando *OpenGL* e *DirectX* podem ser encontrados em [29]. Dada a equação do plano $Ax + By + Cz + D = 0$, para determinarmos se um ponto qualquer p, no espaço 3-D, está dentro do *frustum* a seguinte equação deve ser satisfeita:

$A * px + B * py + C * pz + D <= 0$, onde $(p_{\mathrm{X}}, p_{\mathrm{Y}}, p_{\mathrm{Z}})$ correspondem às coordenadas do ponto p no espaço 3-D.

Como temos seis planos, no pior caso todos eles deverão ser testados contra o ponto. Porém, a partir do momento em que descobrimos que o ponto

está fora de um dos planos, não há necessidade de continuar testando os outros. O correspondente pseudo-código é mostrado na Figura 3.2.

```
Resultado Interseção::entre(const Planos& planos,
                            const Ponto& ponto)
{
  para cada um dos planos
  {
    se distancia entre ponto e plano < 0
      return FORA;
  }

  return DENTRO;
}
```

Figura 3.2: Pseudo-código de descarte de pontos.

3.2 Estado da arte

Tendo como objetivo principal a incorporação do algoritmo de *frustum culling* na placa gráfica, verificou-se a necessidade de termos um algoritmo com bom desempenho em CPU que pudesse servir de parâmetro de comparação com as novas abordagens desenvolvidas. Para isso foram estudadas várias técnicas de aceleração a fim de termos o melhor algoritmo em CPU.

3.2.1 Volumes Envolventes

Para que um objeto qualquer seja considerado não visível, todos os seus vértices devem estar fora do *frustum*. Caso os testes sejam feitos individualmente em um objeto muito denso, o custo da execução do algoritmo de *culling* poderá ser maior do que a própria renderização. Para contornar esse problema são utilizados os volumes envolventes (*bounding volumes*). A ideia básica é ter um objeto mais simples, normalmente esfera ou caixa, que englobe toda a geometria do objeto original, servindo apenas para acelerar a execução de um algoritmo, não contribuindo assim para a imagem final. Além de esferas e caixas, existem vários tipos de volumes envolventes como cilindros, prismas [54], elipsóides, *k-dops*, entre outros. A Figura 3.3 mostra que elevando a complexidade do volume envolvente, o ajuste com a geometria do objeto melhora, porém aumenta o custo dos testes. Mais detalhes do impacto da complexidade de volumes envolventes sobre o algoritmo de *frustum culling* podem ser encontrados em [17, 72].

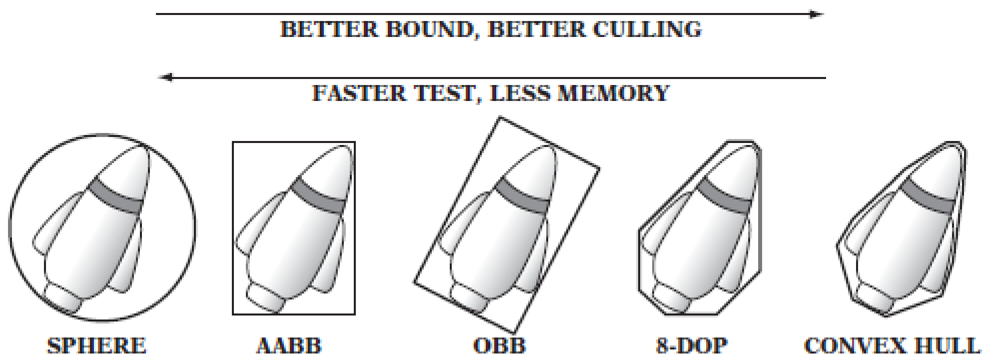


Figura 3.3: Volumes envolventes, imagem divulgada por Ericson *et al.* [23]

Este trabalho utiliza como volume envolvente a caixa alinhada com os eixos (AABB - *axis aligned boundind box*). Outra caixa bastante utilizada é a OBB (*oriented bounding box*)[45], muito parecida com a AABB diferindo apenas de uma orientação. A AABB foi escolhida por utilizar menos memória que a OBB, por possuir construção mais rápida e mais otimizações nos testes contra o *frustum* que serão vistas mais à frente. A caixa alinhada com os eixos possui seis valores $p^{max} = (p_x^{max}, p_y^{max}, p_z^{max})$ e $p^{min} = (p_x^{min}, p_y^{min}, p_z^{min})$ que vão corresponder aos valores limites dos vértices do objeto em cada eixo. A Figura 3.4 mostra o pseudo-código do teste da AABB contra os planos do *frustum*.

Este teste segue a mesma ideia do teste com pontos, adicionando apenas a condição de parada caso todos os pontos da AABB estejam totalmente fora de um dos planos. Este teste poderia retornar apenas visível ou invisível, porém determinar se está ocorrendo interseção é primordial quando estamos utilizando hierarquia (Seção 3.2.3).

3.2.2 Heurísticas de culling

O algoritmo mais comum para fazer o descarte de volumes envolventes utiliza, em câmera perspectiva, uma pirâmide truncada para determinar quais são os limiares da cena. Os cálculos que envolvem esses limites são a extração dos planos desta pirâmide e a partir destes determinar se os objetos estão visíveis ou não. Mais recentemente, uma nova abordagem para determinação de visibilidade foi desenvolvida por Placeres *et al.* [53], que ficou conhecida como radar. Mais conservativo que o algoritmo que utiliza planos, a ideia básica na utilização do radar é guardar alguns parâmetros da câmera e através deles calcular os limites do *frustum* que serão testados contra os objetos. Inicialmente para a construção da câmera é necessário termos o ângulo de visão (FOV - *field-of-view*) horizontal e vertical, e as distâncias para os planos

```
Resultado Interseção::entre(const Planos& planos,
                            const Caixa& caixa)
{
  Resultado resultado = DENTRO;
  para cada um dos planos
  {
    para cada um dos vertices da AABB
    {
      se distancia entre vertcice e plano < 0
        fora++;
      caso contrario
        dentro++;
    }

    se dentro == 0
      returna FORA;
    se fora
      resultado = INTERCEPTANDO;
  }

  return resultado;
}
```

Figura 3.4: Teste de descarte de AABB.

de *near* e *far*. A partir do ângulo na horizontal é possível calcular os valores limitantes da câmera à esquerda e à direita, e para cima e para baixo no caso do ângulo vertical. As distâncias dos planos servem para verificar se o objeto está entre os planos de *near* e de *far*. Esses valores só precisam ser inicializados na construção da câmera ou quando forem modificados, o que não é muito comum. A Figura 3.5 ilustra a classificação de um ponto P contra um sistema de radar em 2-D.

$$rfactor = \tan(\frac{FOV}{2}) = \frac{ladooposto}{ladoadjacente} = \frac{rLimit}{f}$$

$$rfactor = \frac{rLimit}{f}$$

$$rLimit = rfactor * f$$

Em duas dimensões o *frustum* se torna um triângulo e, para testarmos se um ponto P qualquer está dentro dele, é suficiente verificar os limites *-rLimit*, *rLimit*, *near* e *far*. A mesma ideia pode ser estendida para um sistema 3-D.

Quando ocorrem translações e rotações da câmera é necessário atualizar a posição da câmera e dos vetores *forward*, *right*, *up*. Após esta atualização os testes de *culling* podem ser realizados. O primeiro teste no espaço 3-D,

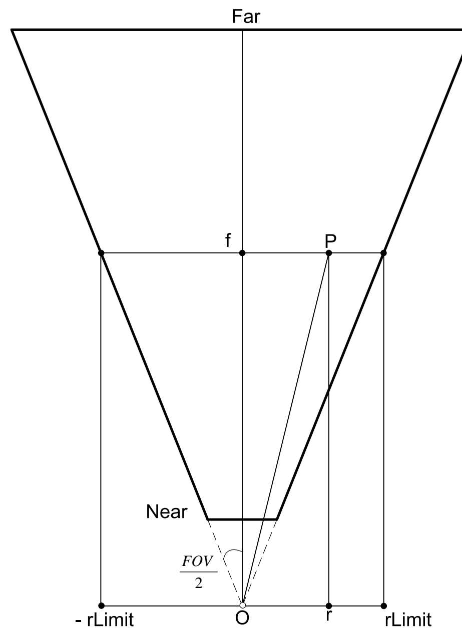


Figura 3.5: Classificação de um ponto P contra o frustum.

referente ao vetor *forward*, deve identificar se o objeto está dentro dos limites do *near* e do *far*. O segundo teste, referente ao vetor *right*, verifica se o objeto está visível à esquerda e à direita da câmera. O último teste, que leva em conta o vetor *up*, identifica se o objeto está visível para as partes de cima e de baixo da câmera. Caso um destes testes falhe, podemos classificar o objeto como não visível. A Figura 3.6 mostra o teste de *culling* feito para um ponto.

```
Resultado Interseção::entre(const Radar& radar,
                             const Ponto& ponto)
{
    se ponto estiver fora dos limites do radar
      return FORA;

    return DENTRO;
}
```

Figura 3.6: Teste de pontos contra radar.

O teste para volumes envolventes segue a mesma ideia do teste feito para pontos. A única diferença é a possibilidade de haver interseção entre o volume de visão e o volume envolvente. A Figura 3.7 apresenta o pseudo-código do teste de *culling* para AABB.

Esse pseudo-código retorna as três opções possíveis para o teste do volume envolvente contra o *frustum* (dentro, interseção ou fora). Quando não há a utilização de hierarquia é suficiente retornar apenas se o volume esta dentro ou fora do campo de visão. Com isso o pseudo-código acima pode ser otimizado

```
Resultado Interseção::entre(const Radar& radar,
                            const Caixa& caixa)
{
    para todos os vertices da caixa
    {
        se vertice estiver fora dos limites do radar
          ++fora;
    }

    se fora == 0
      return DENTRO;

    se fora == 8
      return FORA;

    return INTERCEPTANDO;
}
```

Figura 3.7: Teste de AABB contra radar.

para que a cada teste, se o volume estiver dentro de um dos limites da câmera, o *loop* seja interrompido. Os resultados com as duas situações são descritas com mais detalhes no capítulo 4.

3.2.3 Hierarquia

Como foi dito anteriormente, fazer testes utilizando volumes envolventes ao invés da própria geometria pode ser muito vantajoso. Porém ainda é necessário testar todos os volumes envolventes individualmente para determinar se estão visíveis, podendo transformar-se no gargalo da aplicação dependendo do tipo de cena. [13] utilizou a ideia de divisão do espaço 3-D em uma árvore para determinar a visibilidade de superfícies. Neste trabalho, as construções das hierarquias utilizou AABB como volume envolvente e árvore binária como estrutura hierárquica. A árvore binária foi escolhida para forçar mais cálculos de interseção e facilitar a determinação do algoritmo mais eficiente.

As duas formas mais conhecidas de divisão de uma cena 3-D no espaço do objeto são conhecidas como *spatial partitioning* e hierarquia de volumes envolventes (BVH - *bounding volume hierarchies*). Os principais algoritmos no grupo *spatial partitioning* são *BSP trees*, *octrees*, *hierarchical grids* e kd-trees [44]. Normalmente esses algoritmos dividem a cena recursivamente em regiões através de planos onde os objetos são arrumados de acordo com o lado do plano pertencente.

A ideia básica do BVH, utilizado neste trabalho, é que o volume envolvente do nó pai englobe o dos filhos, e os nós folhas possuam as informações da geometria dos objetos (Figura 3.8). Dessa maneira, durante o percurso [1] da hierarquia, caso o nó pai esteja fora ou totalmente dentro do *frustum*, é garantido que seus filhos também estejam. Quando ocorre interseção do volume envolvente com *frustum*, é necessário descer na hierarquia para determinar quais nós filhos estão visíveis. Caso a interseção chegue até o nó folha, ele deve ser rotulado como visível e o estágio de *clipping* irá determinar quais dos seus polígonos estão visíveis.

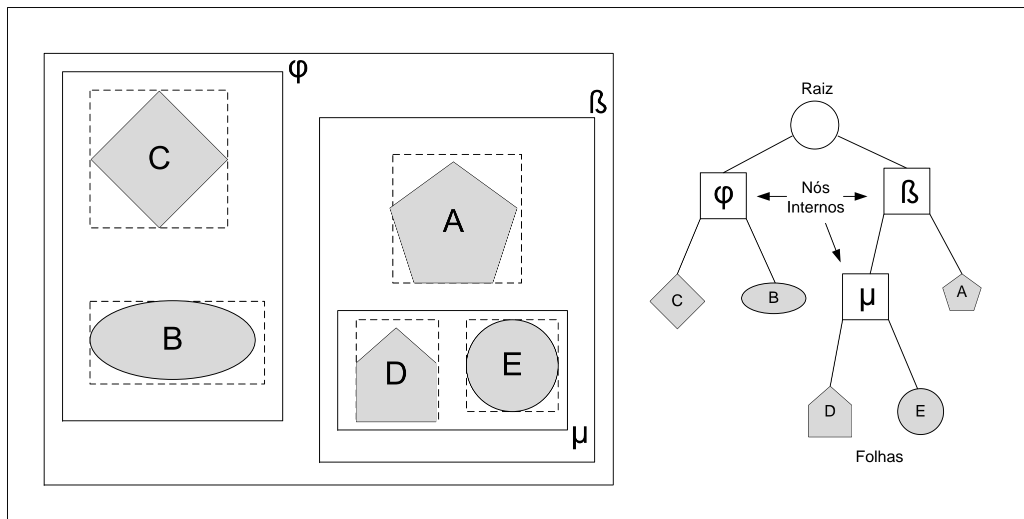


Figura 3.8: Hierarquia de volumes envolventes.

Existem várias formas para construção de hierarquia, cada uma se adequando melhor a cenários distintos. Normalmente para cenários estáticos, a hierarquia é construída em pré-processamento, enquanto em cenas dinâmicas a hierarquia é reconstruída durante a execução do programa. [43] enfatiza três estratégias de construção de hierarquia:

– *Top-down* - a estratégia *top-down* constrói a árvore a partir de uma lista de nós contendo, cada um, os volumes envolventes dos objetos. A cada iteração a lista é separada em dois grupos, gerando dois novos nós. O procedimento é repetido recursivamente até que só sobre um objeto em cada nó ou alguma outra condição de parada retorne verdadeiro. Além das condições de parada, existem outras decisões que devem ser tomadas ao longo da recursão que vão influenciar diretamente na qualidade da

[1] *percurso* - visitar nós da hierarquia, onde as duas maneiras mais conhecidas são em largura e profundidade. Neste trabalho foi utilizado o percurso em profundidade. Mais detalhes podem ser encontrados em [69].

árvore, tais como estratégia de particionamento, escolha do eixo e ponto de corte.

- *Bottom-up* - este tipo de construção inicia com uma lista de volumes envolventes (nós folha contendo geometria) e a cada iteração dois elementos são escolhidos segundo algum critério (*clustering rule*) e um nó pai é criado a partir deles. Em seguida os nós filhos selecionados são retirados da lista e o novo nó (pai) adicionado. O processo termina quando a lista tiver apenas um elemento (nó raiz).
- *Insertion* - esta estratégia constrói a hierarquia inserindo nós de forma incremental na árvore. O nó a ser inserido percorre a árvore até encontrar a sua localização ideal. Caso seu volume envolvente seja maior que os existentes, sua localização tenderá a ficar perto do nó raiz, caso contrário irá se aproximar do final da hierarquia. Este tipo de construção é muito utilizado em aplicações dinâmicas, onde há a necessidade de atualizar a hierarquia constantemente, pois não é necessário reconstruir toda a hierarquia quando houver mudanças.

Mesmo não existindo estratégia perfeita de construção de hierarquia para todas as cenas, [51] afirma que a estratégia *top-down*, gera árvores de pior qualidade quando comparada com a estratégia *bottom-up* na maioria dos casos. Outra característica da estratégia *bottom-up* citada é que sua implementação é mais complicada e tem tempo de construção superior a *top-down*. Neste trabalho as construções de BVH utilizaram a estratégia *top-down*, uma vez que os modelos de testes são todos estáticos. A Seção 4.3 demonstra as decisões tomadas para construção das hierarquias assim como seus resultados comparativos.

3.2.4 Otimizações

Vários algoritmos têm sido criados a fim de acelerar os cálculos de *frustum culling*. Um desses algoritmos reduz o número de testes dos planos contra as caixas envolventes. Ao invés de testar todos os oito vértices da caixa, apenas dois são necessários para determinar o estado do volume envolvente [30, 27]. Os dois vértices que serão testados são os que estiverem mais distantes nas direções positivas (*p-vertex*) e negativas (*n-vertex*) da normal do plano a ser testado. A Figura 3.9 ilustra a identificação dos vértices n e p em dois casos.

Caso a distância entre o vértice n e plano for positiva, então toda a caixa está dentro do plano e nenhum teste adicional precisa ser feito. Se a distância do vértice p retornar um valor negativo então a caixa está fora do plano. Quando

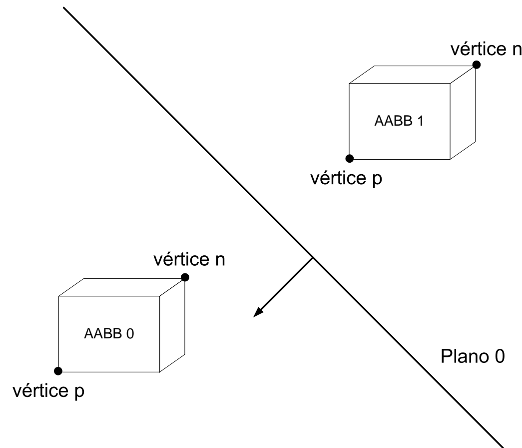


Figura 3.9: Classificação dos vértices n e p.

a caixa está interceptando o plano, o contrário acontece: a distância do plano para o vértice n é negativa e para o vértice p é positiva. Esta otimização só pode ser feita em AABBs [3].

Assarsson e Möller *et al.* [3] desenvolveram quatro otimizações para o algoritmo de *frustum culling*. A fim de achar a melhor combinação entre eles, foram feitos testes utilizando caminhos de câmera em cenas estáticas. As otimizações implementadas foram *plane-coherency test*, *octant test*, *masking* e *TR coherency test*.

– *plane-coherency test* - a ideia básica desta otimização é explorar a coerência temporal da câmera. Como a movimentação da câmera normalmente é suave, não ocorrem grandes mudanças em *frames* consecutivos. Caso um determinado objeto esteja fora de um dos planos do *frustum* em um *frame* x, provavelmente estará fora no *frame* $x + 1$, com isso este plano deverá ser testado primeiro. Para guardar as informações da ordem dos planos a serem testados são utilizadas informações adicionais em cada um dos nós da hierarquia de volumes envolventes.

– *octant test* - esta otimização explora os *frustum* simétricos, para diminuir o número de planos a serem testados. Dado uma esfera envolvente e um *frustum* simétrico dividido em oito partes, é possível determinar em qual dos octantes se encontra a esfera envolvente a partir de seus centros. Uma vez identificada a sua localização é suficiente testar apenas os três planos externos para determinar se a esfera envolvente está visível. A mesma ideia pode ser aplicada a outros volumes envolventes [3].

– *masking* - durante o percurso da hierarquia, caso um volume envolvente estiver completamente dentro de um dos planos do *frustum*, seus filhos

também estarão e não há necessidade de testá-los contra este plano. A comunicação entre os nós da hierarquia, sobre quais os planos que não precisam ser testados, pode ser feita através de *bit fields.* [2].

– *TR coherency test* - esta otimização explora a coerência temporal em aplicações onde em alguns momentos a movimentação da câmera se restringe à rotação em apenas um eixo ou à translação. Dado que um objeto está fora do *frustum* e é conhecido o fator de translação e rotação entre *frames*, é possível determinar sem cálculos de interseção se o objeto está visível.

Além das otimizações vistas até agora, existem testes rápidos que podem ser feitos antes de maneira simples e eficiente. Basicamente os testes são feitos calculando o volume envolvente do *frustum* de visão e testando-o contra os volumes dos objetos. Tais testes são úteis quando os objetos estão totalmente invisíveis, uma vez que testes mais caros não precisarão ser feitos. Quando o teste retorna interseção ou visível, como o volume envolvente é uma aproximação grosseira do *frustum*, testes mais apurados devem ser feitos para determinar a visibilidade dos objetos. A Figura 3.10 mostra a AABB do *frustum* (em verde) construída a partir dos pontos em azul.

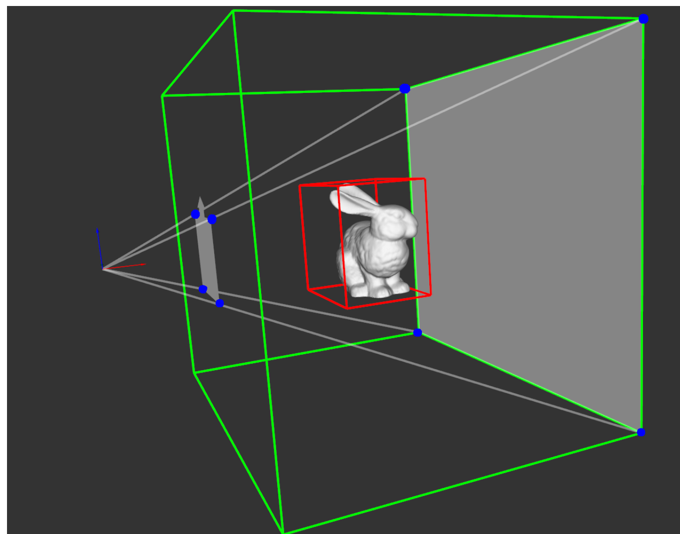


Figura 3.10: Volume envolvente do frustum.

O cálculo de interseção de três planos, necessário para obtenção dos pontos que servem de base para achar a AABB correspondente pode ser encontrado em [21].

Outra forma de descarte mais conservadora que o radar, porém fora dos padrões normais de câmera, foi desenvolvido por Reshetov [58]. Seu trabalho

[2]bit fields - sequência de *bits* que guardam informações booleanas.

introduz a ideia de um algoritmo eficiente para rodar em GPU. Para tal, foi utilizado um modelo de câmera com apenas quatro planos e ao invés de determinar quais os vértices n e p, de todos os volumes envolventes, que serão testados contra os planos em cada *frame*, ele separa as normais dos planos em valores positivos e negativos. Uma vez tendo estes valores separados basta utilizar os valores, já conhecidos, de mínimo e máximo das AABBs para realizar o teste, como pode ser visto na Figura 3.11. Nessa figura os valores $\mathrm{X_n}$, $\mathrm{Y_n}$ e $\mathrm{Z_n}$ são referentes as normais dos planos e $\mathrm{X_{min/max}}$, $\mathrm{Y_{min/max}}$ e $\mathrm{Z_{min/max}}$ aos valores mínimos e máximos da AABB.

max(X_1,0)	max(X_2,0)	max(X_3,0)	max(X_4,0)	* X_{min} +	min(X_1,0)	min(X_2,0)	min(X_3,0)	min(X_4,0)	* X_{max} = X-Plane
max(Y_1,0)	max(Y_2,0)	max(Y_3,0)	max(Y_4,0)	* Y_{min} +	min(Y_1,0)	min(Y_2,0)	min(Y_3,0)	min(Y_4,0)	* Y_{max} = Y-Plane
max(Z_1,0)	max(Z_2,0)	max(Z_3,0)	max(Z_4,0)	* Z_{min} +	min(Z_1,0)	min(Z_2,0)	min(Z_3,0)	min(Z_4,0)	* Z_{max} = Z-Plane

X-Plane + Y-Plane + Z-Plane = F-Plane

Figura 3.11: Teste de descarte com apenas quatro planos.

Este tipo de descarte se mostra bem eficiente, pois a separação das normais dos planos em positivas e negativas é feita apenas uma vez por *frame*, enquanto que no trabalho de Assarsson [3] essa separação é feita em todos os volumes envolventes. Feita a separação, com apenas seis multiplicações e cinco adições em modo SIMD (Seção 3.2.6), é possível determinar a visibilidade através do valor de $F-PLANE$. Caso todos os *bits* mais significativos de cada uma das quatro partes inteiras de $F-PLANE$ forem iguais a zero, então a AABB está fora dos planos. Esse tipo de descarte não foi implementado por sua câmera possuir apenas quatro planos e sua resposta não retornar possíveis interseções do *frustum* com o volume envolvente, e com isso hierarquia não pode ser utilizada.

3.2.5 Percurso sem pilha

O percurso da hierarquia é um algoritmo comum em diversas áreas da computação como *raytracing* e detecção de colisão. Normalmente para visitar os nós da árvore, são utilizadas estruturas auxiliares como pilhas (mais comuns) e filas. Apesar de serem simples, essas estruturas podem introduzir *overhead* na aplicação, dependendo da forma em que são implementadas. Uma

solução para eliminar a necessidade de estruturas auxiliares no percurso foi desenvolvido por [48], porém com a finalidade de tornar a estrutura viável para a realização de *raytracing* em GPU. A mesma ideia foi seguida para implementação em CPU, onde os nós da hierarquia são numerados de acordo com a ordem que eles serão visitados. Caso durante o percurso os nós filhos do corrente não precisarem ser visitados, o próximo nó a ser processado é o que contiver o índice correspondente ao *escape index*. A informação do *escape index* é processada *offline* em cada nó da hierarquia. A Figura 3.12 ilustra uma hierarquia contendo nós com seus respectivos *escape index* indicados pelas setas.

