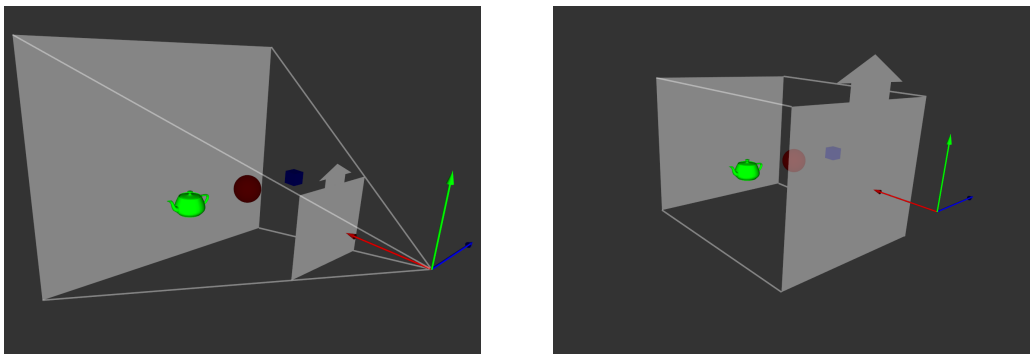
3
Frustum culling

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].

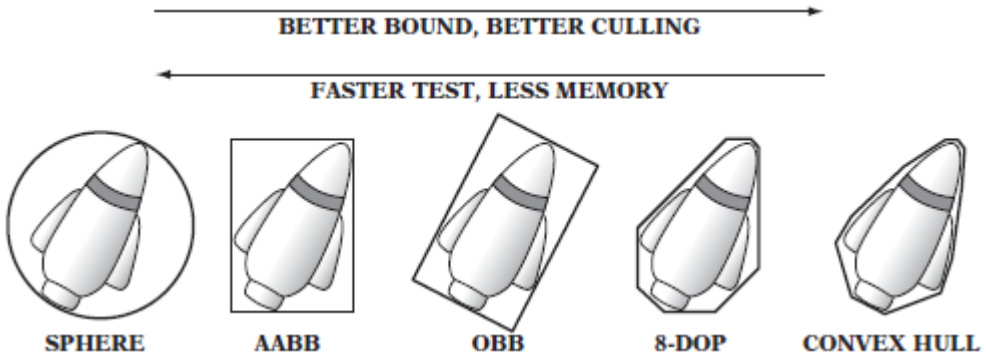


Figura 3.3: Volumes envolventes, imagem divulgada por Ericson *et al.* [23]

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,

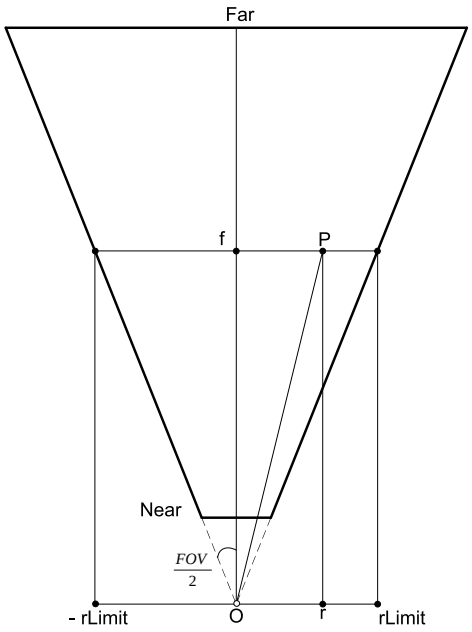


Figura 3.5: Classificação de um ponto P contra o frustum.

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.

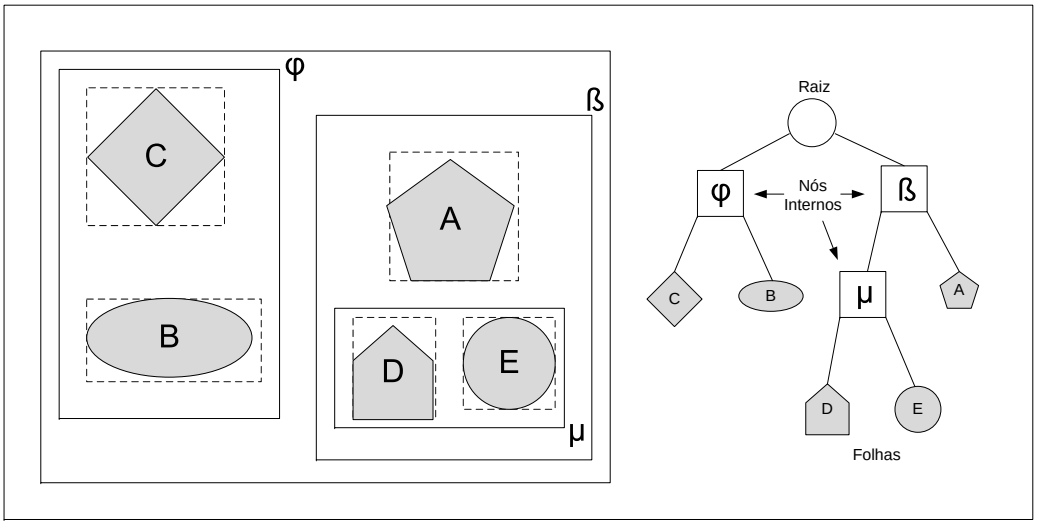


Figura 3.8: Hierarquia de volumes envolventes.