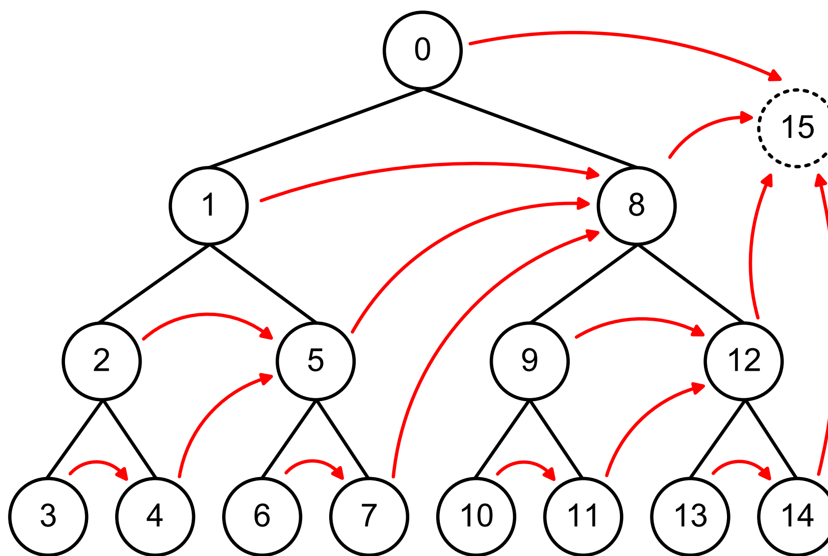


Figura 3.12: Escape index.

[18] definiu três regras para obter o *escape index* em árvores binárias de acordo com a sua posição em relação ao nó pai.

1. O *escape index* do nó filho a esquerda será o nó filho à direita do seu nó pai.

2. O nó filho à direita tem o mesmo *escape index* do seu nó pai.

3. O nó raiz é tratado como um filho à esquerda, e seu *escape index* receberá o valor do último nó da árvore mais uma unidade.

O percurso da hierarquia é feito de forma incremental nos índices dos nós. Caso seja necessário, o *escape index* é utilizado. O índice 15 ilustrado na Figura 3.12 identifica que o percurso terminou. Além de facilitar as operações do percurso na GPU, este método tornou mais simples o algoritmo em CPU, eliminando também o *overhead* introduzido quando a pilha é implementada

dinamicamente. Os detalhes sobre o ganho de desempenho na utilização desta forma de percurso serão levantados no capítulo 4.

3.2.6 SIMD

Flynn [24] classifica os computadores segundo o fluxo de instruções e dados dentro do processador. Dentro dessas classificações podemos citar SISD (***S****ingle* ***I****nstruction* ***S****ingle* ***D****ata*, onde o computador funciona de forma serial com apenas uma unidade de controle), SIMD (***S****ingle* ***I****nstruction,* ***M****ultiple* ***D****ata*, que será discutido com mais detalhes adiante), MIMD (***M****ultiple* ***I****nstruction* ***M****ultiple* ***D****ata*, onde o computador tem vários processadores operando independentemente) e MISD (***M****ultiple* ***I****nstruction* ***S****ingle* ***D****ata*, onde o computador tem vários processadores operando sobre um único fluxo de dados).

SIMD permite paralelizar cálculos agrupando dados em registradores especiais. Em 1999, a Intel incorporou um conjunto de instruções SIMD na série de processadores Pentium III conhecidas como SSE (***S****treaming* ***SIMD*** ***E****xtensions*). Desde então esse número de instruções vem crescendo e se tornando cada vez mais popular. Essas instruções são utilizadas normalmente para acelerar processamento de áudio e vídeo, porém sua utilização vem crescendo em áreas como *raytracing*. Apesar de conseguir realizar cálculos mais rapidamente, as instruções SIMD em CPU, quando comparadas com as placas gráficas, que possuem SIMD nativo, deixam a desejar em termos de desempenho. Mais detalhes sobre a arquitetura SIMD, assim como a sua evolução e ganho de desempenho em várias plataformas podem ser encontrados em [65].

A ideia básica do SIMD é utilizar registradores de 128 *bits* para fazer cálculos em apenas um ciclo de *clock* do processador. Com isso, em máquinas de *32 bits*, podemos agrupar 4 tipos *float* (4 *bytes* = 32 *bits* cada) em apenas um registrador e processar seus cálculos de maneira mais eficiente. A Figura 3.13 ilustra a diferença entre a operação de soma de 4 tipos *float* de 32 *bits* feitas da maneira tradicional e com SIMD.

Os resultados e comentários sobre a inserção do SIMD no algoritmo de *frustum culling* serão discutidos na Seção 4.6.

3.2.7 Multiprocessamento

Esta seção foi motivada pelo trabalho de Assarsson, Stenstrom *et al.* [4] que analisou a utilização de máquinas multiprocessadas para acelerar o

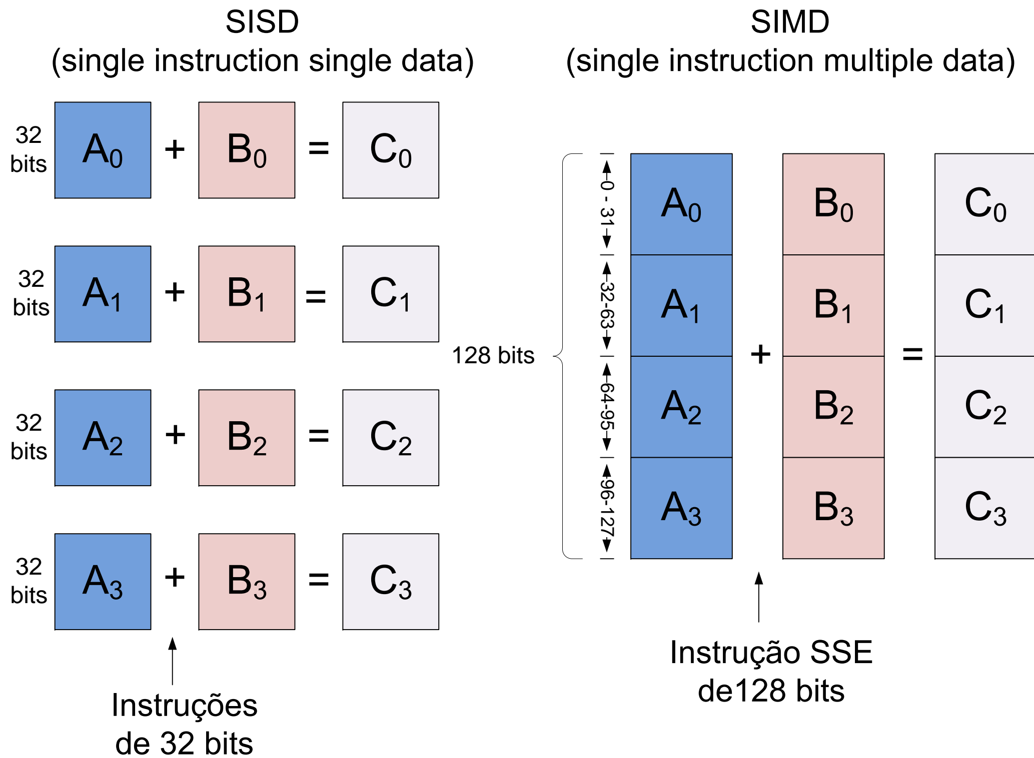


Figura 3.13: SISD X SIMD.

algoritmo de *frustum culling*, principalmente em árvores com elevado número de nós. A grande dificuldade encontrada pelos autores acima foi desenvolver uma maneira de fazer a distribuição de tarefas a serem executadas por cada um dos processadores de forma balanceada, pois a priori não se sabe se as tarefas a serem desenvolvidas por cada um dos processadores vai terminar juntamente com as tarefas dos outros processadores. Outra dificuldade encontrada é que o custo computacional para processar um nó é baixo e a comunicação entre os processadores tem custo alto, sendo assim necessário avaliar se realmente vale a pena distribuir as tarefas.

Assarsson utilizou quatro abordagens para distribuir as tarefas entre os processadores de forma dinâmica.

1. *Global Task Queue* - a distribuição de tarefas é feita através de uma fila global onde todos os processadores têm acesso para inserção e remoção. Foi observado que houve um ganho de performance de 1.5, utilizando três processadores em cenas pequenas, quando comparado com a implementação feita com um processador. O grande problema desta distribuição é o custo de acesso à fila de tarefas.

2. *Global Counter Scheme* - baseada na implementação *Global Task Queue*, esta abordagem acrescenta a cada processador uma fila local e um conta-

dor global, eliminando assim o tempo gasto no acesso à fila global. Desta vez o novo gargalo se tornou o *lock* do contador global para não haver acessos múltiplos. Esta distribuição obteve um ganho de performance de 1.9 em cima do original utilizando mais de oito processadores.

3. *Hybrid Scheme* - esta distribuição utiliza a mesma ideia apresentada no *Global Counter Scheme* acrescido de duas otimizações. A primeira é baseada na utilização de *escape index*, já apresentada anteriormente na Seção 3.2.5, que além de não precisar de estrutura adicional para realizar o percurso, também facilita a distribuição de tarefas que antes eram feitas por nós e agora pode ser feita por intervalo de índices. A segunda otimização não distribui subárvores que contenham um número reduzido de nós, pois o custo de distribuí-los é maior que processá-los no mesmo processador. Essas duas otimizações aumentaram o desempenho do algoritmo de 1.9 para 2.2, quando 10 processadores são utilizados.

4. *Lock-free Scheme* - a ideia deste tipo de distribuição é eliminar a necessidade de *locks* e qualquer tipo de sincronização. Isso foi feito utilizando duas filas em cada processador sendo que uma para processamento interno e outra para receber tarefas de outros processadores. A implementação foi feita utilizando *ring buffers* com dois índices apontando para o inicio e o fim da fila que compartilha tarefas. A ideia básica para não haver necessidade de sincronismo é que apenas um processador possa realizar a inserção ou remoção de tarefas, nunca as duas operações. A utilização deste tipo de distribuição obteve ganho de 4.3 vezes maior (utilizando 13 processadores) quando comparado com o original.

Todos os testes foram feitos em uma máquina Sun Enterprise 4000 *shared-memory multiprocessor* (14 UltraSPARC-II CPUs a 248MHZ) que para a época era um supercomputador e atualmente está totalmente obsoleto. Enquanto uma simples operação de raiz quadrada com um tipo *float* em um Intel Core 2 Duo demora 1.6 nanossegundo, na máquina utilizada por Assarsson demoraria 140 nanossegundos.

A evolução natural dos processadores e a não evolução do trabalho de Assarsson também motivaram a utilização de multiprocessamento neste trabalho. Os sistemas que antes eram chamados de *shared-memory multiprocessor* agora são chamados também de *multicores* (em PCs) contendo até 8 núcleos (Core i7). Os processadores modernos possuem além de melhor desempenho, bibliotecas auxiliares cada vez mais amigáveis para o desenvolvimento de aplicações que tiram proveito do multiprocessamento. O capítulo 4 mostrará os algoritmos desenvolvidos para explorar o algoritmo de *frustum culling* de forma paralela.

3.3 Memória utilizada em CPU

Além dos problemas enfrentados com o desempenho das aplicações que utilizam modelos massivos, outro problema recorrente é a memória utilizada. Mesmo com o grande aumento da quantidade de memória disponível nas máquinas atuais, seu uso deve ser controlado principalmente quando estamos trabalhando com modelos massivos. Muitos desses modelos ocupam muito espaço em disco rígido e alguns não podem ser carregados em memória RAM com espaço de endereçamento de 32 *bits*, muito menos na memória da placa de vídeo. Normalmente esses modelos são visualizados em *clusters* contendo vários processadores e quantidade de memória bem maior que as encontradas nos PCs. Como este trabalho foi desenvolvido em um PC e utilizando modelos massivos (mais detalhes na Seção 3.4), cuidados extras tiveram que ser tomados.

Para que os cálculos de *frustum culling* acelerem a aplicação, é necessário guardar além dos dados do modelo, a maioria das estruturas auxiliares em memória RAM. A utilização da memória no algoritmo de *frustum culling* divide-se em duas partes: uma dependente do modelo e a outra fixa. A memória não dependente do modelo (fixa) refere-se aos parâmetros de câmera necessários para realizar o descarte de objetos. Já a outra parte vai depender do tamanho do modelo e quantidade de objetos nele contidos. A parte fixa, quando comparada com parte dependente do modelo e as suas estruturas auxiliares pode ser desprezada.

Existem dois tipos de modelos que são utilizados neste trabalho: os modelos com malhas poligonais e os modelos com informações paramétricas. Mais detalhes sobre a diferença entre eles podem ser encontradas na Seção 5.2. Os modelos mais comuns, que contêm malha de triângulos, normalmente trazem informações como vértices, normais e cor, que utilizam grande parte da memória. Um dos modelos utilizados neste trabalho, o Boeing 777, contém aproximadamente 350 milhões triângulos, que correspondem a 4.2 GB em memória, e não cabem na memória principal das máquinas de 32 *bits*. Alguns sistemas de visualização utilizam técnicas que carregam os dados necessários em um determinado *frame* por demanda, não dependendo assim que todo o modelo esteja na memória principal. Como este não é o foco deste trabalho, as técnicas utilizadas para minimizar a quantidade memória para o algoritmo de *frustum culling* ficaram para as estruturas auxiliares.

A estrutura que mais utiliza memória é a hierarquia de volumes envolventes. Cada informação contida no nó é destinada a alguma parte do algoritmo. A Figura 3.14 ilustra a quantidade de memória utilizada por dois tipos

de estrutura de um nó.

Estrutura completa

Node*	parent
Node*	leftChild
Node*	rightChild
unsigned int	planeId
unsigned int	planeMask
unsigned int	startVBOIdx
unsigned int	endVBOIdx
unsigned int	height
Box	bbox

Estrutura reduzida

id
escapeId
planeId
planeMask
startVBOIdx
endVBOIdx
bbox

Figura 3.14: Dados utilizados no nó da hierarquia.

A estrutura completa representa um nó onde é utilizada pilha para fazer o percurso da hierarquia. Os ponteiros são utilizados para percorrer a hierarquia e guardar na pilha nós que serão processados posteriormente. Os valores de *planeId* e *planeMask* guardam informações necessárias para implementação de otimizações. Os índices referentes a *startVBOIdx* e *endVBOIdx* são uma variante da hierarquia utilizada por [10], onde apenas as folhas contêm informações da geometria. A informação *height* é utilizada no percurso da hierarquia do *frustum culling* híbrido (mais detalhes no Capítulo 6). A classe *Box* guarda as informações do volume envolvente do nó.

A estrutura reduzida substitui os ponteiros dos nós por duas variáveis utilizadas no percurso sem pilha (explicado na Seção 3.2.5). Outra diferença para a estrutura completa é a eliminação do campo *height* que pode ser determinado em tempo de execução. Com isso houve uma redução de 8 *bytes* em máquinas de 32 *bits* e de 20 *bytes* em máquinas de 64 *bits*.

3.4 Ambiente de benchmark

Para a realização dos testes foi utilizada uma maquina Intel QX9650 3.0ghz Quad Core com 8 GB de memória e uma placa de vídeo GTX280 com 1GB de memória utilizando o sistema operacional Windows XP Professional X64 Edition. A linguagem de programação utilizada foi C++ e *GLSL* [3] para

[3]GLSL (Open**GL** **S**hading **L**anguage) - linguagem de programação de alto nível criada pela OpenGL ARB (**A**rchitecture **R**eview **B**oard) destinada a programação do *pipeline* das placas gráficas.

os *shaders* [4]. Também foram utilizadas bibliotecas auxiliares como OpenGL para a renderização, Qt 4.4.1 [57] para a interface gráfica, libQGLViewer 2.3.1 [56] para auxiliar o desenvolvimento do visualizador e libglsl 1.0.0 [26] para os *shaders*. A Figura 3.15 ilustra a interface da aplicação desenvolvida para os testes.

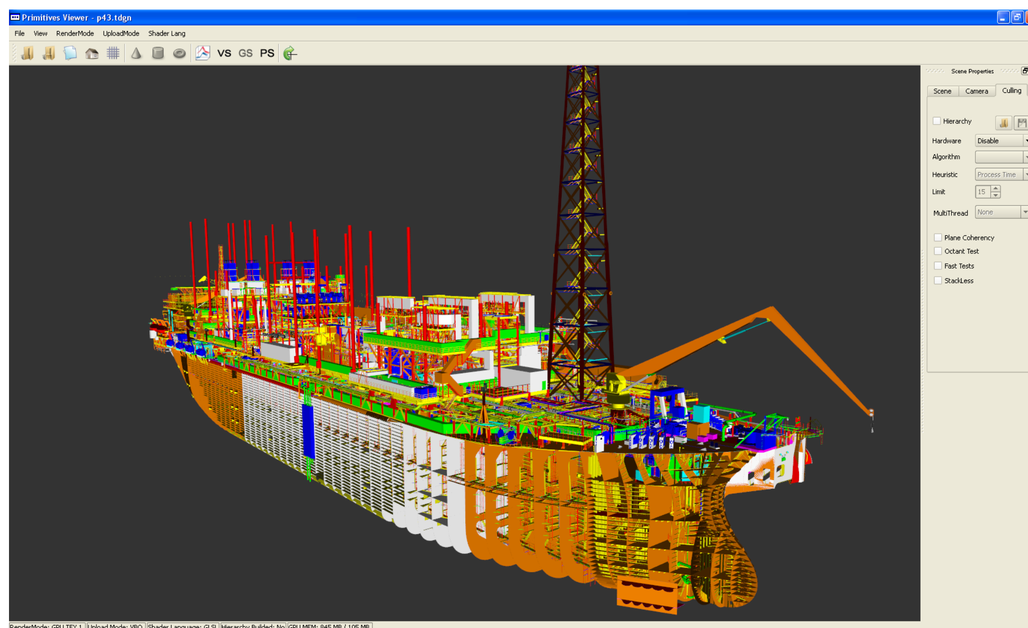


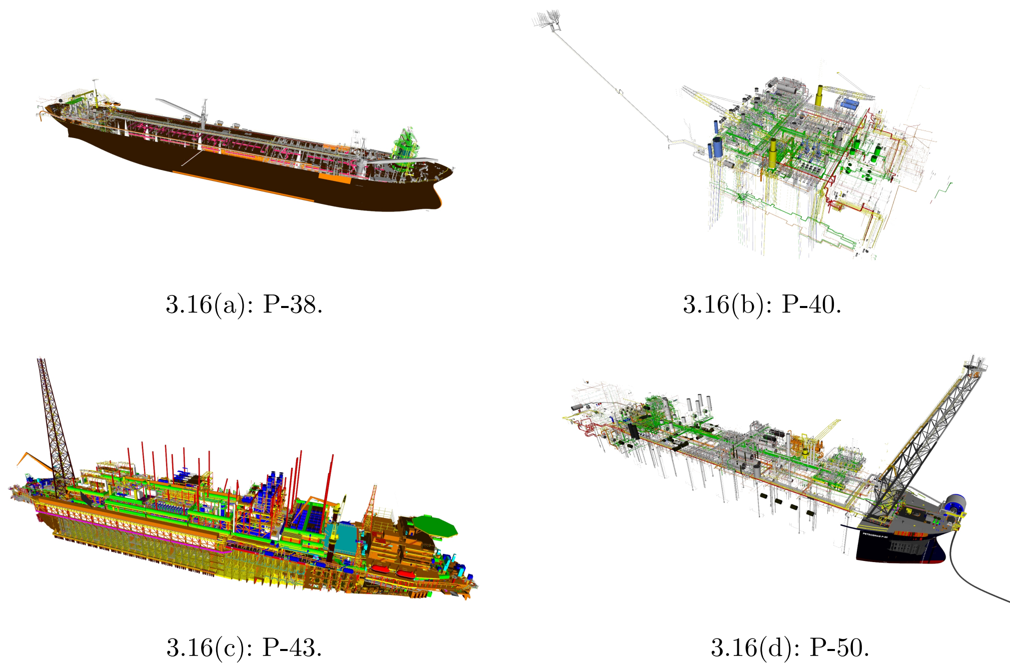
Figura 3.15: Aplicação desenvolvida para os testes.

Os modelos utilizados nos testes se subdividem em dois grupos: modelos com informações paramétricas e modelos com malhas poligonais. Os modelos contendo informações paramétricas têm o formato TDGN. O arquivo TDGN é um formato que faz a ligação entre ferramentas CAD e os módulos de visualização 3D desenvolvidos no Tecgraf [67]. Seus dados são provenientes de partes do arquivo DGN (**D**esi**GN** file) lido na íntegra pelo programa MicroStation [42]. As Figuras 3.16 ilustram parte de algumas plataformas de petróleo utilizadas nos testes.

Os passos para a renderização dos dados paramétricos de forma eficiente podem ser encontrados na Seção 5.2. Além dos objetos paramétricos, este arquivo também contém dados de malha. A Tabela 3.1 traz as principais informações sobre os modelos TDGN utilizados como testes.

O formato utilizado nos modelos com malhas poligonais foi o OBJ [50]. Desenvolvido pela Wavefront Technologies, este formato foi escolhido por ser de simples interpretação e por um dos modelos utilizados neste trabalho, Boeing 777 (Figura 1.1(a)), já estar neste formato. A Tabela 3.2 traz informações dos

[4] shaders - programas desenvolvidos para serem rodados diretamente na placa gráfica.



3.16(a): P-38.

3.16(b): P-40.

3.16(c): P-43.

3.16(d): P-50.

Figura 3.16: Modelos com informações paramétricas.

Modelos	Fig.	# Cilindros	# Cones	# Joelhos
P-38	3.16(a)	81374	3917	0
P-40	3.16(b)	221933	3814	39586
P-43	3.16(c)	280123	13212	0
P-50	3.16(d)	336591	14192	341168

Tabela 3.1: Modelos paramétricos.

modelos com malhas triangulares nas duas primeiras linhas e nas outras as informações adicionais dos modelos paramétricos.

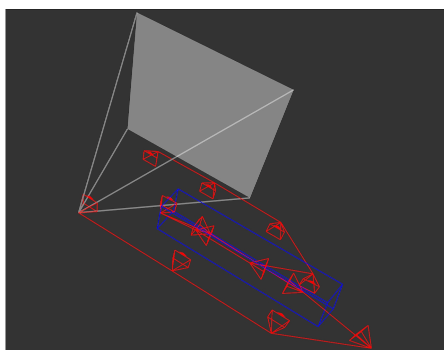
Para fazer os testes dos algoritmos foram construídos caminhos de câmera ao longo dos modelos como pode ser visto na Figura 3.17, onde a caixa azul representa o modelo, a linha vermelha o caminho de câmera feito, a representação de câmera em vermelho ilustra alguns dos *frames* chave para realização da interpolação de câmera e a representação de câmera em branco a posição corrente do observador.

Tais caminhos tentam explorar o maior número de situações possíveis em uma interação, alternando o número de colisões entre o *frustum* e os volumes envolventes. O caminho de câmera feito para o modelo contendo apenas malhas triangulares segue a mesma ideia e pode ser visto na Figura 3.18.

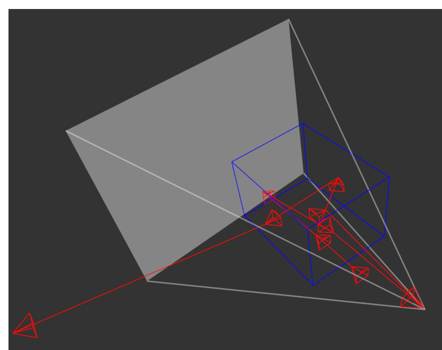
Todos os testes realizados nesta dissertação foram executados três vezes e utilizou-se a média, a fim de obter um valor mais próximo da realidade.

Modelos	# Triângulos	# Malhas	# Objetos	Tamanho(GB)
P-38	37653750	53578	138869	1.94
P-40	28513755	82	265415	1.34
P-43	17462952	699521	280123	1.02
P-50	18477097	21484	713435	1.14
Boeing 777	333730321	712823	712823	14.00

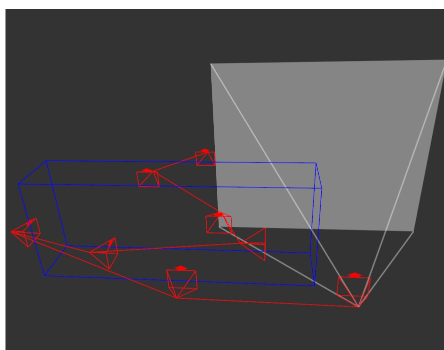
Tabela 3.2: Modelos com malhas triangulares.



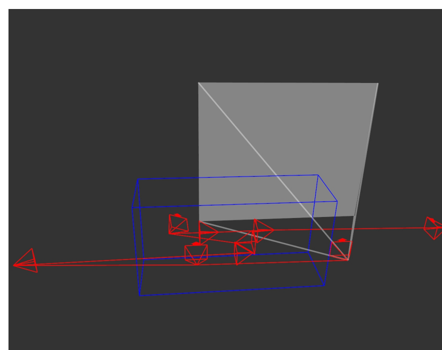
3.17(a): P-38 (80 segundos).



3.17(b): P-40 (87 segundos).



3.17(c): P-43 (72 segundos).



3.17(d): P-50 (92 segundos).

Figura 3.17: Caminhos de câmera pelas plataformas.

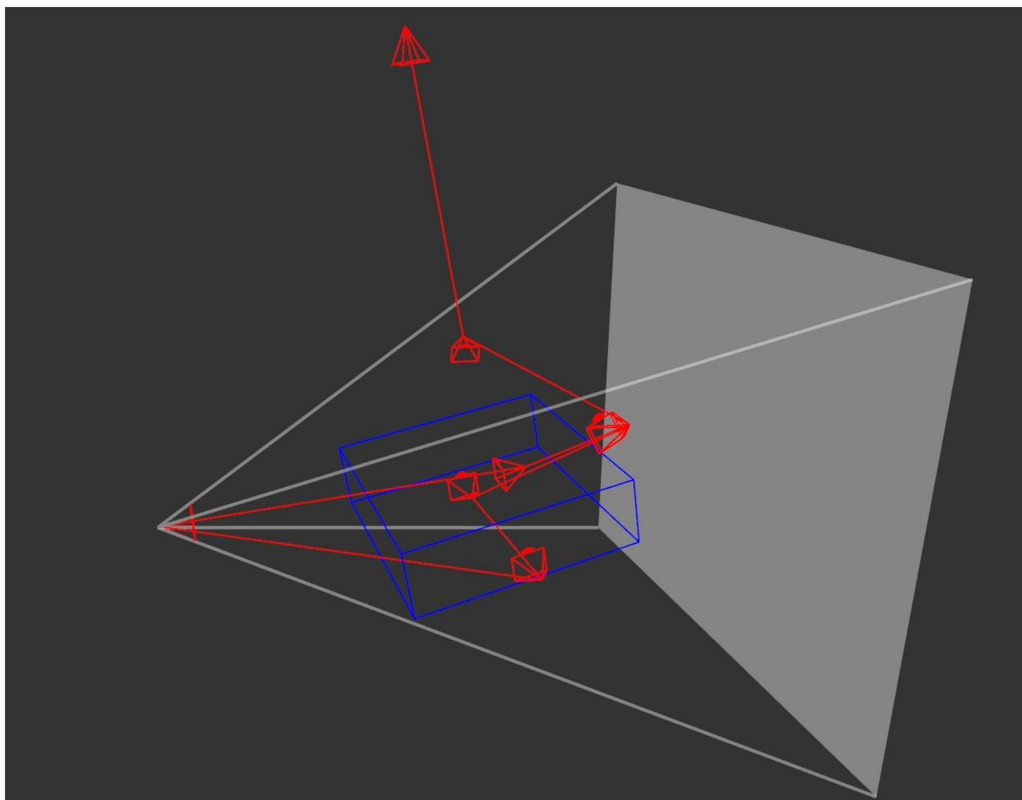


Figura 3.18: Caminho de câmera pelo Boeing (73 segundos).

4
Implementação em CPU

Este capítulo tem por finalidade avaliar os algoritmos e técnicas de aceleração levantadas anteriormente, a fim de obter a melhor combinação entre eles avaliando o desempenho conseguido em cada um dos caminhos de câmera pelos modelos. Ao final do capítulo pretende-se obter o estado da arte do algoritmo de *frustum culling* juntamente com as suas técnicas de aceleração.

4.1
Renderização

A renderização de modelos massivos, mesmo com as placas gráficas mais modernas, ainda é um desafio e necessita de algoritmos especiais para que possam ser viáveis. Segundo [16] as GPUs mais recentes são capazes de processar de 10 a 200 milhões de polígonos por segundo, assumindo que a geometria já esteja na memória da GPU. Mesmo as GPUs mais modernas não possuem espaço necessário para alocar modelos massivos. Mesmo se coubessem, a renderização de um *frame* do Boeing 777, por exemplo, levaria 1,75 segundos, que é bem longe da taxa iterativa de 0,33 segundos (30 *frames* por segundo).

Como o foco deste trabalho é estudar técnicas de *frustum culling*, e não foram implementadas técnicas avançadas para viabilizar a renderização dos modelos massivos em taxas iterativas, na maioria dos testes a renderização foi desligada, a fim de facilitar a localização dos gargalos. Assim, durante o caminho de câmera apenas o algoritmo de *frustum culling* é realizado. Os testes de *frustum culling* com as GPU *primitives* foram os únicos que tiveram a renderização ligada.

4.2
Radar X Planos

A Seção 3.2.2 levantou o descarte de volumes envolventes, diferente da abordagem tradicional por planos, conhecido como radar. O trabalho [66] faz a comparação entre as duas abordagens e mostra o ganho de performance do radar. Porém não são descritos detalhes de implementação, como a utilização de hierarquia e os cálculos de descarte das AABBs.

A primeira análise entre essas duas técnicas é referente à atualização dos dados da câmera. A abordagem clássica precisa extrair a equação dos planos do *frustum* de visão, como foi visto anteriormente, o que demanda uma série de cálculos. Em contrapartida, a abordagem do radar só precisa calcular os limites da câmera uma vez no início da aplicação, ou quando seus parâmetros são modificados, e depois é suficiente atualizar seus vetores (*up*, *right* e *forward*) quando a câmera muda sua posição ou orientação. Esta atualização quando comparada com a extração de planos mostra-se bastante eficiente. A segunda análise refere-se aos cálculos de descarte. A abordagem do radar realiza menos operações que a abordagem clássica como pode ser visto na Tabela 4.1, onde são expostos os números de operações necessárias para atualização e descarte de volumes envolventes no melhor e pior caso. Estes números foram conseguidos contabilizando cada uma das instruções do código gerado, atribuindo pesos de acordo com a instrução executada [71].

Método	# Atualizações	Melhor caso	Pior caso
Planos	316 operações	7 operações	336 operações
Radar	112 operações	8 operações	192 operações
Planos+Otimização	316 operações	10 operações	120 operações

Tabela 4.1: Radar X Planos.

Analisando os algoritmos de planos e radar, a escolha mais correta seria adotar o radar como representante do algoritmo de *frustum culling*, porém quando é incorporado ao algoritmo de planos a otimização onde apenas 2 vértices são testados contra os planos [30, 27] (última linha da Tabela 4.1), seu desempenho aumenta tendo um melhor caso médio. A atualização dos atributos dos algoritmos, onde o radar levou ampla vantagem, quando comparado com os cálculos de descarte dos volumes envolventes, consomem uma parte ínfima do processamento e não precisa ser levado em conta. A diferença de performance dos três algoritmos citados fica clara nos testes realizados com os caminhos de câmera nas plataformas como pode ser visto na Figura 4.1.

O algoritmo de planos juntamente com a otimização obteve melhor desempenho em todo o caminho de câmera das plataformas de petróleo. Tendo melhor desempenho, o caminho de câmera é feito mais rápido e com isso em alguns casos o gráfico termina antes que os outros como por exemplo na P-43. O comportamento de simetria nos gráficos dos algoritmo de radar e planos com otimização pode ser explicado pelo fato de que o ponto forte de um algoritmo é o ponto fraco do outro. Enquanto o algoritmo de planos com otimização consegue detectar que um volume envolvente está fora do *frustum* de maneira rápida, o algoritmo de radar precisa realizar mais cálculos para chegar a este

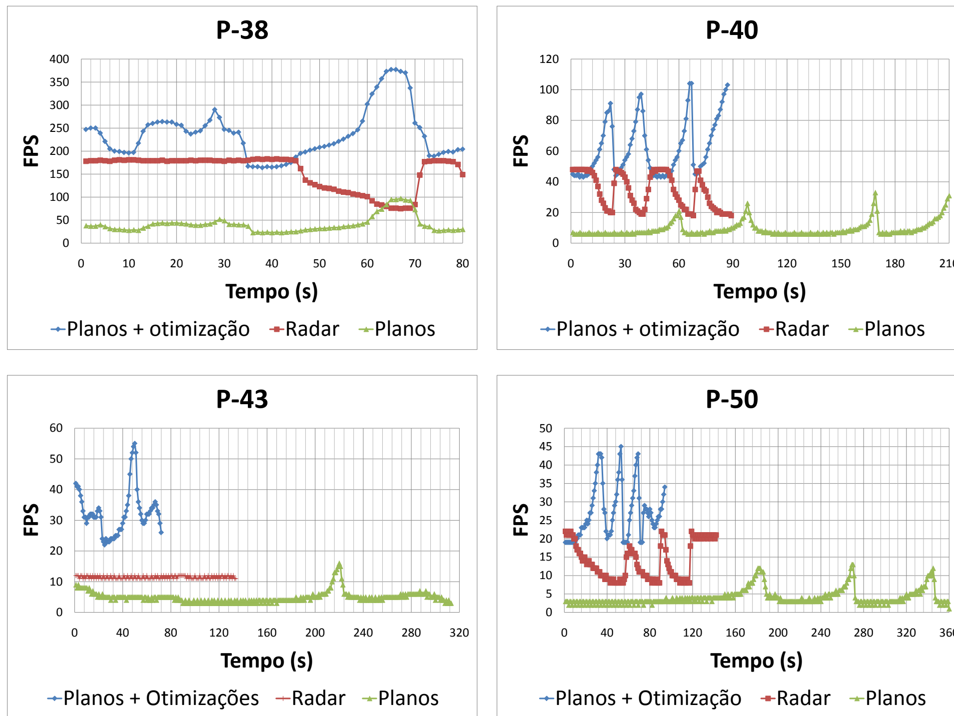


Figura 4.1: Caminhos de câmera sem hierarquia nas plataformas.

resultado. No caso do Boeing, o radar superou o algoritmo de planos com otimização como pode ser visto na Figura 4.2.

A execução dos três algoritmos, nesses testes, não utilizou hierarquia, e com isso não houve a necessidade de retornar o resultado de interseção dos volumes com o *frustum*. Quando há a necessidade de retornar possíveis interseções, quando a hierarquia é utilizada, o algoritmo de radar implementado perde performance e foi superado no único caminho de câmera que ainda faltava, como pode ser visto na Tabela 4.2.

Método	Frames Processados	FPS Mínimo	FPS Máximo	Média do Caminho
Radar sem interseção	2674	34	37	36.217
Radar com interseção	1553	15	17	16.096
Planos + Otimização	2522	32	39	34.171

Tabela 4.2: Radar sem interseção X Radar com interseção para o Boeing.

Como a utilização de hierarquia é primordial para o desempenho do algoritmo, a abordagem que utiliza planos com a otimização de testar apenas dois vértices da AABB foi escolhida como algoritmo de descarte de volumes envolventes.

Neste momento o gargalo da aplicação encontra-se no descarte dos volumes devido à grande quantidade de cálculos executados. As próximas seções

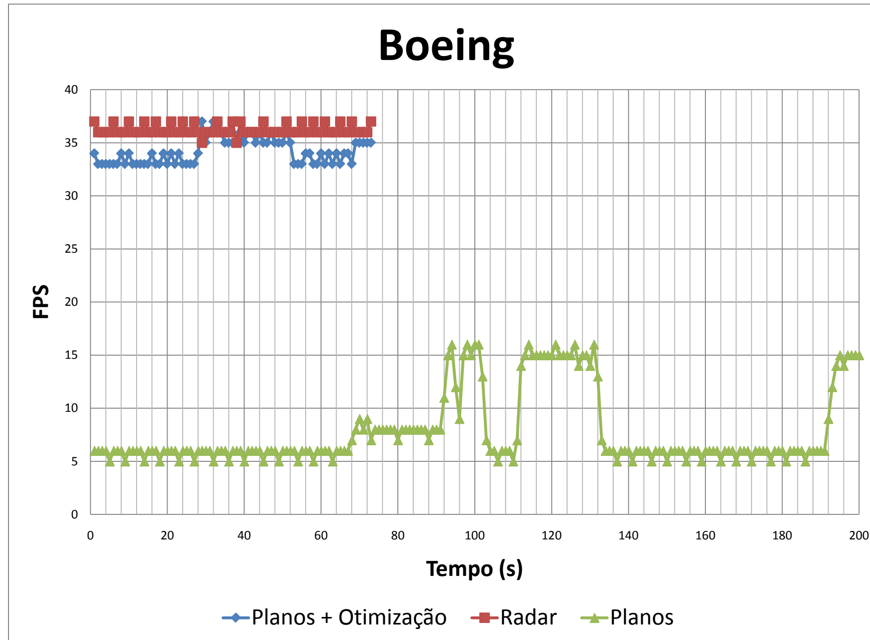


Figura 4.2: Caminhos de câmera sem hierarquia no Boeing.

tentam incorporar as otimizações descritas anteriormente para minimizar os efeitos da inserção do algoritmo de *frustum culling* na aplicação.

4.3 Hierarquia

Na Seção 3.2.1 foram levantados vários tipos de volumes envolventes, onde, à medida que se ajustam melhor com os objetos, demandam maior processamento na execução dos algoritmos. Também foi decidido que a AABB seria o volume envolvente utilizado neste trabalho. Para acelerar os cálculos do *frustum culling* foi introduzida a ideia de hierarquia de volumes envolventes. Ficou para ser decidido qual seria a melhor forma de construção da hierarquia e qual a estrutura de particionamento do espaço seria utilizada. Dada a diversidade e complexidade das cenas, não existe heurística ideal para todos os modelos.

Foram construídas hierarquias do tipo *top-down* com o particionamento do espaço, utilizando a média e a mediana dos centróides das caixas envolventes. Todos os modelos foram submetidos aos diferentes tipos de construção de hierarquia a fim de analisar qual delas proporciona a melhor árvore. Neste caso, a melhor árvore é a que resultar em melhor *fps* nos seus respectivos caminhos de câmera. A Tabela 4.3 ilustra os resultados obtidos com a construção de hierarquia utilizando mediana.

A construção da hierarquia utilizando mediana obteve árvores de al-

Modelos	# Nós	# Folhas	Altura	Tempo de construção(s)
P-38	174533	87267	20	3.29
P-40	559303	279652	22	3.64
P-43	1104911	552456	23	7.70
P-50	1119449	559725	23	8.92
Boeing 777	1383049	691525	21	7.37

Tabela 4.3: Hierarquias utilizando mediana.

tura menor quando comparadas com as hierarquias obtidas utilizando média (Tabela 4.4). A construção utilizando mediana demorou mais tempo em virtude da necessidade de calcular a mediana dos pontos ao longo do maior eixo, o que envolveu um algoritmo de ordenação.

Modelos	# Nós	# Folhas	Altura	Tempo de construção(s)
P-38	225833	112917	24	2.66
P-40	756209	378105	26	2.81
P-43	1388287	694144	27	5.65
P-50	1541347	770674	28	6.84
Boeing 777	1390645	695323	29	4.85

Tabela 4.4: Hierarquias utilizando média.

A construção das hierarquias adotou como único critério de parada, a existência de três objetos de cada tipo. Dessa forma um nó folha terá no máximo três identificadores de geometria de cada tipo. Esta restrição foi imposta a fim de obter árvores com maior altura e consequentemente forçar mais cálculos em determinados *frames*, o que ajuda na determinação do algoritmo de melhor eficiência. As Figuras 4.3 e 4.4 mostram os caminhos de câmera feitos com o algoritmo de planos e a otimização de testes de apenas dois vértices nos dois tipos de hierarquias construídas.

Os caminhos de câmera para as plataformas P-38 e P-50 e o Boeing (Figura 4.3) tiveram melhor performance nas hierarquias construídas utilizando média dos centróides.

A Tabela 4.5 mostra com mais detalhes as performances dos caminhos de câmera ao longo das hierarquias construídas para a P-38 e P-50 e o Boeing.

Os caminhos de câmera nas plataformas P-40 e P43 (Figura 4.4) obtiveram melhor performance utilizando hierarquias construídas a partir da mediana dos centróides dos volumes envolventes. A Tabela 4.6 mostra com mais detalhes as suas performances.

Apesar da diferença entre a performance dos dois tipos de hierarquia ser pequena, ela foi levada em conta e os que obtiveram melhor desempenho foram

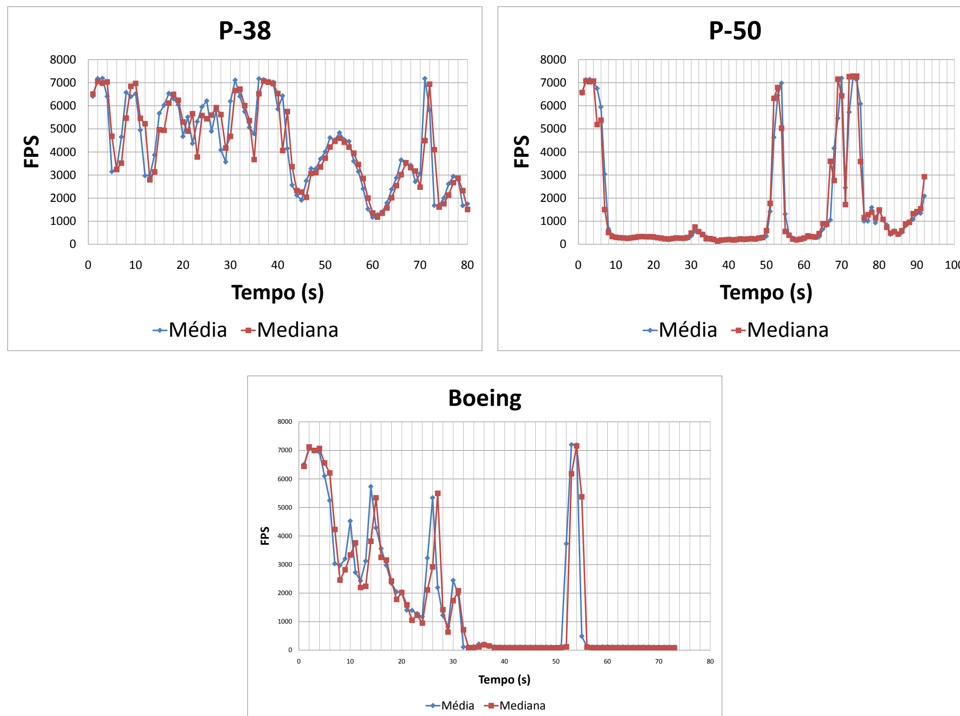


Figura 4.3: Caminhos de câmera com hierarquias onde média foi melhor.

utilizados em conjunto com os demais algoritmos.

Uma forma de otimizar o tempo de processamento do algoritmo de *frustum culling* é interromper o percurso da hierarquia a partir de certa altura da árvore. Esse tipo de otimização não foi feita pois a ideia deste trabalho é buscar o algoritmo que obtenha melhor desempenho, porém retornando o mínimo de geometrias não visíveis.

4.4 Percurso

O percurso da hierarquia sem a utilização de pilha trouxe dois grandes benefícios à aplicação. A primeira foi a redução do uso de memória, de suma importância em modelos massivos. No pior caso, a pilha teria o tamanho da altura da árvore. O uso de memória ainda pode ser maior quando são utilizados vários processadores no percurso, como será visto na Seção 4.7.

A outra vantagem obtida com a utilização do percurso sem pilha é o ganho de performance. A utilização de pilha implica em três possibilidades de alocação de memória. A primeira é alocar estaticamente o máximo de memória que pode ser utilizada no percurso, o que agravaria mais ainda o problema diagnosticado anteriormente, e descobrir o mínimo de memória que será utilizado previamente é uma tarefa difícil. A segunda seria fazer uma alocação dinâmica de memória, o que implica em introduzir um *overhead* no

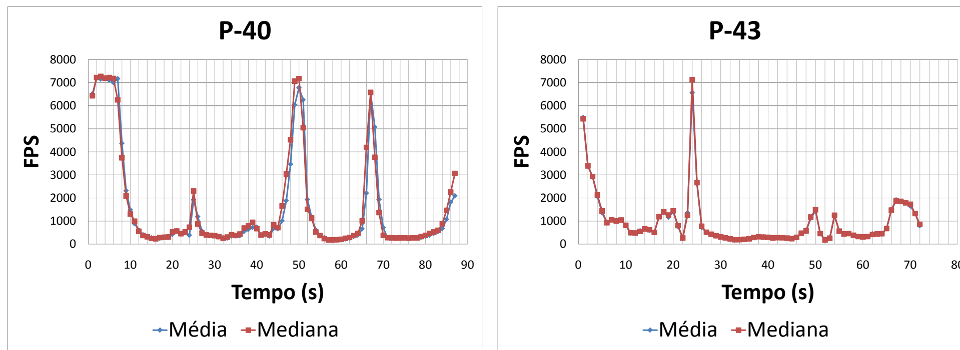


Figura 4.4: Caminho de câmera com hierarquias onde mediana foi melhor.

Método	Frames Processados	FPS Mínimo	FPS Máximo	Média do Caminho
Mediana na P-38	347447	**1168**	7101	4296.6
Média na P-38	**348543**	1152	**7200**	**4345.7**
Mediana na P-50	144655	**142**	**7303**	1570.5
Média na P-50	**146609**	137	7274	**1584.4**
Mediana no Boeing	127491	84	**7235**	1727.7
Média no Boeing	**129179**	**103**	7215	**1766.8**

Tabela 4.5: Detalhes dos caminhos com hierarquias de média e mediana.

Método	Frames Processados	FPS Mínimo	FPS Máximo	Média do Caminho
Média na P-40	138886	169	7232	1580.7
Mediana na P-40	**142160**	**179**	**7284**	**1625.3**
Média na P-43	68456	**182**	6577	943.0
Mediana na P-43	**69493**	178	**7153**	**957.2**

Tabela 4.6: Detalhes dos caminhos com hierarquias de média e mediana.

algoritmo. A terceira utilizaria recursão e não foi implementada. Os ganhos em termos de desempenho na remoção do uso de pilha podem ser observados nas Figuras 4.5 e 4.6.

Em todos os testes a remoção da necessidade de pilha no percurso da hierarquia obteve ganhos que variaram de 1.5% no caso da plataforma P-50 até 11.69% no caso do Boeing. Os testes foram feitos utilizando o algoritmo de planos com a otimização de testes de apenas dois vértices e as hierárquias que obtiveram melhor performance em cada modelo.

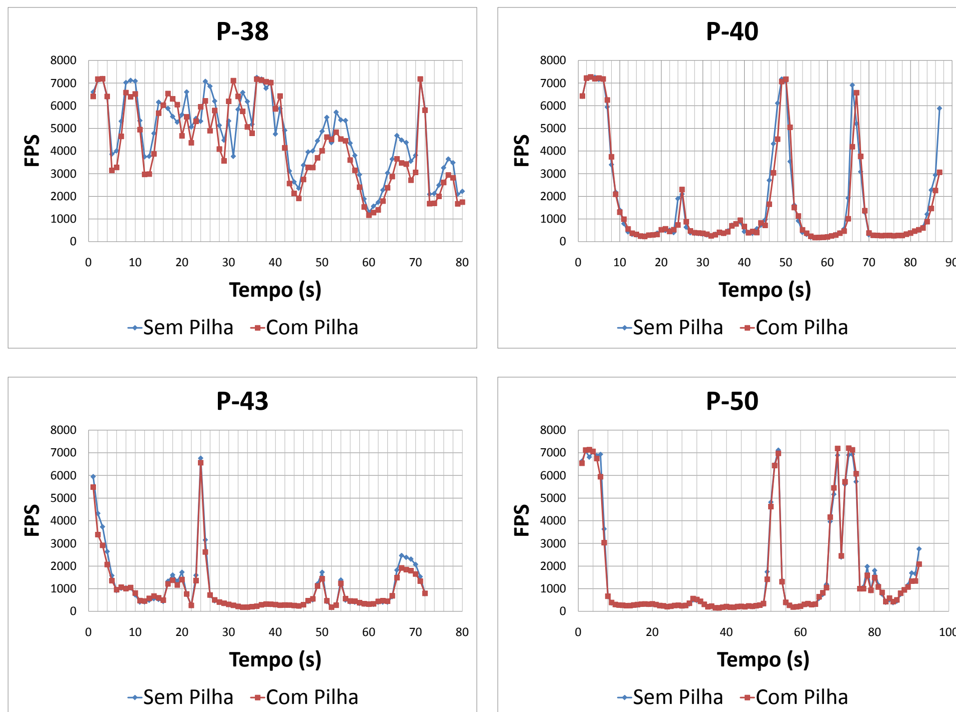


Figura 4.5: Caminho de câmera sem pilha nas plataformas.

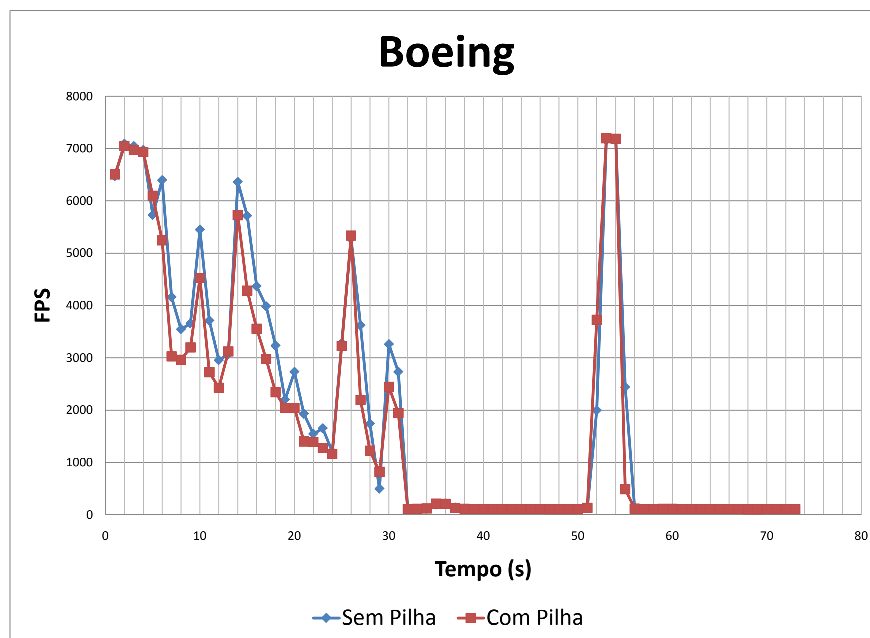


Figura 4.6: Caminho de câmera sem pilha no Boeing.

4.5 Otimizações

A primeira otimização a ser inserida no *pipeline* do *frustum culling* foi a remoção da necessidade de testar todos os oito vértices da AABB contra os planos. Como foi visto na Seção 4.2 esta otimização aumentou o desempenho da aplicação em duas vezes quando comparado ao algoritmo de radar e em quatro vezes em relação a implementação de planos sem otimização.

A conclusão do trabalho de Assarsson e Möller *et al.* [3], citados anteriormente, identificou que a combinação entre as otimizações *plane-coherency test* e *octant test* obtiveram a melhor performance, como pode ser visto na Tabela 4.7.

Path	1	1	1	2	2	2	3	3	3	4	4	4
Model	1	2	3	1	2	3	1	2	3	1	2	3
Only Basic intersection test	2.8	1.9	3.9	2.2	2.0	3.1	3.9	2.5	4.3	3.1	2.2	3.7
Plane-coherency + octant test	**4.0**	**2.4**	**5.1**	**2.8**	**2.6**	**3.9**	4.8	**3.5**	**5.6**	3.3	3.0	5.1
Plane-coherency + TR coherency	3.8	2.0	4.0	2.5	2.2	3.0	5.0	2.8	4.4	**8.3**	3.1	**11.0**
Plane-coherency + octant test + TR coherency	3.7	2.2	4.5	2.6	2.4	3.6	**5.1**	3.0	4.8	8.0	**3.3**	9.0

Tabela 4.7: Resultados obtidos no trabalho de Assarsson.

A Tabela 4.7 mostra os resultados obtidos por Assarsson fazendo diferentes combinações de otimizações em três modelos de tamanhos variados contendo quatro camanhos de câmera diferentes. Segundo Assarsson, a otimização *masking* não se mostrou competitiva com as outras. Seus valores estão indicando o quanto os algoritmos foram mais eficientes quando comparados com o caminho de câmera sem teste de interseção. A inserção desses dois algoritmos no *pipeline* do *frustum culling*, neste trabalho, obteve os resultados expressos na Figura 4.7 para as plataformas e na Figura 4.8 para o Boeing.

A utilização das duas otimizações aumentou a performance da aplicação em todos os casos de testes quando comparada com o percurso da hierarquia no caminho de câmera sem nenhuma otimização.

A última otimização citada na Seção 3.2.4, convertia o *frustum* de visão em uma AABB e realizava testes entre duas AABB para determinar se o objeto estaria totalmente fora do *frustum* de forma mais rápida do que o teste convencional. Essa otimização eleva o tempo de processamento da atualização da câmera, pois é necessário determinar a AABB da câmera, porém diminui o esforço gasto no descarte dos volumes envolventes em algumas situações. A determinação da AABB da câmera quando comparado com os testes de descarte pode ser ignorado em termos de tempo de processamento. Apenas os

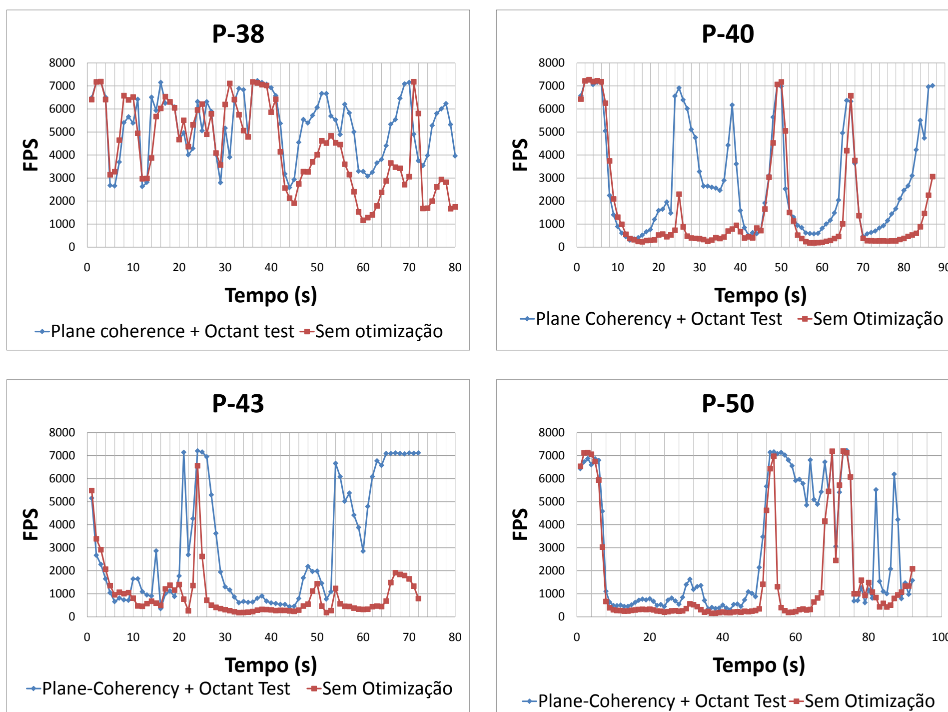


Figura 4.7: Otimizações sugeridas por Assarsoon nas plataformas.

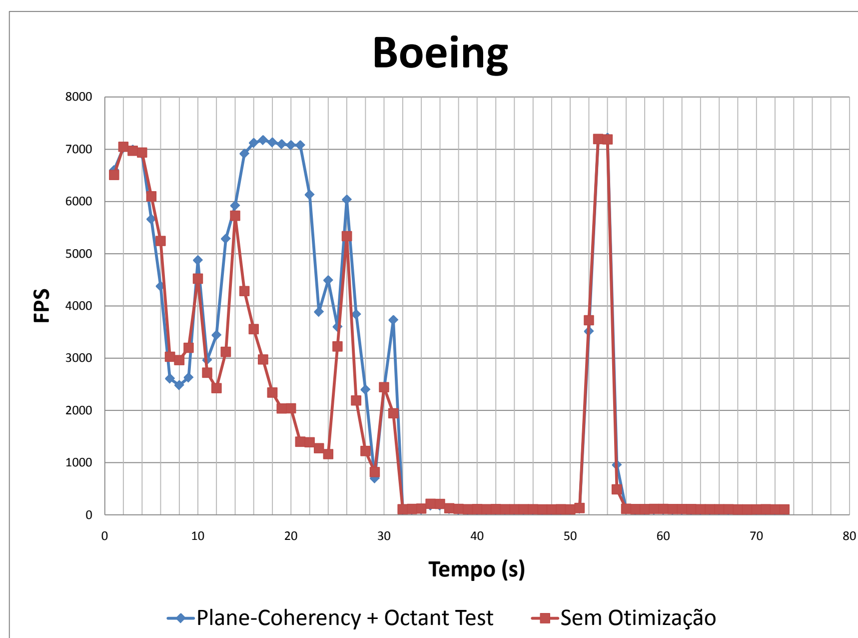


Figura 4.8: Otimizações sugeridas por Assarsoon no Boeing.

testes com as plataformas P-38 e P-50 aumentaram os seus desempenhos com a utilização do teste entre AABBs em 1.58% e 1.28% respectivamente.

As Figuras 4.9 e 4.10 mostram as performances obtidas nos caminhos de câmera com todas e sem nenhum tipo de otimização nas plataformas e no Boeing. Os testes sem otimização foram feitos utilizando apenas hierarquia e testando dois vértices da AABB contra o *frustum*, enquanto os testes com otimização diferem apenas na utilização ou não do teste entre a AABB da câmera e o volume envolvente da geometria e o tipo de hierarquia construída, como pode ser visto na Tabela 4.8.

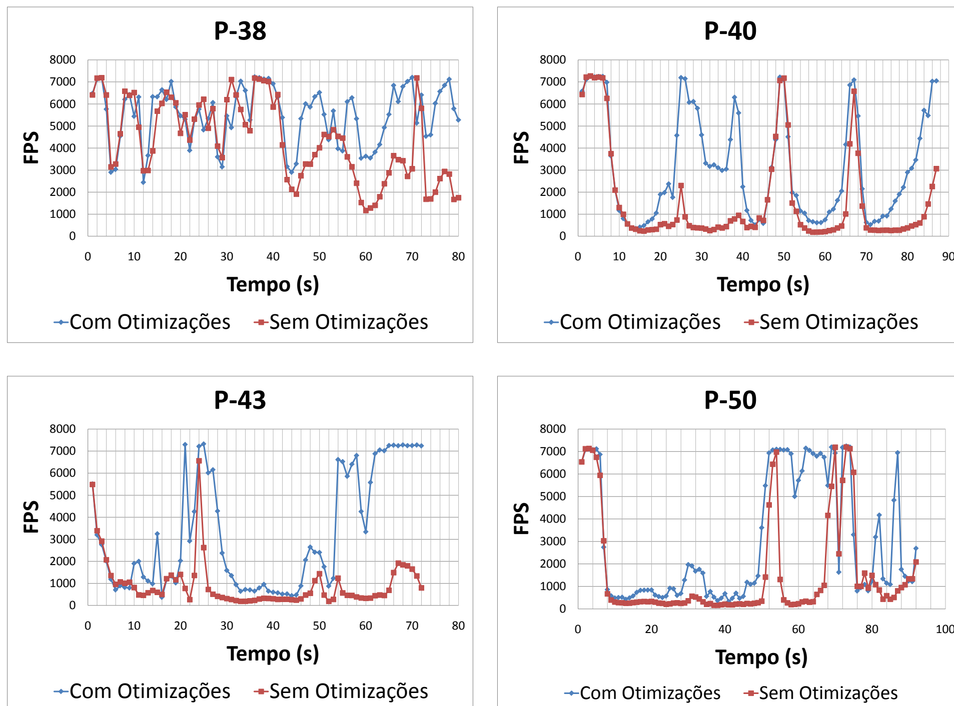


Figura 4.9: Caminhos de câmera pelas plataformas com otimizações.

Em poucos *frames* as otimizações não obtiveram performance superior ao teste de *frustum culling* sem otimização. As causas desse problema podem ser explicadas pela captura destes *frames* em momentos diferentes ou a influência de fatores externos como escalonamento de processos da máquina.

4.6 SIMD

Foram identificados três lugares para a utilização de SIMD no algoritmo de *frustum culling*, sendo cada um deles referente a uma otimização. O primeiro local onde o SIMD pode ser utilizado é na extração dos planos do *frustum*, na abordagem vista na Seção 3.1. Esses cálculos envolvem normalizações que

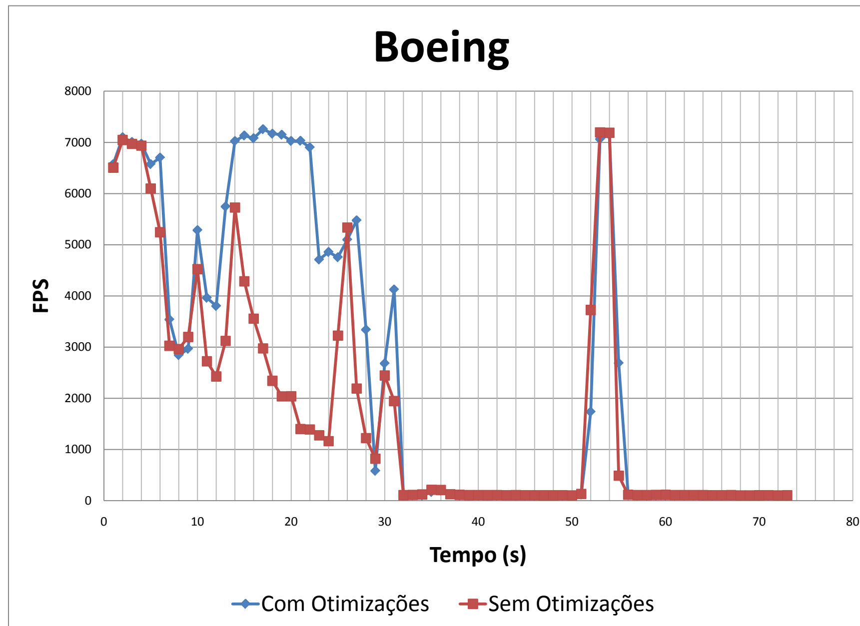


Figura 4.10: Caminhos de câmera pelo Boeing com otimizações.

Modelo	Hierarquia	Planos n e p	Plane-coherency	Octant-test	Ausência de pilha	Teste entre AABBs
P-38	Média	Sim	Sim	Sim	Sim	Sim
P-40	Mediana	Sim	Sim	Sim	Sim	Não
P-43	Mediana	Sim	Sim	Sim	Sim	Não
P-50	Média	Sim	Sim	Sim	Sim	Sim
Boeing	Média	Sim	Sim	Sim	Sim	Não

Tabela 4.8: Melhor combinação de otimizações utilizadas em cada modelo.

podem ser tratadas de maneira paralela. Apesar do gargalo da aplicação não estar nessa parte do código, quando se utiliza modelos massivos, essa técnica foi incorporada ao *pipeline* para ser aproveitada em modelos menores que os tratados nesta dissertação.

Outro local onde é possível tirar proveito do SIMD é nos cálculos de descarte de volumes envolventes, mais especificamente no cálculo de distância. A utilização de SIMD nesta parte do algoritmo não foi de grande valia, uma vez que é necessário construir um novo tipo de 128 *bits* no mínimo para cada teste, o que é mais custoso que fazer o teste diretamente. A Tabela 4.9 compara os caminhos de câmera com e sem SIMD na plataforma P-43.

A construção de no mínimo um novo tipo de 128 *bits* a cada teste feito diminuiu a performance do caminho de câmera da P-43 em 51.66%.

O SIMD também foi utilizado no cálculo da interseção dos planos para construção da AABB do *frustum* que é utilizada nos testes feitos entre AABBs. Essa parte do código segue a mesma ideia da atualização da câmera, pois é

Método	Frames Processados	FPS Mínimo	FPS Máximo	Média do Caminho
Sem SIMD	2234	21	70	37.233
Com SIMD	1473	15	47	24.55

Tabela 4.9: SIMD nos cálculos de descarte da P-43.

feito apenas uma vez a cada *frame* e seu cálculo envolve várias operações.

A conclusão é que a utilização do SIMD só é válida quando há cálculos mais pesados envolvidos que o de descarte de volumes envolventes. Sendo assim, só foram incorporados SIMD na atualização da câmera e no cálculo de sua AABB, porém sem trazer ganho significante para o algoritmo de *frustum culling* nos modelos em questão.

4.7 Multiprocessamento

Como sistemas multiprocessados estão ficando cada vez mais populares, muitas aplicações têm sido desenvolvidas em cima desses sistemas. Um sistema multiprocessado possui dois ou mais processadores físicos, onde cada um é capaz de executar processos simultâneos automaticamente compartilhando um único espaço de memória. O desenvolvimento da aplicação *multithread* foi feito com a biblioteca OpenMP [52], que é uma API para o desenvolvimento de aplicações *multithread* em C, C++ e Fortran disponível em vários sistemas operacionais.

A ideia da utilização de *multithread* tenta reduzir o gargalo da aplicação nos cálculos de descarte. O grande problema enfrentado por Assarsson e Stenstrom *et al.* [4] foi encontrar o melhor balanceamento das tarefas entre os processadores. Assarsson utilizou quatro abordagens diferentes a fim de encontrar o melhor balanceamento fazendo o teste em três tipos de cenas. Mesmo tendo poder de processamento e cenas muito menores do que as utilizadas neste trabalho, foram tomadas como base as duas maiores cenas onde a abordagem chamada de *Lock-free Scheme* obteve melhor desempenho, como pode ser visto na Figura 4.11.

Na cena de tamanho médio em alguns momentos a distribuição *Hybrid Scheme* obtém melhor desempenho que o *Lock-free Scheme*, o que não ocorre na cena maior, onde sempre que o número de processadores é aumentado, o desempenho melhora a uma velocidade maior que as outras distribuições. A distribuição de tarefas *Lock-free Scheme*, desenvolvida por Assarsson, foi implementada com o auxílio da biblioteca OpenMP versão 2.5 (2005). Além dessa abordagem, foram desenvolvidas mais duas para servirem como base de comparação. A primeira não compartilha tarefas entre os processadores, sendo

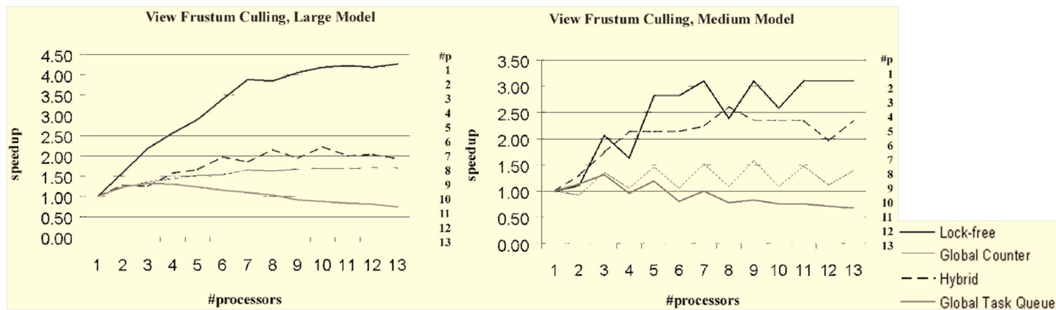
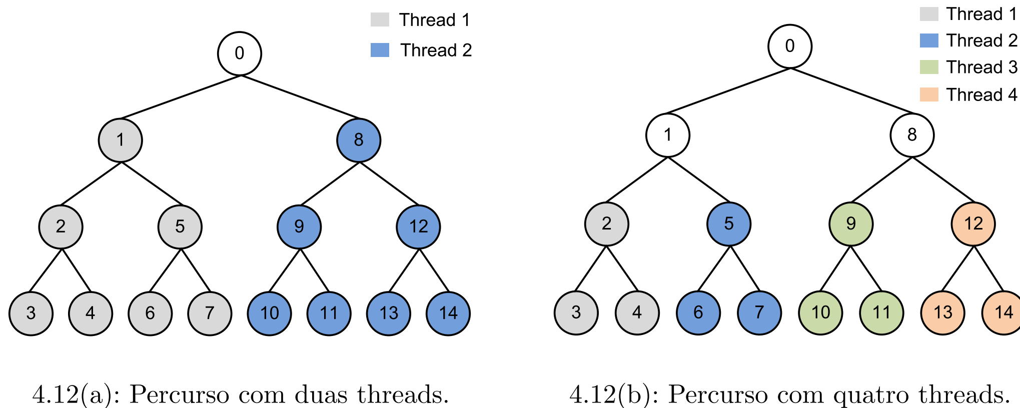


Figura 4.11: Resultados obtidos por Assarsson em duas cenas.

assim, quando as tarefas acabam para um processador ele fica aguardando a tarefa dos outros terminar. A Figura 4.12 ilustra a distribuição de tarefas para dois 4.12(a) e quatro processadores 4.12(b), onde os nós de mesma cor irão ser processados pelas *threads* correspondentes à sua cor.



4.12(a): Percurso com duas threads.

4.12(b): Percurso com quatro threads.

Figura 4.12: Percurso com *multithread*.

A segunda abordagem tenta explorar a versão 3.0 (2008) da biblioteca OpenMP. Esta versão introduziu a ideia de *tasks*. *Tasks* são tarefas desenvolvidas dentro das *threads* que podem ser suspensas ou interrompidas. A grande vantagem da utilização de *tasks* ocorre quando dentro de uma *thread* todos as suas tarefas terminam. Neste ponto suas *tasks* podem ser compartilhadas com as outras *threads*, o que não era possível na versão 2.5 da biblioteca OpenMP. No caso do percurso de hierarquia com *multithread* utilizando *tasks*, a distribuição de tarefas fica implícita não havendo necessidade de troca de dados entre *threads*.

As Figuras 4.13 e 4.14 ilustram a comparação de desempenho obtido com o percurso da hierarquia utilizando uma, duas e quatro *threads* nas plataformas e no Boeing.

Nos testes realizados, apenas na plataforma P-50 com 2 *threads* houve ganho no percurso da hierarquia sem o compartilhamento de tarefas entre *threads*. Em todos os outros modelos a utilização de apenas uma *thread* no

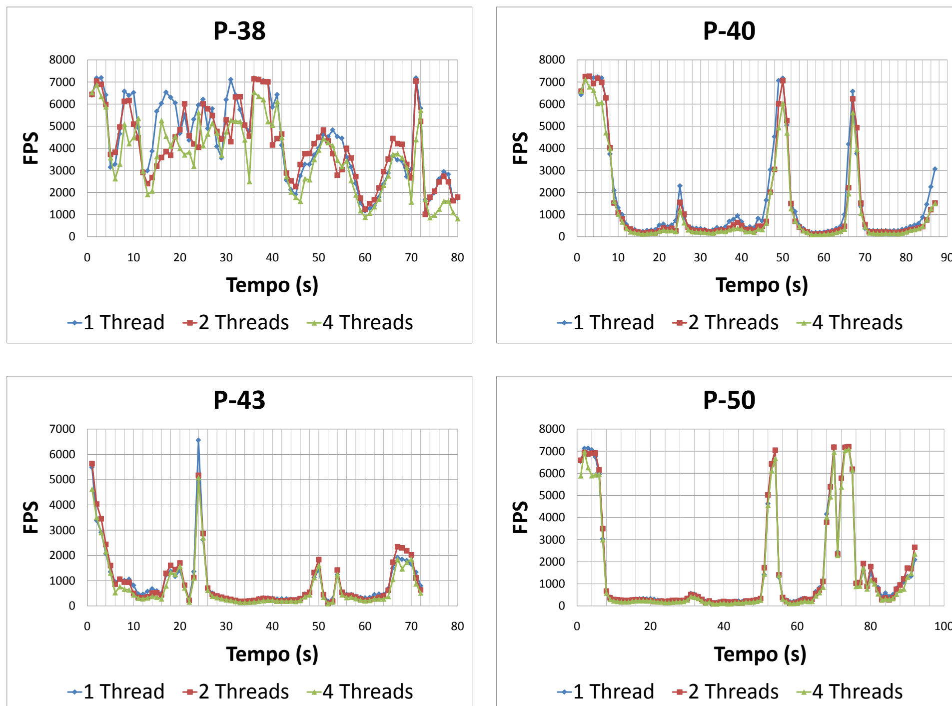


Figura 4.13: Multiprocessamento sem troca de tarefas entre threads.

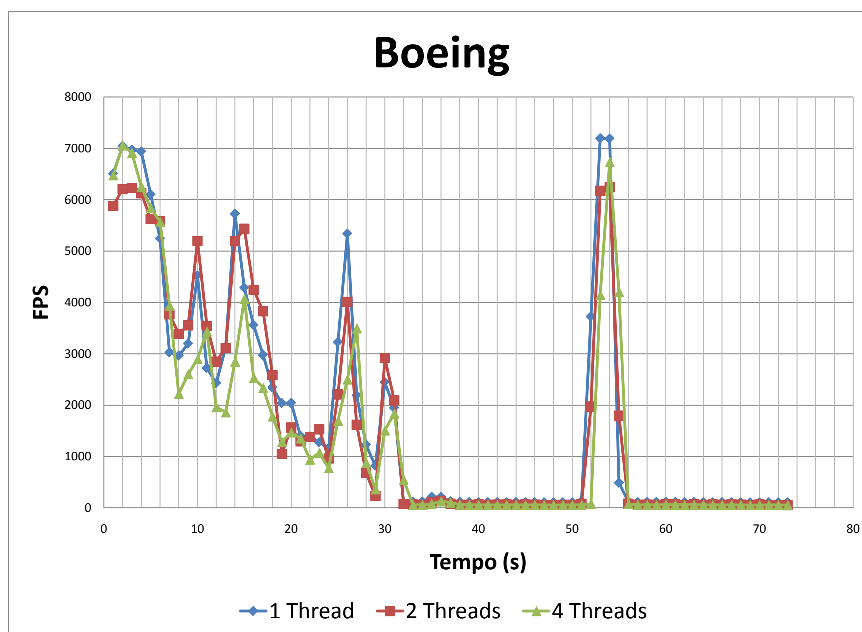


Figura 4.14: Multiprocessamento sem troca de tarefas entre threads no Boeing.

percurso obteve melhor performance. Quanto mais *threads* foram colocadas para realizar o percurso, menor o desempenho.

As Figuras 4.15 e 4.16 ilustram a performance dos algoritmos que trocam tarefas entre as *threads*. Os dois algoritmos implementados, *Lock-Free Scheme* e *Tasks*, utilizaram apenas duas *threads*.

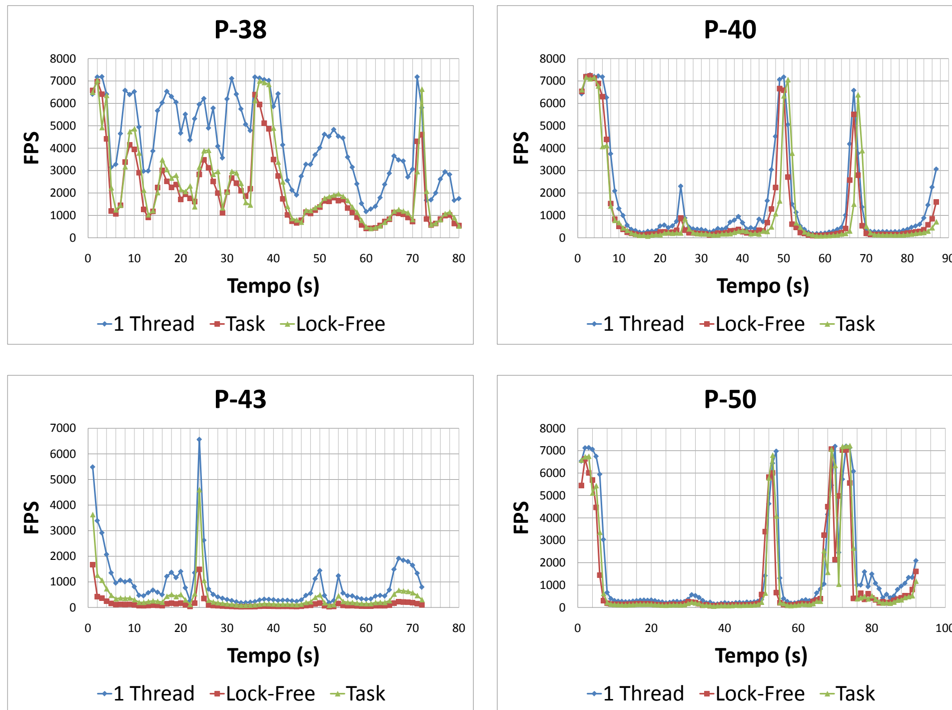


Figura 4.15: Multiprocessamento com troca de tarefas entre threads nas plataformas.

Os resultados obtidos com a implementação do algoritmo *Lock-free Scheme* não foram satisfatórios. Foram encontradas dificuldades de implementação que não estão bem explicadas no trabalho de Assarsson, como por exemplo uma métrica para distribuição dos nós. Outro fator que pode ter influenciado é a quantidade de processadores. Neste trabalho foram utilizados quatro e no trabalho de Assarsson quatorze.

A implementação de *tasks* não obteve o desempenho esperado, uma vez que em teoria poderia obter melhor performance. O baixo desempenho dessa abordagem pode ser explicado por configurações do compilador utilizado e o número de *tasks* a serem criadas.

A conclusão sobre o multiprocessamento é que a princípio não vale a pena o custo de distribuir as tarefas de percurso da hierarquia por mais de uma *thread* e muito menos compartilhá-las entre as *threads* utilizando a biblioteca OpenMP.

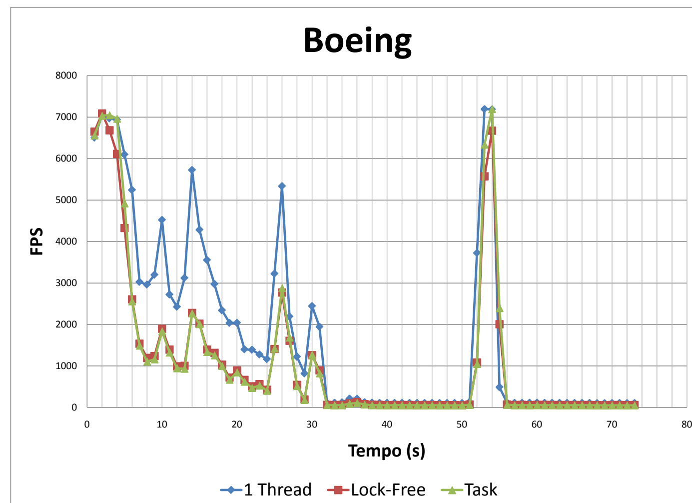


Figura 4.16: Multiprocessamento com troca de tarefas entre threads no Boeing.

4.8 Pipeline final em CPU

O esquema do melhor algoritmo de *frustum culling* implementado em CPU para a maioria dos modelos, pode ser visto na Figura 4.17.

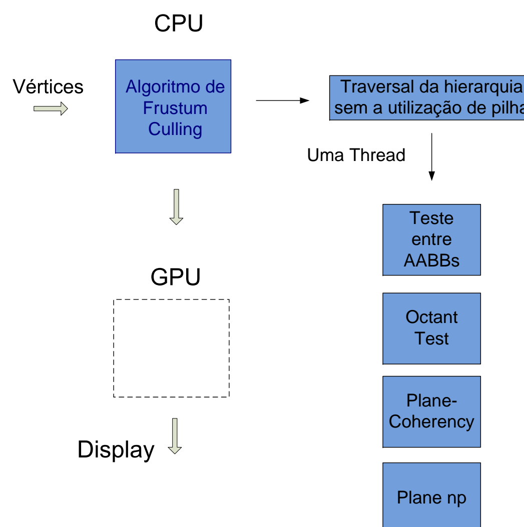


Figura 4.17: Pipeline final em CPU.

5
Frustum culling em GPU

5.1
Timeline gpu

Dois dos pioneiros na área de computação gráfica foram os professores da universidade de Utah, David Evans e Ivan Sutherland, que formaram uma empresa em 1968 chamada Evans and Sutherland que produzia *hardware* para rodar sistemas desenvolvidos na universidade. Grande parte dos seus empregados era formada por estudantes, entre eles Jim Clark que em 1982 iria fundar a Silicon Graphics. Neste período, conhecido como pré-gpu, os *hardwares* tinham propósitos bem definidos, eram muito caros e pouco populares.

Os *hardwares* gráficos ficaram mais populares na década de 90, onde a primeira geração (1994-1998) de placas gráficas ficou conhecida por fazer mapeamento de uma ou duas texturas e rasterização de triângulos pré-processados. Esta geração ficou marcada pelas placas NVIDIA TNT, ATI Rage e 3dfx Voodoo.

A próxima geração de placas, compreendida entre 1999-2000, já suportava a transformação de vértices, iluminação e mapeamento de textura cúbica. Nesta geração a NVIDIA lançou em 1999 a Geforce 256 e criou o termo *graphics processing unit* (GPU) para diferenciar esta placa das outras que só tinham a capacidade de rasterização. Outras placas que tiveram destaque nesta geração foram NVIDIA GeForce 2, ATI Radeon 7500 e S3 Savage3D.

A grande novidade da terceira geração (2001) de placas gráficas foi o suporte ao estágio de vértice programável contendo uma sequência de instruções para o seu processamento. Destaque para as placas NVIDIA GeForce 3, GeForce 4 Ti e ATI Radeon 8500.

A quarta geração (2002-2003) foi marcada pelo suporte a programação do estágio de fragmento que antes não era suportado. Outras características desta geração foram a possibilidade de *loop* no programa de vértices e o aumento das instruções para os estágios programáveis. As placas NVIDIA GeForce FX 5950, ATI Radeon 9700 e ATI Radeon 9800 foram as que se destacaram.

A quinta geração (2004-2006) disponibilizou suporte a *multiple render*

target, *loop* e diretivas condicionais no programa de fragmentos. Nessa geração as principais placas foram NVIDIA GeForce 6800, NVIDIA GeForce 7800, ATI Radeon X800 e ATI Radeon X1800.

A sexta e atual geração (2007-hoje) de placas disponibilizou um novo estágio programável conhecido como estágio de geometria e mudanças para arquitetura unificada que aloca dinamicamente o processamento de cada um dos estágios, minimizando assim a ociosidade. Atualmente as placas mais poderosas são NVIDIA GeForce 280GTX e ATi Radeon HD 4870.

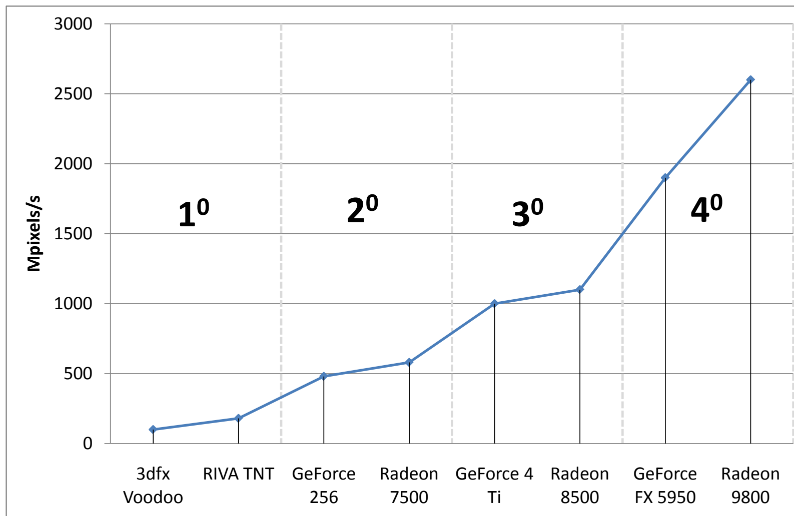


Figura 5.1: Evolução das placas gráficas até a quarta geração.

Os avanços entre as placas de gerações diferentes também foi marcado pelo grande aumento do poder de processamento, como pode ser visto na Figura 5.1. A medida mais comum utilizada para avaliar a evolução das placas até a quarta geração foi o número de *pixels* que a placa consegue renderizar por segundo. Na Figura 5.1, este número esta expresso em milhões.

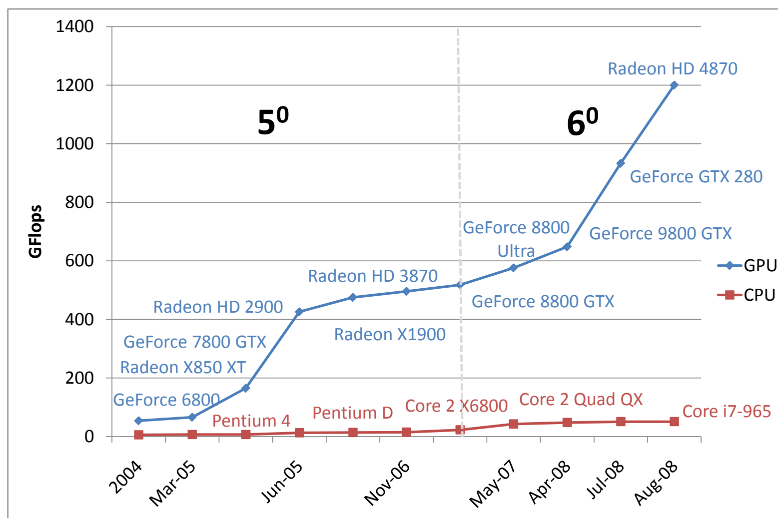


Figura 5.2: Evolução das placas gráficas a partir da quarta geração.

A partir da quarta geração, a medida passou a ser *flops* (Seção 1.1), como pode ser visto na Figura 5.2, onde a medida é expressa em Giga *flops*. Pode ser observado também que o poder de processamento das placas gráficas vem evoluindo bem mais rápido que o dos processadores, superando com folga a lei de Moore (Seção 1.1).

5.1.1 Estágio programáveis e GPGPU

Como foi dito anteriormente, a partir da quarta geração de placas gráficas alguns estágios do *pipeline* da placa gráfica tornaram-se programáveis. Desde então o número de instruções aumenta constantemente e as linguagens para programação em placa vem se tornando cada vez mais amigáveis. Atualmente o *pipeline* dentro da placa pode ser descrito como ilustra a Figura 5.3 (imagem baseada em [44]), onde as caixas verdes indicam que o estágio é programável, as amarelas indicam estágio configurável e os azuis são fixos.

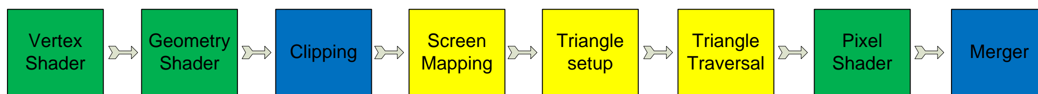


Figura 5.3: Pipeline das placas gráficas modernas.

O grande poder de processamento, estágios programáveis e arquitetura SIMD (Seção 3.2.6) motivaram o surgimento de uma nova técnica conhecida como *GPGPU* (*General-purpose computing on graphics processing units*) que utiliza a GPU para resolver problemas anteriormente solucionados na CPU. Muitos algoritmos já têm sua versão migrada para a GPU sendo processados em menos tempo, como na área de visão computacional, busca, ordenação, processamento de áudio e vídeo, entre outros. Atualmente as grandes dificuldades de portar algoritmos para a GPU são vetorizar os algoritmos e a ausência de um ambiente de desenvolvimento maduro.

A Seção 5.2 apresenta uma forma renderização de primitivas desenvolvida por Toledo [70] que utiliza os estágios programáveis da placa gráfica para renderizar primitivas e a Seção 5.3 levanta formas de realizar o algoritmo de *frustum culling* utilizando em alguns casos técnicas de GPGPU.

5.2 Gpu primitives

Uma das contribuições da tese de doutorado de Toledo [70] foi o desenvolvimento de um *framework* de *ray casting* em GPU para a renderização de primitivas como cones, cilindros e torus. Este *framework* funciona de forma

híbrida, possibilitando assim a inserção de objetos que sofreram *ray cast* na GPU no mesmo *buffer* de imagem dos objetos rasterizados da forma tradicional. O problema de visibilidade entre os objetos rasterizados de forma diferente é resolvido atualizando o *z-buffer* dos dois métodos. Os objetos que são renderizados através desse *framework* são chamados de *GPU primitives*. O *pipeline* de renderização das *GPU primitives* é dividido em *vertex shader* e *pixel shader*. O *vertex shader* calcula a posição final dos vértices nas coordenadas do olho e transmite algumas informações que serão necessárias no próximo estágio. Para otimizar o estágio do *pixel shader*, apenas um conjunto de *pixels* são utilizados. Esse conjunto de *pixels* é classificado como *Ray-Casting Area* (RCA). A Figura 5.4 ilustra as variáveis de entrada e saída do estágio do *vertex shader*. As variáveis de entrada são transmitidas a partir da CPU para calcular a posição final dos vértices e transmiti-las adiante no *pipeline*.

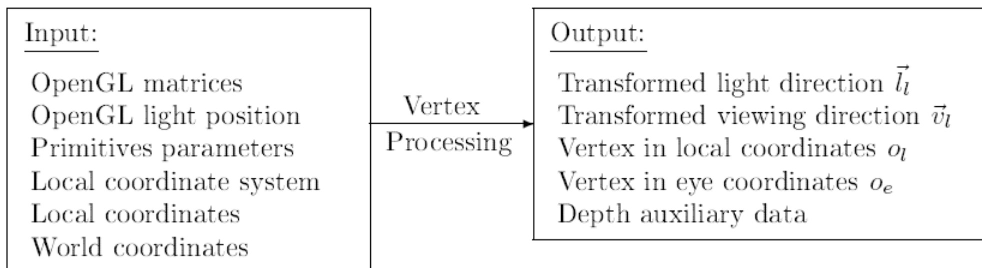


Figura 5.4: Variáveis do vertex shader das gpu primitives (extraída de [70]).

Já o *pixel shader* executa a interseção entre o raio (definido pela origem e direção do observador recebidas do *vertex shader*) e a superfície da primitiva. Caso não haja interseção o *pixel* é descartado. A Figura 5.5 mostra as informações recebidas do *vertex shader* e a saída do estágio do *pixel shader*.

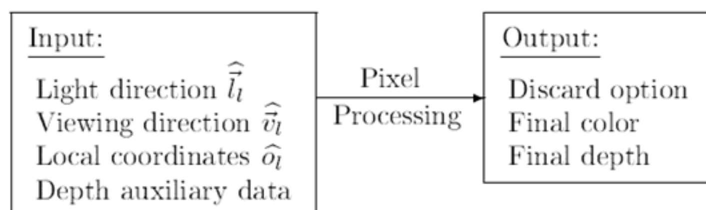


Figura 5.5: Variáveis do pixel shader das gpu primitives (extraída de [70]).

A utilização das *gpu primitives* tem vantagens de performance, qualidade e memória. Segundo o autor a performance na maioria dos modelos é duas vezes maior quando comparada com a utilização de malhas triangulares. A qualidade é a melhor possível, uma vez que os detalhes das primitivas são tratadas no nível de *pixel*. A memória utilizada pelo *framework* se restringe às informações

paramétricas da primitiva que vão estar alocadas na forma de textura na GPU e o volume envolvente que sofrerá *ray cast*. Mais detalhes sobre as vantagens das *gpu primitives* podem ser encontradas em [19]. As Figuras 5.6, 5.7 e 5.8 ilustram as informações paramétricas de algumas primitivas assim como a saída dos estágios de *vertex* e *pixel shader*. Todas essas informações são passadas para os *shaders* na forma de textura e o acesso é feito através de coordenadas de textura. A quantidade de vértices que serão passados para o *vertex shader* depende da primitiva.

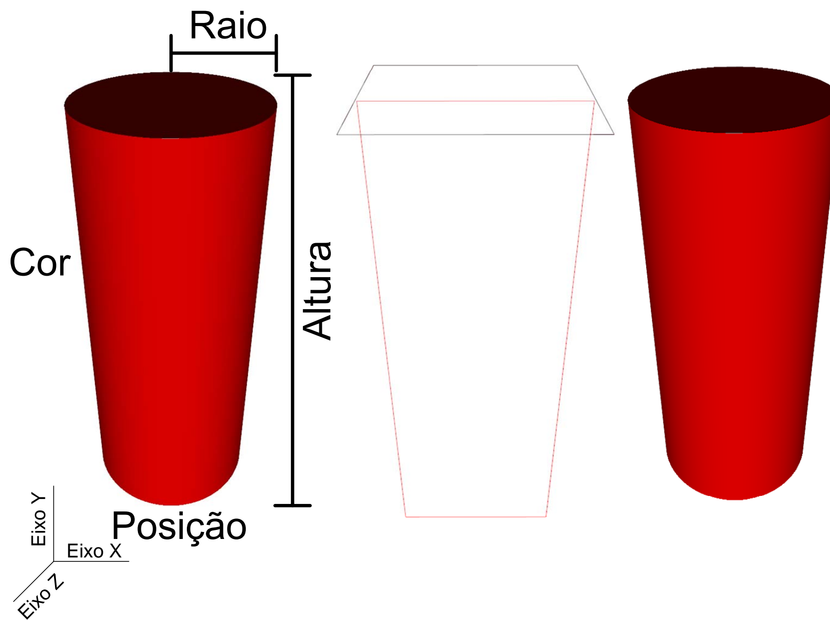


Figura 5.6: Cilindro Gpu primitives.

Para a renderização de um cilindro utilizando as *gpu primitives* é necessário saber a posição inicial do cilindro representado por três valores, o raio do seu tampo representado por apenas um valor e a altura é expressa por um vetor 3-D que informa também a sua direção. Com isso são necessários dois *texels* [1] para alocar estes valores. Além das informações passadas por textura, também é necessário passar quatro vértices que servirão de *input* para a construção da RCA.

O cone é representado por dois vetores que informam a posição e a direção para onde o cone cresce e dois raios, sendo um para base e o outro para o topo. Além do cone da Figura 5.7, também é levantado por Toledo *et al.* [70] o cone truncado, sendo o número de vértices a sua única diferença. Enquanto o cone truncado utiliza oito vértices, o outro precisa de apenas cinco. Nas duas

[1] *texels - é a representação de um elemento em uma textura. Muito similar a um pixel, o texel possui valores r,g,b e a. As GPUs atuais permitem texturas descritas em floating-point, ou seja, cada texel pode comportar até quatro escalares.*

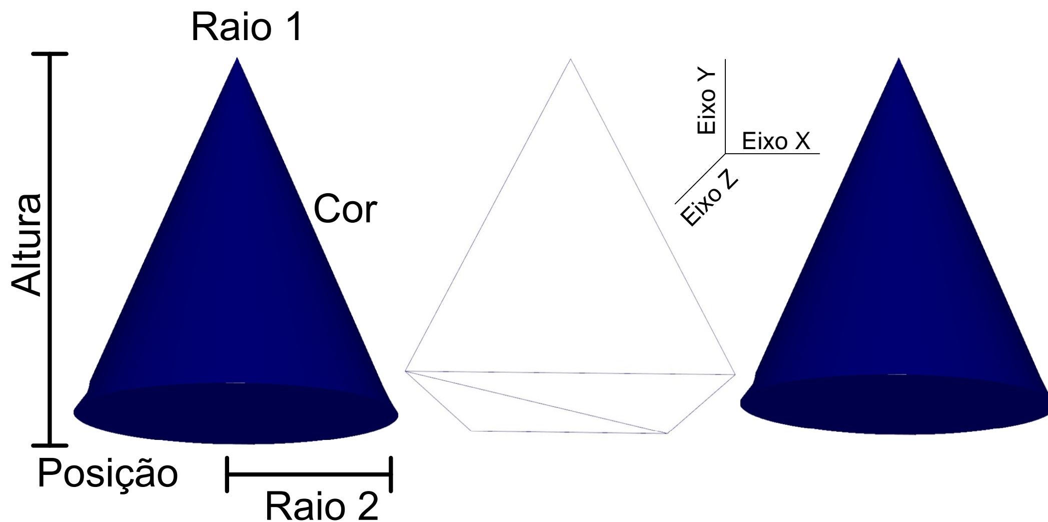


Figura 5.7: Cone GPU primitives.

situações dois *texels* são o suficiente para as informações, pois compartilham os mesmos dados.

A última primitiva analisada neste trabalho é o *torus slice* (joelho), que pode ser descrito com sua posição, dois vetores para identificar a direção para onde o torus vai crescer e um ângulo que informa o quão aberto ele será. Para guardar esses dez valores em textura são necessários três *texels* e quatorze vértices para a construção do RCA no *vertex shader*.

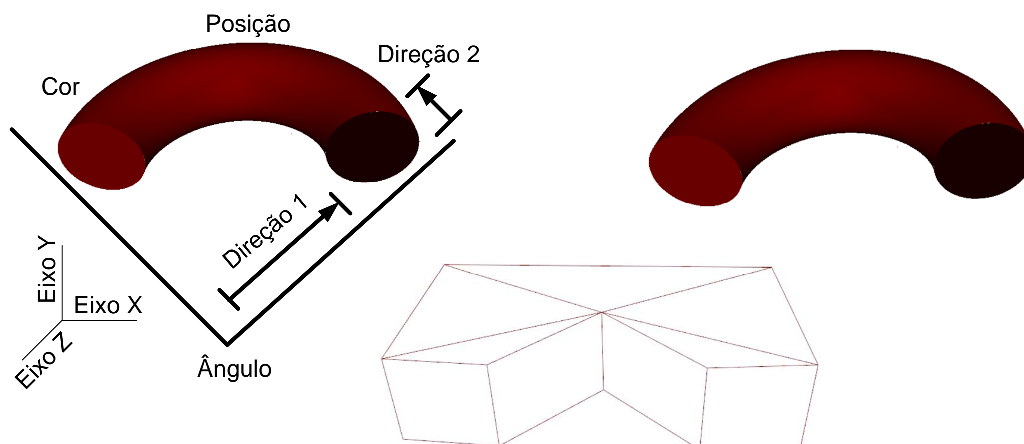


Figura 5.8: Torus slice GPU primitives.

Dois tipos de modelos são utilizados por Toledo *et al.* [70]. O primeiro são modelos com malhas triangulares que sofrem engenharia reversa a fim de encontrar as primitivas presentes no modelo que poderão ser substituídos pelas *gpu primitives*. O segundo são os modelos TDGN já apresentados na Seção 3.4.

Como foi dito na Seção 5.1, a partir da série 8 das GPUs da Nvidia surgiu um novo estágio programável chamado de *geometry shader*, localizado

entre os estágios de *vertex* e *pixel shader*. Suas principais funcionalidades são a possibilidade de emitir ou não primitivas iguais ou diferentes das fornecidas como entrada e desabilitar o estágio de rasterização. A próxima seção irá explorar esse novo estágio a fim de inserir o algoritmo de *frustum culling* na placa gráfica.

5.3 Algoritmos de frustum culling em GPU

Esta seção irá explorar possíveis modos de inserção do algoritmo de *frustum culling* na GPU para as *gpu primitives* e para os modelos com malhas triangulares.

5.3.1 Frustum culling nas GPU primitives

A primeira forma de implementação do algoritmo de *frustum culling* na placa gráfica teve o intuito de eliminar o estágio de *pixel shader* das *gpu primitives* que não estejam visíveis de forma rápida. Isso é vantajoso uma vez que o estágio de *pixel shader* é o mais custoso na maioria das primitivas do *framework*. Para implementação desta abordagem, duas informações a mais são passadas para a GPU: os planos do *frustum* e os volumes envolventes das primitivas. A utilização de planos ao invés de radar para a implementação do algoritmo foi baseada na boa performance do teste contra apenas dois vértices da AABB feito em CPU. As etapas dos cálculos executados pela GPU podem ser vistas na Figura 5.9.

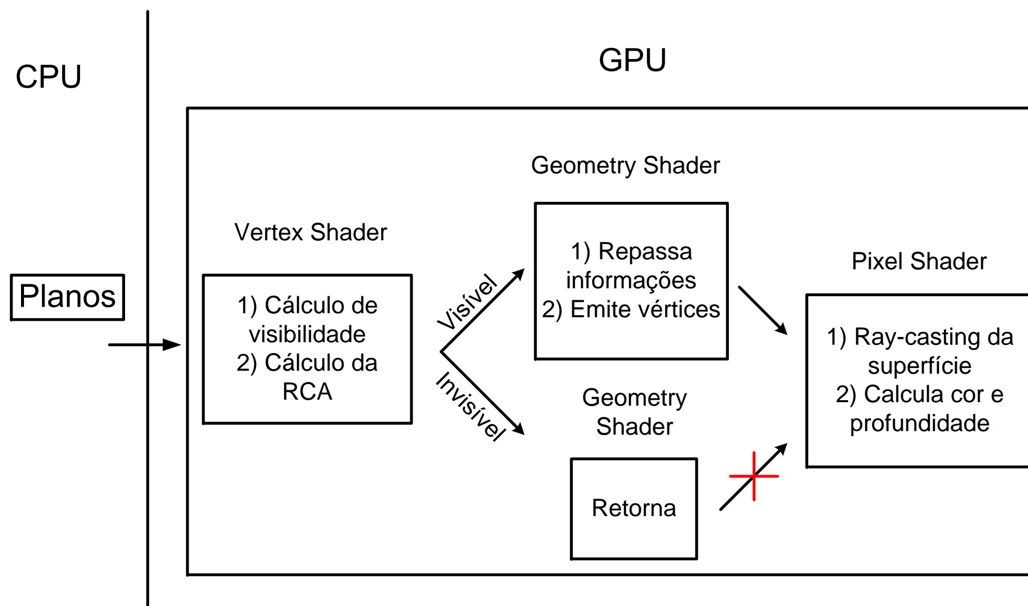


Figura 5.9: Frustum culling junto com GPU primitives.

Depois de receber o *input* das variáveis necessárias para o cálculo do *frustum culling* e renderização das primitivas, o primeiro passo é determinar se a primitiva está visível. Caso esteja, as posições finais de seus vértices são calculadas e enviadas para o *geometry shader* seguindo o fluxo normal do *pipeline*, caso contrário o *geometry shader* não envia informação para o *pixel shader*. A utilização do estágio de *geometry shader* tem a função de filtrar as primitivas que estejam invisíveis, porém a sua inserção no *pipeline* traz outros problemas que serão discutidos com mais detalhes na Seção 5.6.

Outra forma de descarte das *GPU primitives* desenvolvida foi a utilização de um *shader* separado, ou seja, fora da renderização das primitivas. A ideia básica é enviar os dados necessários para fazer os cálculos e retornar os resultados de maneira que possam ser aproveitados como *input* para a renderização das primitivas. A Figura 5.10 mostra o esquema do algoritmo.

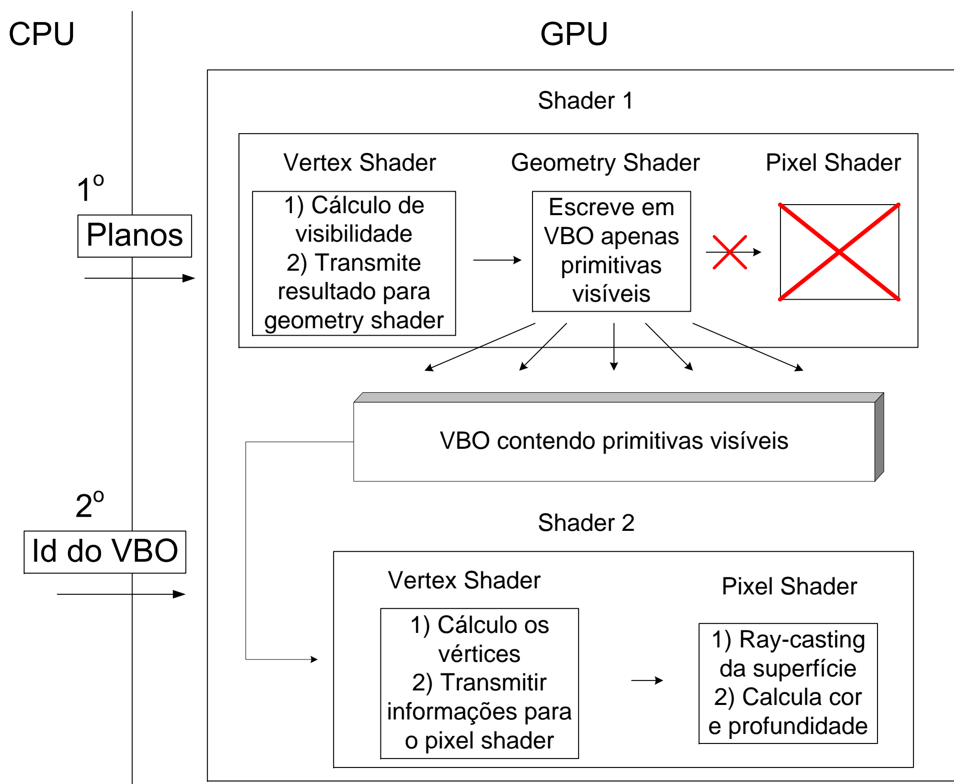


Figura 5.10: Frustum culling em shader separado.

Primeiramente o *shader* que vai realizar os cálculos de *frustum culling*, representado pelo número 1 na Figura 5.10, recebe os parâmetros (planos do *frustum* e volumes envolventes) da CPU. Os resultados (apenas primitivas visíveis) são guardados na memória da GPU e servirão de *input* para a renderização das primitivas visíveis representado pelo número 2. A fase de rasterização do primeiro *shader* pode ser desligada por não ter cálculos para serem feitos.

As duas formas de *culling* implementadas para as *GPU primitives* só levaram em conta os cilindros, porém a inserção das outras é possível. Essa decisão foi tomada pois o cilindro faz uso de apenas quatro vértices e é mais fácil de ser implementado. No primeiro algoritmo, como são enviados quatro vértices para a renderização, o processo de descarte é dividido entre seus vértices. Desta forma cada vértice processa dois dos seis planos, sendo que o último vértice repete o teste com dois planos. O resultado de cada um dos vértices é enviado para o estágio de *geometry shader* e os vértices só são emitidos caso todos estejam visíveis. No segundo algoritmo esta divisão não precisa ser feita, pois apenas um vértice por primitiva é enviado para a placa gráfica em um *shader* separado e caso esteja visível ou interceptando o plano, os quatro identificadores de acesso à textura do cilindro são gerados e guardados na memória da GPU para servirem de *input* no segundo *shader*.

5.3.2 Frustum culling em modelos genéricos

A utilização da placa gráfica para realizar os cálculos de *frustum culling* em modelos de malhas triangulares não pode ser feita da mesma maneira que foram tratadas as *gpu primitives*. Isso porque os vértices não podem ser tratados individualmente, como é feito na primeira abordagem das *gpu primitives* pelo desconhecimento prévio do número de vértices em cada malha e pelo limite de emissão de vértices. Esta limitação também ocorre no caso das *gpu primitives*, porém no máximo quatorze vértices precisam ser emitidos, no caso do (*torus slice*), o que não pode ser determinado para as malhas triangulares. Por último, calcular dinamicamente o volume envolvente das malhas triangulares a cada *frame* é muito custoso. Um possível esquema de tratamento de modelos genéricos pode ser visto na Figura 5.11.

Nesse esquema, os volumes envolventes dos objetos são guardados em textura juntamente com seus identificadores. A ativação do *vertex shader* é feita por pontos que representam cada um dos volumes envolventes. Depois de processados na GPU, os identificadores dos volumes envolventes visíveis são guardados na memória da GPU. Como cada volume envolvente contém um número variável de vértices e possivelmente maior que 1024, é necessário que os identificadores visíveis sejam levados para CPU e posteriormente renderizar os objetos. A vantagem de ter o estágio de *geometry shader* é que apenas os volumes envolventes visíveis são gravados na memória da GPU, diminuindo assim a quantidade de resultados que serão levados para a CPU.

Outra possível abordagem é eliminar o estágio de geometria e escrever todos os resultados (volumes envolventes visíveis e não visíveis diferenciados

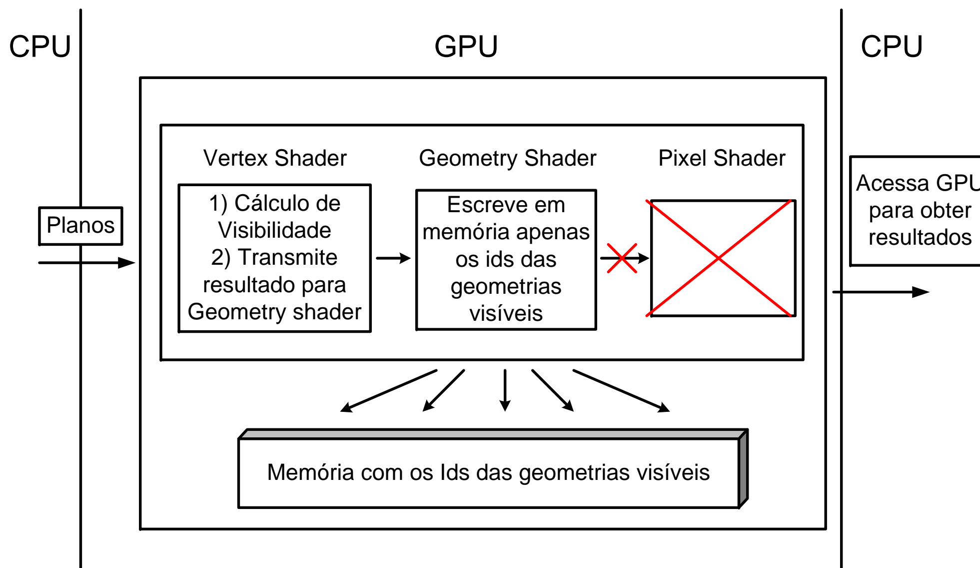


Figura 5.11: Frustum culling em modelos genéricos.

por sinal) na memória da GPU e depois, em CPU, separar os volumes visíveis. Este esquema está ilustrado na Figura 5.12. O impacto de levar os dados para a CPU nos dois esquemas, assim como a comparação com as outras formas de descarte serão discutidos na Seção 5.6.

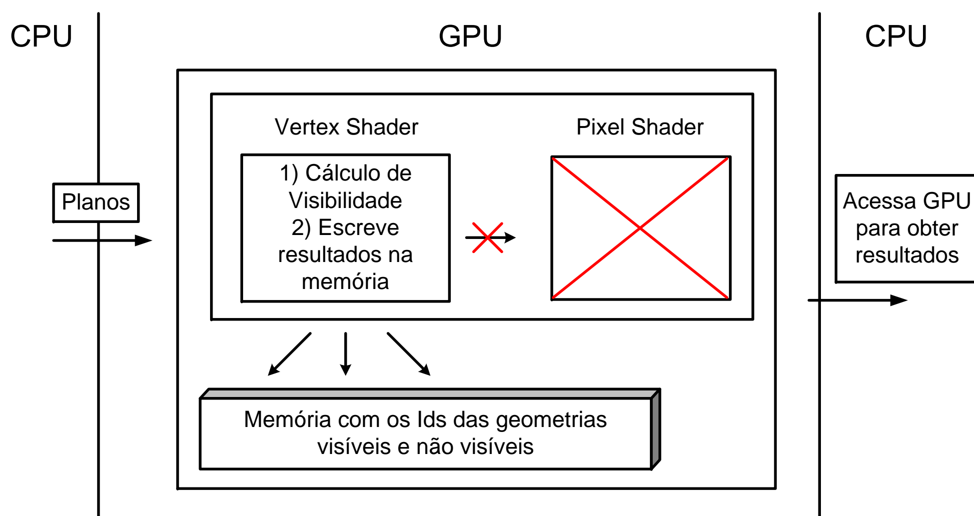


Figura 5.12: Frustum culling em modelos genéricos.

5.4 Memória utilizada em GPU

Para realizar os cálculos do *frustum culling* os volumes envolventes são alocados em forma de textura e consultados dentro dos *shaders*. Como o volume envolvente escolhido foi a AABB, são necessários dois *texels* para guardar os três valores mínimos e os três valores máximos da AABB. A Tabela 5.1 mostra

a quantidade de memória necessária para alocar os volumes envolventes dos modelos utilizados para testes.

Modelos	# Cilindros	# Cone	# Joelhos	# Total	Memória (MB)
P-38	81374	3917	0	85291	2.60
P-40	221933	3814	39586	265333	8.09
P-43	280123	13212	0	293335	8.95
P-50	336591	14192	341168	691951	21.11

Tabela 5.1: Memória necessária para os volumes envolventes.

Mesmo com a aumento de memória disponível em GPU, quando acrescentamos aos volumes envolventes os dados dos modelos como vértices, cores e texturas a memória pode se tornar um problema. Esse problema é agravado quando é utilizada hierarquia, pois a quantidade de nós pode ser maior dependendo do tipo de hierarquia construída, como será visto na próxima seção.

5.5 Percurso sem pilha em GPU

Como foi levantado na Seção 3.2.3, a hierarquia tem papel fundamental para reduzir a quantidade de cálculos a serem realizados no descarte. A utilização de hierarquia em GPU traz problemas de memória, acesso a textura e execução do algoritmo.

A quantidade de memória utilizada normalmente aumenta, uma vez que o número de nós da hierarquia pode gerar mais volumes envolventes que antes, dependendo do tipo de construção. Nas hierarquias utilizadas, a quantidade de volumes envolventes duplicou na maioria dos modelos de testes. A quantidade de informação alocada por volume envolvente na textura não precisa ser aumentada, uma vez que apenas um valor precisa ser adicionado (o *escape index*). Esse valor pode ser alocado utilizando os mesmos dois *texels* de antes como pode ser visto na Figura 5.13.

O percurso da hierarquia em CPU envolve acesso a memória para identificar o próximo nó a ser processado. Essa operação em GPU, que envolve acesso a textura, pode se tornar o gargalo caso o número de acessos seja elevado. Esse caso pode ocorrer quando há um grande número de volumes envolventes interceptando o *frustum* de visão. O principal problema de realizar a operação de percurso da hierarquia na GPU é a ordem fixa de execução dos cálculos, descrito com mais detalhes em [48]. Com isso o poder de processamento em paralelo da GPU não é explorado. A Seção 5.6 discutirá a viabilidade de inserção do percurso da hierarquia em GPU nos algoritmos propostos.

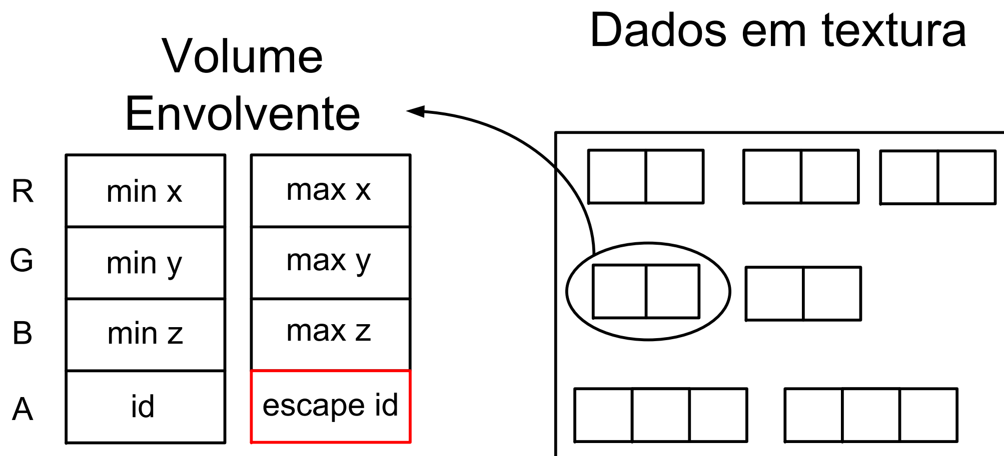


Figura 5.13: Memória utilizada no percurso em GPU.

5.6 Implementação

A implementação do algoritmo de *frustum culling* em GPU foi dividida em dois grupos de modelos, o primeiro contendo apenas *GPU primitives* e o segundo contendo apenas malhas triangulares. Nos algoritmos para as *GPU primitives*, foram propostos dois tipos de algoritmos diferentes.

As duas técnicas implementadas não necessitam trazer os resultados do *frustum culling* para a CPU. A primeira executa os cálculos no mesmo *shader* de renderização das *GPU primitives* e a outra utiliza um *shader* a parte. Os dois algoritmos não possuem nenhum tipo de otimização ou hierarquia além do teste de apenas dois vértices contra os planos. Como pode ser visto na Figura 5.14, o desempenho melhora quando a técnica do *shader* separado é utilizada.

O algoritmo que utiliza um *shader* separado obteve melhor performance, na maioria dos *frames*, quando comparado com o que utiliza o mesmo *shader* de renderização. A grande vantagem desse algoritmo é que os cálculos de *frustum culling* são feitos apenas uma vez por vértice e apenas primitivas visíveis são enviadas para o *shader* de renderização, porém em alguns momentos o caminho de câmera sem *frustum culling* obteve melhor performance que o algoritmo de *shader* separado.

No caso dos modelos contendo apenas malhas triangulares foram implementados dois algoritmos em GPU. Os dois precisam trazer os resultados para a CPU, diferenciando apenas da presença ou não do estágio de geometria. A Figura 5.15 ilustra o desempenho desse algoritmo comparado com o melhor algoritmo conseguido CPU. Os três algoritmos não possuem nenhuma otimização e nem hierarquias.

A ampla vantagem do algoritmo puramente em GPU frente ao melhor

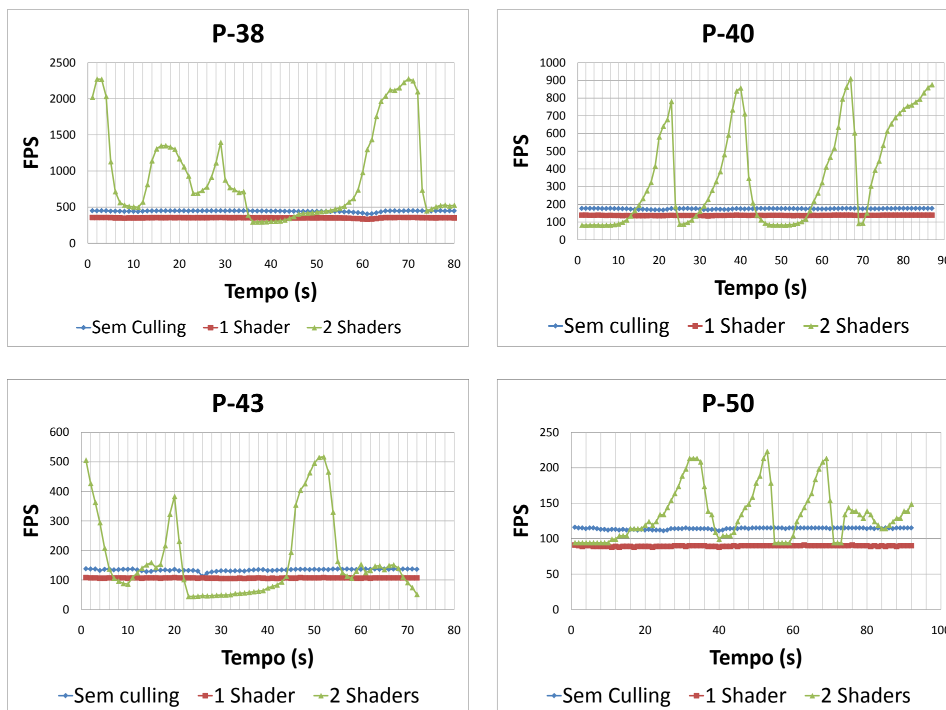


Figura 5.14: Frustum culling nas GPU primitives.

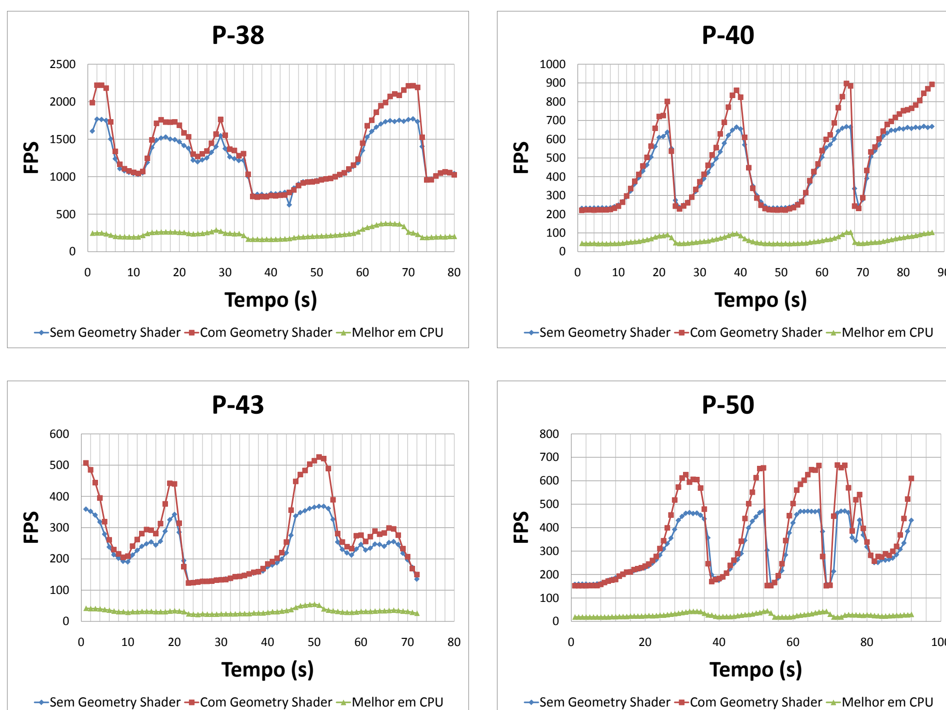


Figura 5.15: Frustum culling em GPU para modelos genéricos.

em CPU se deve ao grande poder de processamento das GPUs, quando o algoritmo é processado em paralelo. Na maioria dos casos, a utilização do *geometry shader* aumentou a performance dos resultados uma vez que menos resultados são trazidos da GPU para CPU e posteriormente não é necessário separar os elementos visíveis dos outros na CPU. O gargalo das duas aplicações se encontra na necessidade de trazer os resultados da GPU para a CPU.

Todos os resultados apresentados até agora não utilizaram nenhum tipo de hierarquia. Além dos problemas de memória, acesso a textura e não paralelismo, a inserção da hierarquia não é possível nos algoritmos propostos para modelos genéricos, pois o estágio de vértice não é capaz de escrever na memória da GPU todos os resultados do percurso a partir do *input* de apenas um vértice e o estágio de geometria está limitado à 1024 escritas na placa em questão. Por esses motivos o percurso de toda hierarquia utilizando apenas a GPU não foi implementado, porém será de suma importância no algoritmo de *frustum culling* híbrido no Capítulo 6.

5.6.1
Pipeline final em GPU

Para as *GPU primitives*, o melhor algoritmo conseguido utiliza um *shader* separado do de renderização para processar o algoritmo de *frustum culling*. A grande vantagem deste comparado com os algoritmos desenvolvidos para os modelos de malhas triangulares é a não necessidade de trazer os resultados obtidos para a CPU o que diminui a performance consideravelmente.

Para os modelos com malhas triangulares o melhor algoritmo obtido escreve os identificadores das primitivas visíveis a partir do estágio de geometria.

Os grandes problemas que dificultaram a utilização da placa gráfica para o algoritmo de *frustum culling* foram a não utilização de hierarquia, o que força a execução de um número muito grande de descartes e a necessidade de trazer um número muito grande de dados da GPU para a CPU. Mesmo que a GPU tenha superado a CPU nos cálculos de descarte, a não utilização de hierarquia torna-se inviável em modelos massivos, por isso esse problema foi contornado no algoritmo de *frustum culling* híbrido que será discutido no próximo capítulo.

6
Frustum culling híbrido

Com a evolução rápida do poder de processamento das placas gráficas, muitos algoritmos têm migrado suas partes ou todas suas rotinas para GPU. Porém, vetorizar alguns problemas nem sempre é uma tarefa fácil e em alguns casos não vale a pena. Este trabalho visou até agora mostrar o estado da arte do *frustum culling* em CPU e maneiras de torná-lo portável para a GPU. Este capítulo propõe algumas heurísticas para determinar o momento ideal, onde a CPU possui pior performance, para utilização da GPU a fim de acelerar o algoritmo de determinação dos objetos visíveis.

6.1
Heurísticas

O estado da arte do algoritmo de *frustum culling* em CPU obtém boa performance em cenas pequenas e médias. O grande impacto no seu desempenho ocorre em cenas com grande número de objetos, comum em modelos CAD. Isso ocorre devido à grande quantidade de nós internos da árvore e consequentemente o excessivo número de testes contra o *frustum*. Os resultados conseguidos em GPU confirmaram o grande poder de processamento da placa gráfica quando os problemas são tratados em paralelo. A utilização de hierarquia comprometeu a execução dos algoritmos desenvolvidos em GPU, uma vez que o percurso deve respeitar uma ordem fixa e a existência de limitações no estágio de *geometry shader*. Mesmo tendo seus pontos fracos, tanto a implementação em CPU quanto em GPU apresentam boa performance em situações diferentes.

A fim de minimizar o impacto da inserção do *frustum culling* no *pipeline* das aplicações que visam a manipulação de modelos massivos, foi implementado o *frustum culling* híbrido. A ideia básica é inicialmente fazer descarte de primitivas utilizando o melhor algoritmo conseguido em CPU e quando for identificado que a quantidade de cálculos está elevada ao nível de influenciar a performance da aplicação, a GPU assume o controle do algoritmo de *frustum culling* de forma que seu tempo de processamento seja menor que o da CPU. As grandes questões dessa abordagem são decidir quando é o momento certo

de tratar os dados na GPU, como fazer com que todo o poder da GPU seja utilizado e quando retornar os cálculos para a CPU. Esses assuntos serão abordados nas Seções 6.1.1 e 6.1.2.

6.1.1
Identificação do momento ideal

A identificação do momento para a utilização da GPU é crucial para melhorar a performance da aplicação, pois se não for bem escolhido pode diminuir o desempenho. Idealmente a CPU deve controlar o processamento do algoritmo de *frustum culling* na maior parte do tempo possível, isso porque possui hierarquia e técnicas de otimização que diminuem o número de cálculos a serem feitos. A medida que os cálculos vão crescendo, a única saída da CPU para acelerar a performance do algoritmo é adotar alguma técnica mais conservativa que acabaria enviando dados não visíveis para serem renderizados. A transição de *hardware* para execução do algoritmo servirá de alternativa para o baixo desempenho da CPU em momentos de muitos cálculos e ao mesmo tempo evitar que objetos não visíveis sejam processados desnecessariamente. As Figuras 6.1 e 6.2 fazem uma comparação das performances dos melhores algoritmos conseguidos em CPU e em GPU.

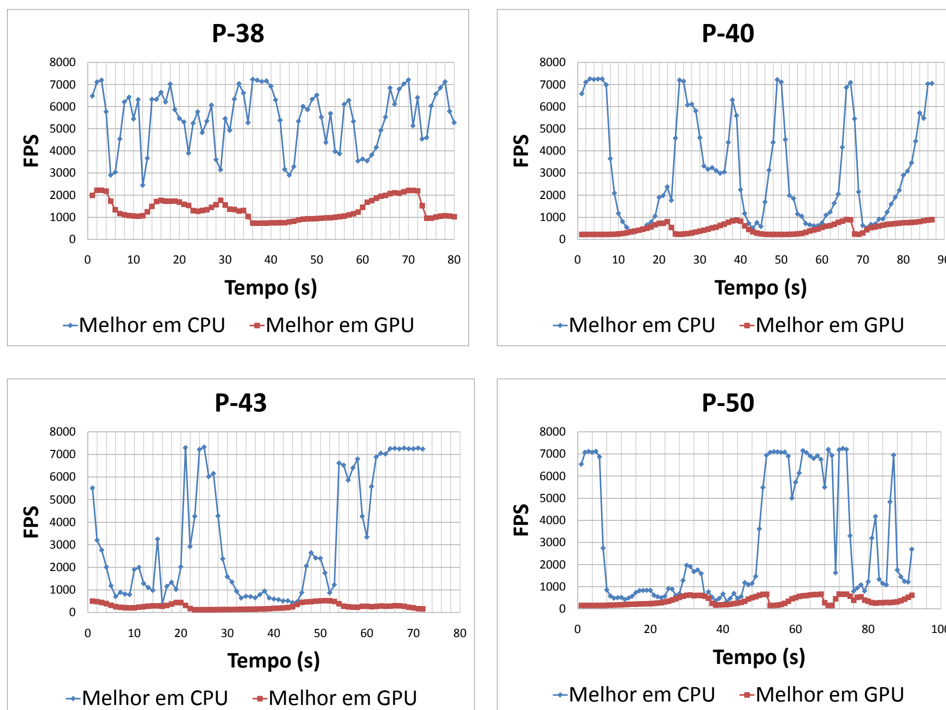


Figura 6.1: Melhores caminhos de câmera em CPU e GPU nas plataformas.

Os melhores resultados com cada um dos *hardwares* executando o algoritmo de *frustum culling* separadamente, mostra que a CPU sempre obtém

melhor performance que a GPU. Apenas em alguns momentos dos modelos maiores a performance da GPU alcança o desempenho obtido pela CPU. Vale lembrar que o algoritmo em CPU possui hierarquia e várias técnicas de aceleração aclopadas ao algoritmo de *frustum culling*, enquanto a GPU só tem a técnica de aceleração onde dois vértices são testados contra o *frustum*, processando todos os volumes envolventes sem hierarquia.

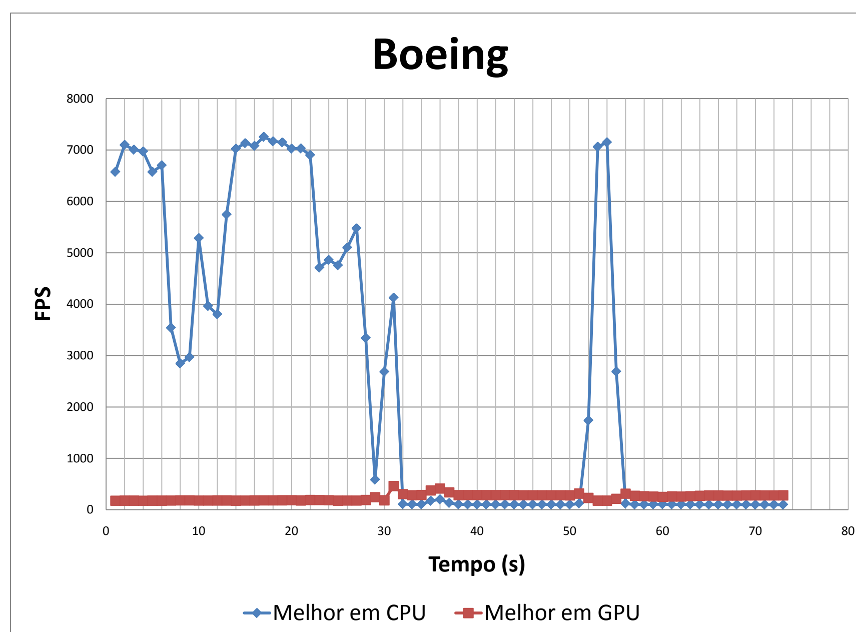


Figura 6.2: Melhores caminhos de câmera em CPU e GPU no Boeing.

No caso do Boeing, a GPU, mesmo processando todos os volumes envolventes, supera o desempenho da CPU em vários *frames*. Tanto no caso das plataformas quanto no caso do Boeing, existem várias quedas de desempenho no algoritmo puramente em CPU. Esses momentos de queda de desempenho da CPU, seriam os momentos mais adequados para a migração do *frustum culling* para a GPU.

Foram experimentadas quatro formas para identificação do melhor momento para utilização da GPU, descritas com mais detalhes abaixo.

1. Altura da hierarquia - identifica gargalos a partir do momento em que o percurso da hierarquia ultrapassa uma determinada altura.

2. Número de interseções - como uma das causas do gargalo do algoritmo ocorre quando há excessivo número de interseção entre os volumes envolventes e o *frustum*, este identifica os gargalos a partir de um certo número de interseções.

3. Porcentagem de nós processados - neste caso, a identificação é feita a partir do momento que o número de nós processados ultrapassa um determinado valor.

4. Tempo de processamento - a identificação neste caso é feita através do tempo de processamento gasto para realizar o percurso da hierarquia.

A Seção 6.2 discute o desempenho de cada uma destas heurísticas e identifica qual delas foi escolhida para ativar o uso da GPU. Tendo os momentos de transição dos cálculos da CPU para a GPU identificados, falta tornar o algoritmo vetorizável a partir dos resultados obtidos na CPU e tirar proveito do poder de processamento da GPU, o que será discutido na próxima seção.

6.1.2 Paralelização do algoritmo

Os grandes problemas de realizar o percurso da hierarquia totalmente em GPU são:

1. O percurso tem que seguir uma ordem, o que dificulta a paralelização do algoritmo.

2. Envolve muitos acessos a textura, que em modelos massivos diminuem a performance do algoritmo.

3. Limitação no *output* do estágio de *geometry shader*, que não permite escrita de todos os resultados em memória quando são utilizados modelos massivos.

Com isso, a ideia é processar apenas uma parte dos nós em paralelo, uma vez que se todos os nós forem processados, a peformance global vai diminuir, como foi visto nos gráficos da seção anterior. Para tal é necessário que toda a hierarquia esteja disponível na GPU para eventuais consultas. Além de fazer o processamento em paralelo, é importante que o resultado fornecido pela GPU contenha apenas as geometrias visíveis, o que envolve o percurso da hierarquia em GPU. Como a ideia é processar apenas uma parte da hierarquia e em paralelo, os acessos a textura são diluídos entre vários nós, o que não acontecia no percurso completo da hierarquia em GPU. Além disso, o processamento dos nós em paralelo contorna a limitação do *geometry shader* de escrever em memória, pois cada nó processado poderá escrever até 1024 resultados na placa utilizada nos testes.

Para viabilizar a utilização da GPU em alguns momentos, juntamente com a CPU, foi explorada a ideia da coerência temporal, que funciona da seguinte forma: se em um determinado *frame* x de processamento em CPU é identificada a necessidade (gargalo em CPU) de uso da GPU, os índices dos últimos nós processados em CPU são enviados para a GPU para que no *frame* $x+1$ eles sejam processados em paralelo na GPU. Assim, pela coerência temporal, é bem possível que os nós processados em GPU não necessitem fazer muitas consultas à textura na operação de percurso pela hierarquia. A Figura 6.3 mostra o esquema do algoritmo.

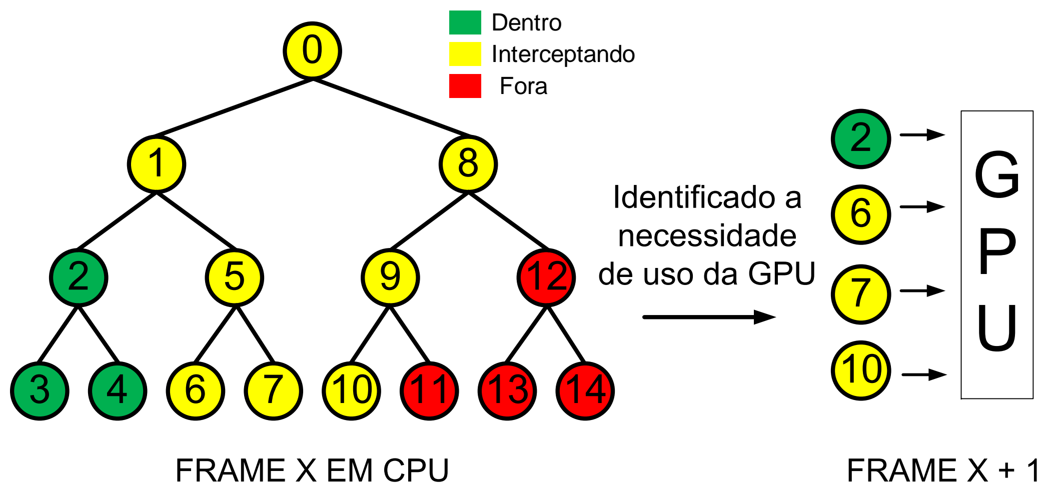


Figura 6.3: Esquema do algoritmo de frustum culling híbrido.

Na Figura 6.3 os nós 2, 6, 7, 10 foram os últimos nós classificados como visíveis, ou seja, suas geometrias participam da imagem final. Uma vez que os nós estejam sendo processados em GPU, os resultados emitidos se tornarão a entrada do *frame* seguinte. Desta forma é mantida a coerência temporal entre *frames*, diminuindo assim o percurso na GPU, o acesso elevado a textura e possíveis erros gerados pela limitação do *geometry shader*. Caso o resultado de um dos nós for negativa, ou seja, fora do *frustum* de visão, o nó pai correspondente deve ser adicionado como *input* de processamento no próximo *frame*. Por exemplo, se no *frame* x o nó 10 for processado em GPU e seu resultado der fora do *frustum*, no *frame* $x + 1$, o nó 9 deve ser processado.

Resolvido o problema de entrada e manutenção do algoritmo em GPU, é necessário também monitorar o momento para deixar de usar a GPU. Isso porque se o algoritmo rodar apenas na GPU, os momentos em que a CPU processa a hierarquia mais rápida serão desperdiçados. Esses momentos ocorrem porque o retorno dos resultados da GPU para a CPU demora mais tempo que a própria execução do algoritmo em GPU tornando-se assim o gargalo quando a GPU é utilizada. Um exemplo dessa situação pode acontecer

quando a câmera está interceptando muitos nós em um determinado *frame* e nos *frames* seguintes nada é observado. Para que a inserção da GPU no *pipeline* não diminua a performance do algoritmo, foram desenvolvidos estágios de transição entre CPU e GPU, como pode ser observado na Figura 6.4.

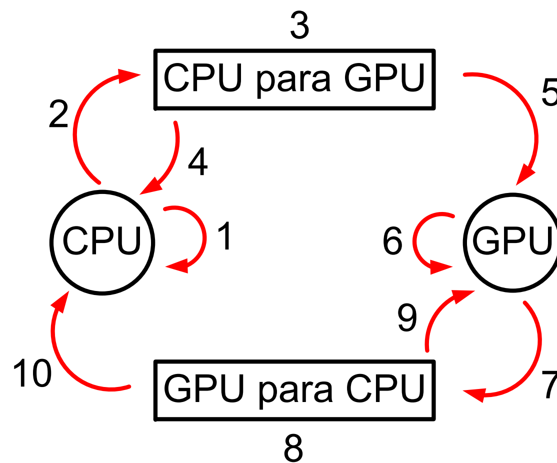


Figura 6.4: Possíveis estados do frustum culling híbrido.

1. Inicialmente o descarte de volumes envolventes é feito no melhor algoritmo conseguido em CPU, sendo verificado a cada iteração se ocorreu algum gargalo no algoritmo. Caso não ocorra, o algoritmo roda apenas em CPU.

2. Caso ocorra gargalo, o estado corrente muda para o número 3 (CPU para GPU).

3. Neste estágio é verificado se nos n *frames* consecutivos o gargalo ainda permanece na CPU.

4. Caso o gargalo na CPU não permaneça, o valor n é zerado e o estado corrente volta para a CPU.

5. Caso o gargalo na CPU permaneça, o contador deste estágio é zerado e a GPU assume o controle do algoritmo.

6. A GPU permanecerá com o controle do algoritmo por m *frames*.

7. Depois de m *frames* processados em GPU, ocorre uma transição para o estágio "GPU para CPU", onde a CPU retoma o controle do algoritmo.

8. Neste estágio é verificado se no *frame* atual ainda ocorre gargalo na CPU.

9. Caso ainda haja gargalo na CPU o valor m é incrementado e a GPU retoma o controle do algoritmo.

10. Caso não haja mais gargalo a CPU retoma o controle do algoritmo.

O que estamos chamando aqui de gargalo, e os valores de n e m *frames* utilizados são discutidos na seção seguinte.

6.2 Implementação

Para que o esquema do *frustum culling* híbrido funcione bem, os gargalos em CPU e o momento de saída da GPU devem ser bem estimados. Várias abordagens foram tentadas para determinar o momento ideal de entrada na GPU em todos os modelos.

1. A identificação por altura em todos os modelos gerou muitos falsos positivos. Isso se deve ao fato de que nem sempre que o percurso em CPU atingia uma certa altura, um gargalo é identificado.

2. Utilizando o número de interseções, a determinação se mostrou eficiente em alguns modelos, porém a determinação de um valor que se encaixe bem para as diferentes hierarquias dificultou a utilização deste.

3. A identificação por porcentagem dos nós processados não obteve bom desempenho pela dificuldade de determinação do valor ideal para as hierarquias.

4. A identificação por tempo de processamento foi a heurística que melhor se enquadrou em todos os modelos. A ideia é que seja identificado um gargalo em CPU quando o seu tempo de processamento ultrapassar o de *download* dos resultados da GPU para CPU no pior caso de testes. O pior caso dos testes foi realizar *download* de todos os *ids* dos volumes envolventes do Boeing com a duração de 2×10^{-3} segundos. Este valor foi reduzido para 3×10^{-4} segundos para aumentar as identificações dos momentos de transição na maioria dos modelos.

Os valores definidos para entrada e saída da GPU foram cinco e dois respectivamente, ou seja, se o tempo de processamento do algoritmo em CPU ultrapassar o valor máximo (3×10^{-4} segundos) por cinco *frames* consecutivos, a GPU é ativada para realização dos cálculos. Inicialmente a GPU processa 500 *frames*, depois desse tempo a CPU retoma o algoritmo e se em dois *frames*, o tempo de processamento ainda estiver acima do valor máximo o tempo de permanência em GPU é dobrado e a GPU volta a processar o algoritmo.

As Figuras 6.5 e 6.6 ilustram a performance do algoritmo de *frustum culling* híbrido, com as configurações levantadas anteriormente, comparado com o melhor algoritmo conseguido utilizando apenas a CPU.

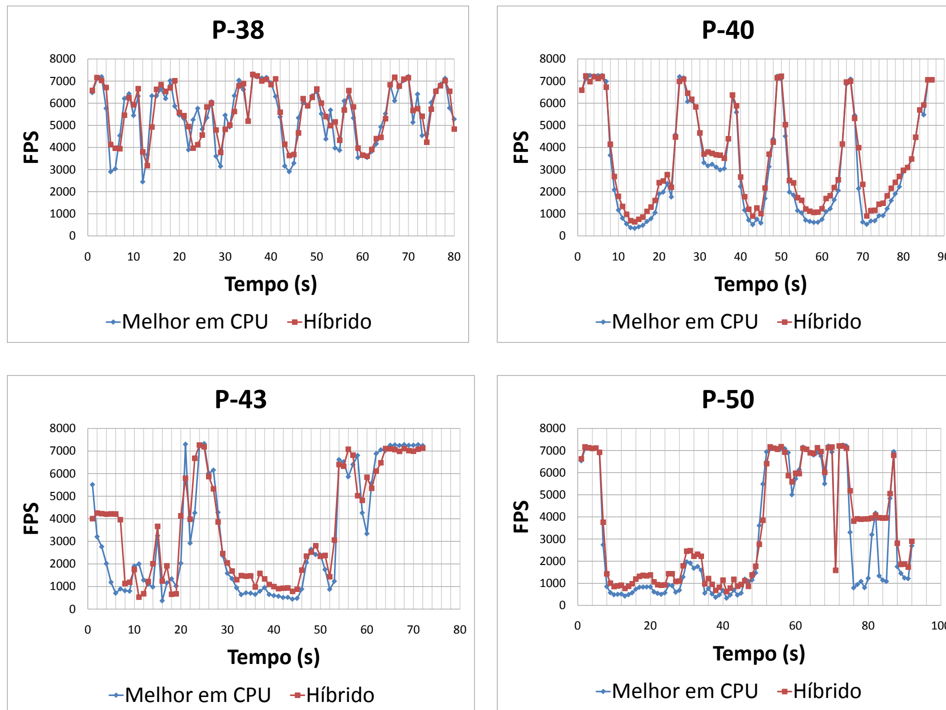


Figura 6.5: Resultados do frustum culling híbrido nas plataformas.

Em todos os testes a presença do *frustum culling* híbrido com os parâmetros especificados sempre melhorou a performance do algoritmo. Em alguns casos como na P-38 a identificação de gargalos não foi muito boa, porém no caso do Boeing, as melhorias foram notórias. Em alguns *frames* é possível notar que a performance fica abaixo do melhor algoritmo em CPU. Isto talvez se deva à perda da coerência temporal utilizada na otimização de *plane-coherency*, ou simplesmente erros de medição.

A Tabela 6.1 mostra com mais detalhes o ganho de performance da abordagem híbrida.

6.2.1 Frustum culling híbrido com a renderização habilitada

Como foi visto anteriormente, em todos os modelos de testes, a abordagem híbrida obteve melhor resultado que as outras estudadas. Porém estes resultados foram obtidos com a renderização desabilitada, ou seja, sem produzir imagem. Quando habilitamos a renderização, o desempenho da aplicação como um todo diminui, pois a placa gráfica passa a ter duas tarefas que em

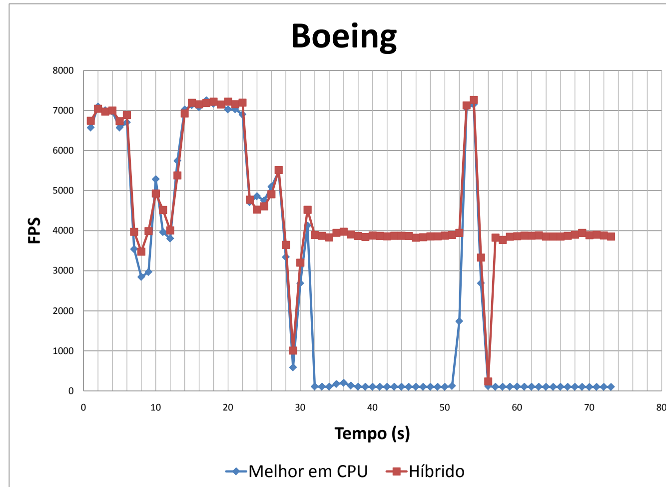


Figura 6.6: Resultados do frustum culling híbrido no Boeing.

Método	Frames Processados	FPS Mínimo	FPS Máximo	Média do Caminho
Melhor CPU na P-38	440594	2368	7241	5494.581
Híbrido na P-38	**451997**	**2756**	**7301**	**5615.358**
Melhor CPU na P-40	274740	320	7263	3143.083
Híbrido na P-40	**305221**	**596**	**7266**	**3490.873**
Melhor CPU na P-43	237059	358	**7334**	3265.365
Híbrido na P-43	**261935**	**527**	7270	**3627.657**
Melhor CPU na P-50	278401	311	**7256**	3022.32
Híbrido na P-50	**323132**	**578**	7241	**3505.028**
Melhor CPU no Boeing	191234	96	7259	2605.687
Híbrido no Boeing	**275646**	**280**	**7300**	**4629.561**

Tabela 6.1: Comparação dos resultados entre melhor algoritmo em CPU e Híbrido.

conjunto acabam tornando-se o gargalo da aplicação. Quando a renderização é chamada, a placa gráfica está ocupada com os cálculos do *frustum culling*, tendo que esperar até que eles terminem. A Figura 6.7 mostra a comparação entre os algoritmos de *frustum culling* com a renderização ligada.

Ao ligar a renderização, houve uma perda de desempenho de 52.4% quando comparamos o algoritmo de *frustum culling* híbrido com o melhor obtido em CPU. Esta queda de desempenho pode ser explicada pelo monitoramento do algoritmo para determinar se os cálculos devem ser feitos em CPU ou em GPU. Quando o algoritmo é executado em GPU ocorre um gargalo por ter muitas tarefas a serem executadas na placa gráfica e retorno dos dados da GPU para CPU. A parte onde o *frustum culling* híbrido ganhou da melhor abordagem em CPU é explicada por ter poucas coisas para serem renderizadas

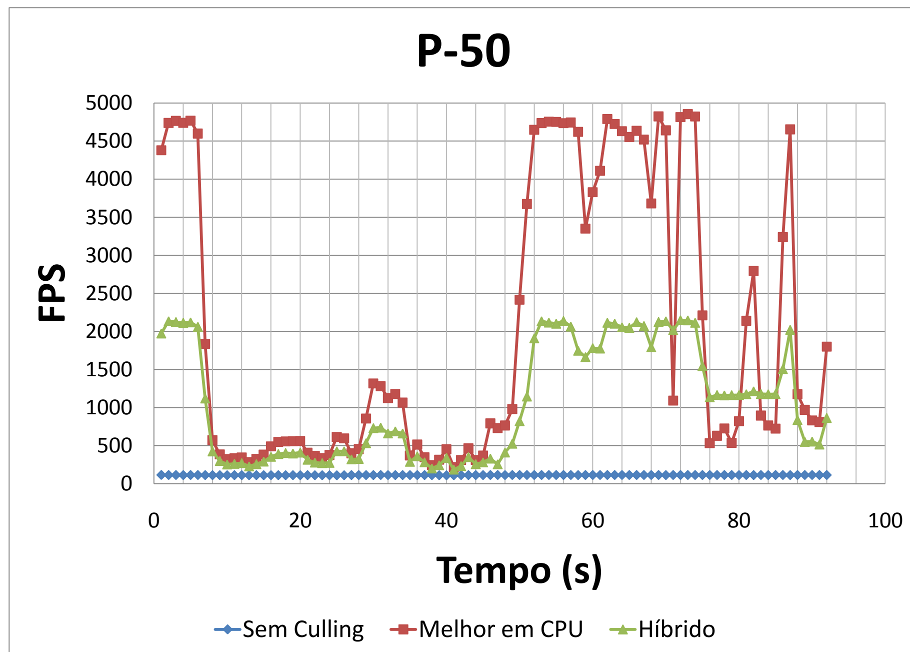


Figura 6.7: Resultados do frustum culling híbrido com render ligado.

e o algoritmo híbrido está rodando em GPU.

Mesmo não tendo o resultado esperado, o algoritmo de *frustum culling* híbrido talvez possa trazer ganhos no caso de várias placas gráficas, visto que ele não dividiria recursos com a renderização.

7
Conclusão e Trabalhos futuros

7.1
Conclusão

Este trabalho fez um levantamento das principais formas do algoritmo de *frustum culling*, indicando o seu estado da arte juntamente com técnicas de aceleração visando melhoria de performance. Como resultado, o algoritmo de extração dos planos do *frustum*, aliado à técnica de aceleração obteve melhor resultado. Na maioria dos testes as técnicas foram: utilização de hierarquia, percurso sem a utilização de pilha, *plane-coherency*, *octant-test* e teste com apenas dois vértices da AABB. Em apenas dois modelos (P-38 e P-50) o teste entre a AABB da câmera e o volume envolvente trouxe ganho de performance.

Com a inserção de hierarquia, as técnicas de testes entre a AABB da câmera e os volumes, *plane-coherency*, *octant test*, hierarquia e percurso sem a utilização de pilha compõem o estado da arte do algoritmo de *frustum culling* em CPU. A partir deste algoritmo foram exploradas novas formas de descarte implementadas em GPU. A motivação para tal foi o grande poder de processamento das placas de vídeo e os seus estágios cada vez mais configuráveis.

O *frustum culling* em GPU foi explorado em cima de dois tipos de modelos: as *GPU primitives* e os modelos com malha triangular. Sem a utilização de hierarquia, os desempenhos dos algoritmos em GPU, para os dois tipos de modelos, sem nenhuma técnica de aceleração obtiveram performance muito superior ao implementado apenas em CPU sem hierarquia. O melhor algoritmo implementado para as *GPU primitives* teve ganho de três vezes em média, enquanto para os modelos genéricos o ganho chegou a dez vezes. Porém, quando a hierarquia é adicionada ao algoritmo em CPU, esse algoritmo supera o da GPU.

A fim de unir os dois, CPU e GPU para executar o algoritmo de *frustum culling*, surgiu o *frustum culling* híbrido. Basicamente a ideia foi utilizar cada um deles onde apresenta melhor desempenho, aumentando assim a performance do algoritmo em geral. A melhor implementação conseguida

envolveu a chamada da GPU quando o tempo de processamento da CPU excede o tempo de *download* de informações (gargalo do algoritmo em GPU). O *frustum culling* híbrido contribuiu para um ganho de performance em todos os testes, sendo mais evidente no modelo do Boeing onde a melhoria foi de 76.75% na performance quando comparado com o melhor algoritmo implementado somente em CPU.

Com o ganho de performance alcançado na maioria dos casos de testes apresentados, os objetivos desta dissertação foram alcançados, deixando alguns assuntos para serem trabalhados no futuro, conforme visto na próxima seção.

7.2 Trabalhos futuros

O algoritmo de radar pode gerar falsos positivos, ou seja, volumes envolventes que estão fora do *frustum* são tratados como visíveis. Uma análise mais detalhada desse caso pode ser feita comparando com os outros algoritmos.

Os algoritmos multiprocessados implementados não corresponderam ao esperado na teoria, principalmente o que utiliza *tasks*. Um estudo melhor sobre essa arquitetura pode melhorar o algoritmo em CPU. Outra possibilidade de analise seria a utilização de POSIX Threads [55].

Um dos grandes problemas dos algoritmos de *frustum culling* implementados em GPU é a necessidade de guardar muitos volumes envolventes na memória da placa gráfica, o que pode se tornar um problema. Para as *GPU primitives*, os volumes envolventes podem ser determinados *on-the-fly*, minimizando assim o uso da memória da GPU. Quando há hierarquia, esse problema é agravado uma vez que o número de volumes envolventes aumenta. As AABBs necessitam de 32 *bytes*, para guardar estas informações além do seu id e o *escape index* do nó, onde são necessários dois *texels*. Uma possível solução para diminuir a memória utilizada é a codificação dos nós da hierarquia utilizando quatro ou oito *bytes*, um *texel*, desenvolvido por Mahovsky *et al.* [41].

Outra linha que pode ser seguida é a melhoria do algoritmo de descarte proposto por Reshetov [58], para dar suporte a seis planos e fazer a determinação de colisão entre o *frustum* e o volume envolvente, testando o algoritmo em CPU e GPU.

Testes podem ser feitos para acelerar o cálculo de descarte na GPU utilizando as técnicas de aceleração propostas e implementadas na CPU.

Uma questão ainda em aberto é determinar uma heurística ideal de entrada, permanência e saída da GPU para modelos diversos.

Com a evolução de múltiplas placas gráficas, seria interessante destinar uma placa gráfica para a renderização e outra para os algoritmos de *frustum*

culling, tentando evitar que o algoritmo de *frustum culling* em GPU dispute recursos com a renderização.

Por fim, a estrutura híbrida implementada nesta dissertação pode ser testada em outros algoritmos como, por exemplo, na detecção de colisão, visando ganho de performance.

Referências Bibliográficas

[1] AIREY, J.; ROHLF, J. ; BROOKS JR, F.. **Towards image realism with interactive update rates in complex virtual building environments**. In: PROCEEDINGS OF THE 1990 SYMPOSIUM ON INTERACTIVE 3D GRAPHICS, p. 41–50. ACM New York, NY, USA, 1990. 2.3

[2] APPEL, A.. **Some techniques for shading machine renderings of solids**. In: PROCEEDINGS OF THE APRIL 30–MAY 2, 1968, SPRING JOINT COMPUTER CONFERENCE, p. 37–45. ACM New York, NY, USA, 1968. 2.1.1

[3] ASSARSSON, U.; MOLLER, T.. **Optimized view frustum culling algorithms for bounding boxes**. JOURNAL OF GRAPHICS TOOLS, 5(1):9–22, 2000. 1.3, 3.2.4, 3.2.4, 4.5

[4] ASSARSSON, U.; STENSTRÖM, P.. **A case study of load distribution in parallel view frustum culling and collision detection**. In: EURO-PAR '01: PROCEEDINGS OF THE 7TH INTERNATIONAL EURO-PAR CONFERENCE MANCHESTER ON PARALLEL PROCESSING, p. 663–673, London, UK, 2001. Springer-Verlag. 1.3, 3.2.7, 4.7

[5] BAKER, D.; HEARN, M. P.. **Computer Graphics C Version**. Prentice Hall Press, New York, 1997. 2.3

[6] BARTZ, D.; MEISSNER, M. ; HUTTNER, T.. **Extending graphics hardware for occlusion queries in opengl**. In: HWWS '98: PROCEEDINGS OF THE ACM SIGGRAPH/EUROGRAPHICS WORKSHOP ON GRAPHICS HARDWARE, p. 97–ff., New York, NY, USA, 1998. ACM. 2.3

[7] BERNARDINI, F.; KLOSOWSKI, J. ; EL-SANA, J.. **Directional Discretized Occluders for Accelerated Occlusion Culling**. In: COMPUTER GRAPHICS FORUM, volume 19, p. 507–516. Blackwell Publishers Ltd, 2000. 2.3

[8] BITTNER, J.; HAVRAN, V. ; SLAVIK, P.. **Hierarchical visibility culling with occlusion trees**. In: COMPUTER GRAPHICS INTERNATIONAL, 1998. PROCEEDINGS, p. 207–219, 1998. 2.3

[9] BLINN, J.. **A trip down the graphics pipeline: the homogeneous perspective transform**. Computer Graphics and Applications, IEEE, 13(3):75–80, 1993. 2.1.2

[10] BRUDERLIN, B.; HEYER, M. ; PFUTZNER, S.. **Interviews3d: A platform for interactive handling of massive data sets**. IEEE COMPUTER GRAPHICS AND APPLICATIONS, p. 48–59, 2007. 3.3

[11] BUSS, S. R.. **3D Computer Graphics: A Mathematical Introduction with OpenGL**. Cambridge University Press, New York, 2003. 3

[12] CATMULL, E. E.. **A Subdivision Algorithm for Computer Display of Curved Surfaces**. PhD thesis, Dept. of CS, U. of Utah, December 1974. 2.1.1

[13] CLARK, J. H.. **Hierarchical geometric models for visible-surface algorithms**. In: SIGGRAPH '76: PROCEEDINGS OF THE 3RD ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, p. 267–267, New York, NY, USA, 1976. ACM. 3.2.3

[14] COHEN-OR, D.; CHRYSANTHOU, Y.; SILVA, C. ; DURAND, F.. **A survey of visibility for walkthrough applications**. IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS, p. 412–431, 2003. 2, 2.1.2, 2.2

[15] COORG, S.; TELLER, S.. **Temporally coherent conservative visibility**. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1995. 2.1.2, 2.3

[16] COZZI, P. J.. **Visibility driven out-of-core hlod rendering**. Master's thesis, University of Pennsylvania, 2008. 4.1

[17] DA SILVA, M. H.. **Tratamento eficiente de visibilidade através de Árvores de volumes envolventes**. Master's thesis, Pontifícia Universidade Católica do Rio de Janeiro (Departamento de Informática), Brazil, February 2002. 3.2.1

[18] DAMKJÆR, J.. **Stackless bvh collision detection for physical simulation**, 2007. http://image.diku.dk/projects/media/jesper.damkjaer.07.pdf, visitado em 12/01/09. 3.2.5

[19] DE TOLEDO, R.; LÉVY, B.. **Visualization of industrial structures with implicit gpu primitives**. In: PROCEEDINGS OF THE 4TH INTERNATIONAL SYMPOSIUM ON ADVANCES IN VISUAL COMPUTING, p. 139–150, 2008. 5.2

[20] DIETRICH, A.; WALD, I. ; SLUSALLEK, P.. **Large-Scale CAD Model Visualization on a Scalable Shared-Memory Architecture**. In: Proceedings of 10th International Fall Workshop - Vision, Modeling, and Visualization (VMV), p. 303–310, 2005. 1.1

[21] DUNN, F.; PARBERRY, I.. **3D Math primer for Graphics and Game Development**. Wordware Publishing Inc, 2002. 3.2.4

[22] DURAND, F.; DRETTAKIS, G.; THOLLOT, J. ; PUECH, C.. **Conservative visibility preprocessing using extended projections**. In: PROCEEDINGS OF THE 27TH ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, p. 239–248. ACM Press/Addison-Wesley Publishing Co. New York, NY, USA, 2000. 2.3

[23] ERICSON, C.. **Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology)**. Morgan Kaufmann, San Francisco, 2004. (document), 3.3

[24] FLYNN, M.. **Computer Architecture: Pipelined and Parallel Processor Design**. Jones and Bartlett Publishers, Inc., USA, 1995. 3.2.6

[25] FUCHS, H.; KEDEM, Z. ; NAYLOR, B.. **On visible surface generation by a priori tree structures**. In: PROCEEDINGS OF THE 7TH ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, p. 124–133. ACM Press New York, NY, USA, 1980. 2.1.1

[26] **Glsl**. http://www.clockworkcoders.com/oglsl/downloads.html, visitado em 23/01/09. 3.4

[27] GREENE, N.. **Detecting intersection of a rectangular solid and a convex polyhedron**. Academic Press Graphics Gems Series, p. 74–82, 1994. 3.2.4, 4.2

[28] GREENE, N.; KASS, M. ; MILLER, G.. **Hierarchical Z-buffer visibility**. In: PROCEEDINGS OF THE 20TH ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, p. 231–238. ACM New York, NY, USA, 1993. 2.3

[29] GRIBB, G.; HARTMANN, K.. **Fast extraction of viewing frustum planes from the world-view-projection matrix**, 2001. http://www2.ravensoft.com/users/ggribb/plane extraction.pdf, visitado em 22/01/09. 3.1

[30] HAINES, E.; WALLACE, J.. **Shaft culling for efficient ray-traced radiosity**. In: EUROGRAPHICS WORKSHOP ON RENDERING, May 1991. 3.2.4, 4.2

[31] HAUMONT, D.; DEBEIR, O. ; SILLION, F.. **Volumetric cell-and-portal generation**. In: COMPUTER GRAPHICS FORUM, volume 3-22 de **EUROGRAPHICS Conference Proceedings**. Blackwell Publishers, 2003. 2.3

[32] HUDSON, T.; MANOCHA, D.; COHEN, J.; LIN, M.; HOFF, K. ; ZHANG, H.. **Accelerated occlusion culling using shadow frusta**. In: SCG '97: PROCEEDINGS OF THE THIRTEENTH ANNUAL SYMPOSIUM ON COMPUTATIONAL GEOMETRY, p. 1–10, New York, NY, USA, 1997. ACM. 2.1.2, 2.3

[33] JAMES, D.; ANDRIES, V.; STEVEN, K. ; JOHN, F.. **Computer graphics: principles and practice**. Addison-Wesley, New York, 1990. 2.1.2

[34] KLOSOWSKI, J.; SILVA, C.; CENTER, I. ; HEIGHTS, Y.. **The prioritized-layered projection algorithm for visible setestimation**. IEEE transactions on visualization and computer graphics, 6(2):108–123, 2000. 2.3

[35] KLOSOWSKI, J.; SILVA, C.; CENTER, I. ; HEIGHTS, Y.. **Efficient conservative visibility culling using theprioritized-layered projection algorithm**. IEEE Transactions on Visualization and Computer Graphics, 7(4):365–379, 2001. 2.3

[36] KOLTUN, V.; CHRYSANTHOU, Y. ; COHEN-OR, D.. **Virtual occluders: An efficient intermediate pvs representation**. In: RENDERING TECHNIQUES 2000: 11TH EUROGRAPHICS WORKSHOP ON RENDERING, p. 59–70, 2000. 2.3

[37] KOLTUN, V.; CHRYSANTHOU, Y. ; COHEN-OR, D.. **Hardware-accelerated from-region visibility using a dual ray space**. In: RENDERING TECHNIQUES 2001: PROCEEDINGS OF THE EUROGRAPHICS WORKSHOP IN LONDON, UNITED KINGDOM, JUNE 25-27, 2001, p. 205. Springer Verlag Wien, 2001. 2.3

[38] LAINE, S.. **A general algorithm for output-sensitive visibility preprocessing**. In: I3D '05: PROCEEDINGS OF THE 2005 SYMPOSIUM ON INTERACTIVE 3D GRAPHICS AND GAMES, p. 31–40, New York, NY, USA, 2005. ACM. 2.3

[39] LEYVAND, T.; SORKINE, O. ; COHEN-OR, D.. **Ray space factorization for from-region visibility**. ACM Transactions on Graphics (TOG), 22(3):595–604, 2003. 2.3

[40] LUEBKE, D.; GEORGES, C.. **Portals and mirrors: simple, fast evaluation of potentially visible sets**. In: PROCEEDINGS OF THE 1995 SYMPOSIUM ON INTERACTIVE 3D GRAPHICS. ACM New York, NY, USA, 1995. 2.1.2, 2.3

[41] MAHOVSKY, J. A.. **Ray tracing with reduced-precision bounding volume hierarchies**. PhD thesis, University of Calgary, Calgary, Alta., Canada, Canada, 2005. 7.2

[42] **Microstation**. http://www.bentley.com/, visitado em 23/01/09. 3.4

[43] MILLINGTON, I.. **Game Physics Engine Development (The Morgan Kaufmann Series in Interactive 3D Technology)**. Morgan Kaufmann, San Francisco, 2007. 3.2.3

[44] MÖLLER, T.. **Real-time rendering**. A. K. Peters, Ltd., Natick, MA, USA, 2008. 3.2.3, 5.1.1

[45] MÖLLER, T.; HAINES, E.. **Real-time rendering**. A. K. Peters, Ltd., Natick, MA, USA, 1999. 2.1.2, 3.1, 3.2.1

[46] MORA, F.; AVENEAU, L. ; MERIAUX, M.. **Coherent and exact polygon-to-polygon visibility**. Proceedings of Winter School on Computer Graphics 2005, p. 87–94, 2005. 2.3

[47] MOREIRA, F.; COMBA, J. ; FREITAS, C.. **Smart visible sets for networked virtual environments**. In: COMPUTER GRAPHICS AND IMAGE PROCESSING, 2002. PROCEEDINGS. XV BRAZILIAN SYMPOSIUM ON, p. 373–380, 2002. 2.3

[48] NIELS THRANE, L. O. S.. **A comparison of acceleration structures for gpu assisted ray tracing**. Master's thesis, University of Aarhus, August 2005. 1.3, 3.2.5, 5.5

[49] NIRENSTEIN, S.. **Fast and accurate visibility preprocessing**. PhD thesis, University of Cape Town, 2003. 2.3

[50] **Obj**. Especificação do arquivo,http://local.wasp.uwa.edu.au/ pbourke/-dataformats/obj/, visitado em 29/01/09. 3.4

[51] OMOHUNDRO, S. M.. **Five Balltree Construction Algorithms**. Technical report, International Computer Science Institute, December 1989. 3.2.3

[52] **Openmp**. Open Multi-Processing,http://openmp.org/wp/, visitado em 29/01/09. 4.7

[53] PALLISTER, K.. **Game Programming Gems**, volume 5, p. 65–76. Charles River Media, Rockland, February 2005. 1.3, 3.2.2

[54] PONCE, J.; FAUGERAS, O.. **An object centered hierarchical representation for 3d objects: the prism tree**. Comput. Vision Graph. Image Process., 38(1):1–28, 1987. 3.2.1

[55] **Posix threads**. POSIX Threads,https://computing.llnl.gov/tutorials/pthreads/, visitado em 9/06/09. 7.2

[56] **libqglviewer**. http://www.libqglviewer.com/, visitado em 23/01/09. 3.4

[57] **Qt**. http://www.qtsoftware.com/, visitado em 23/01/09. 3.4

[58] RESHETOV, A.. **Apparatus and method for a frustum culling algorithm suitable for hardware implementation**, June 19 2008. US Patent App. 12/142,668. 3.2.4, 7.2

[59] ROBERTS, L.. **Machine perception of three-dimensional solids**. Technical report, Massachusetts Institute of Technology, 1963. 2.1

[60] SALOMON, D.. **Transformations and Projections in Computer Graphics**. Springer, New York, 2006. 3

[61] SCHAUFLER, G.; DORSEY, J.; DECORET, X. ; SILLION, F.. **Conservative volumetric visibility with occluder fusion**. In: PROCEEDINGS OF THE 27TH ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, p. 229–238. ACM Press/Addison-Wesley Publishing Co. New York, NY, USA, 2000. 2.3

[62] SCHMALSTIEG, D.; TOBLER, R. F.. **Fast projected area computation for three-dimensional bounding boxes**. J. Graph. Tools, 4(2):37–43, 1999. 2.1.2

[63] SLATER, M.; CHRYSANTHOU, Y.. **View volume culling using a probabilistic caching scheme**. In: PROCEEDINGS OF THE ACM SYMPOSIUM ON VIRTUAL REALITY SOFTWARE AND TECHNOLOGY, p. 71–77. ACM New York, NY, USA, 1997. 2.2

[64] ST-LAURENT, S.. **Shaders for Game Programmers and Artists (Premier Press Game Development)**. Course Technology Ptr, Cambridge, 2004. 2

[65] STEWART, J.. **An investigation of simd instruction sets**. http://noisymime.org/blogimages/SIMD.pdf, visitado em 22/01/09, 2005. 3.2.6

[66] SUNAR, M.; ZIN, A. ; SEMBOK, T.. **Range detection approach in interactive virtual heritage walkthrough**. In: PROCEEDINGS OF THE 16TH INTERNATIONAL CONFERENCE ON ARTIFICIAL REALITY AND TELEXISTENCE–WORKSHOPS, volume 0, p. 599–602, Los Alamitos, CA, USA, 2006. IEEE Computer Society. 4.2

[67] **Tecgraf**. Tecnologia em Computação Gráfica,http://www.tecgraf.puc-rio.br/, visitado em 29/01/09. 3.4

[68] TELLER, S.; SEQUIN, C.. **Visibility preprocessing for interactive walkthroughs**. In: PROCEEDINGS OF THE 18TH ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, p. 61–70. ACM New York, NY, USA, 1991. 2.3

[69] THOMAS H. CORMEN, CHARLES E. LEISERSON, R. R. L.. **Introduction to Algorithms**. The MIT Press, London, 1990. 1

[70] TOLEDO, R.. **Interactive Visualization of Massive Data using Programmable Graphics Cards**. PhD thesis, Loria, INRIA institute, 2007. (document), 1.3, 5.1.1, 5.2, 5.4, 5.5, 5.2, 5.2

[71] **Instruction latencies and throughput for amd and intel x86 processors**. http://gmplib.org/ tege/x86-timing.pdf, visitado em 03/08/09. 4.2

[72] VERTH, J.; BISHOP, L.. **Essential mathematics for games and interactive applications**. Morgan Kaufmann, San Francisco, 2004. 3.2.1

[73] WALD, I.. **Realtime ray tracing and interactive global illumination**. PhD thesis, Universitätsbibliothek, 2004. 1.1

[74] WATT, A. H.. **3D Computer Graphics (3rd Edition)**. Addison Wesley, Toronto, 1999. 3

[75] WONKA, P.; SCHMALSTIEG, D.. **Occluder shadows for fast walkthroughs of urban environments**. In: COMPUTER GRAPHICS FORUM, volume 18, p. 51–60. Blackwell Publishers Ltd, 1999. 2.3

[76] WONKA, P.; WIMMER, M. ; SCHMALSTIEG, D.. **Visibility preprocessing with occluder fusion for urban walkthroughs**. Rendering Techniques 2000, p. 71–82, 2000. 2.3

[77] ZHANG, H.; MANOCHA, D.; HUDSON, T. ; HOFF III, K.. **Visibility culling using hierarchical occlusion maps**. In: PROCEEDINGS OF THE 24TH ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, p. 77–88. ACM Press/Addison-Wesley Publishing Co. New York, NY, USA, 1997. 2.3