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

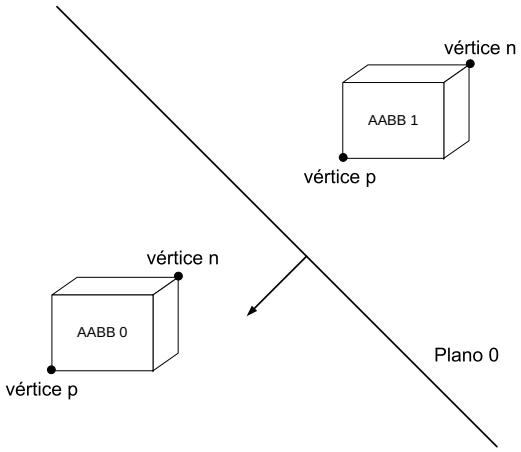


Figura 3.9: Classificação dos vértices n e p.

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.

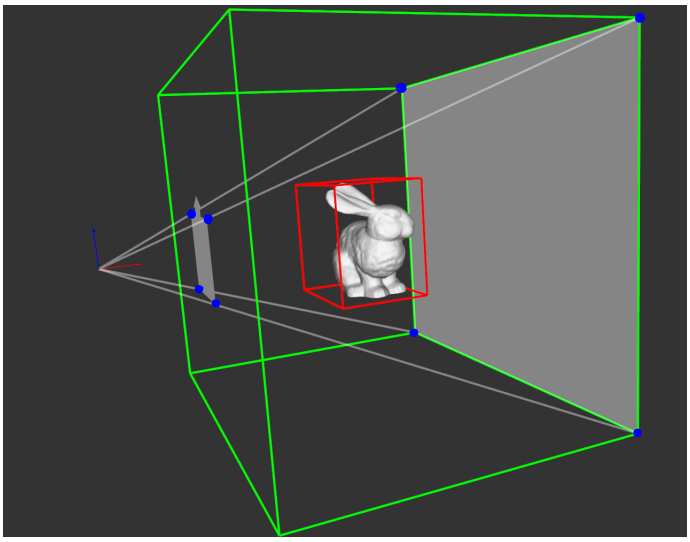


Figura 3.10: Volume envolvente do frustum.

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 X_n, Y_n e Z_n são referentes as normais dos planos e $X_{min/max}$, $Y_{min/max}$ e $Z_{min/max}$ aos valores mínimos e máximos da AABB.

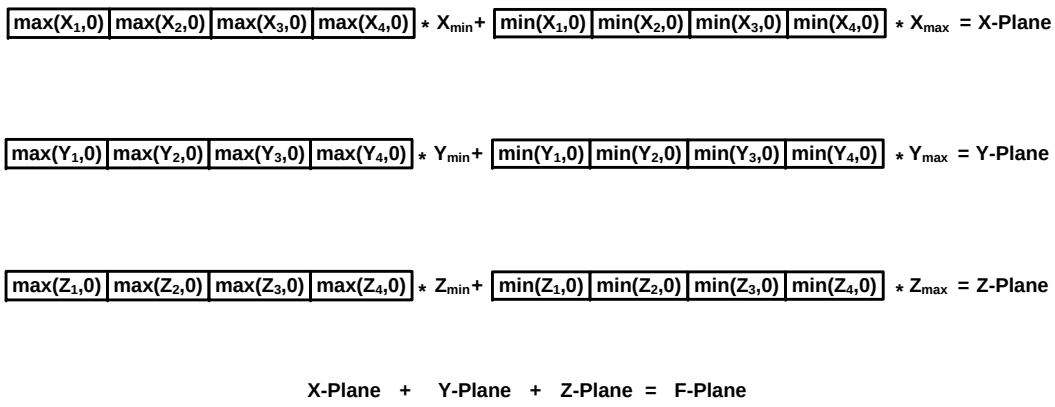


Figura 3.11: Teste de descarte com apenas quatro planos.

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.

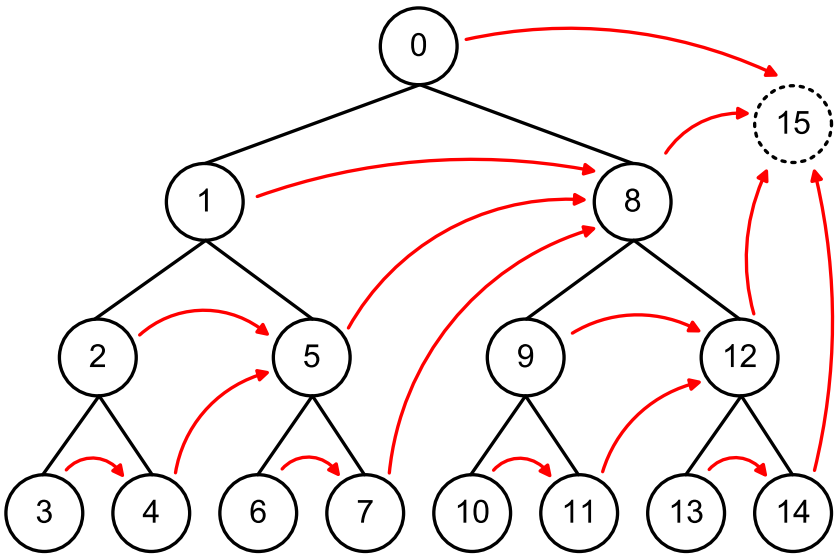


Figura 3.12: Escape index.

[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

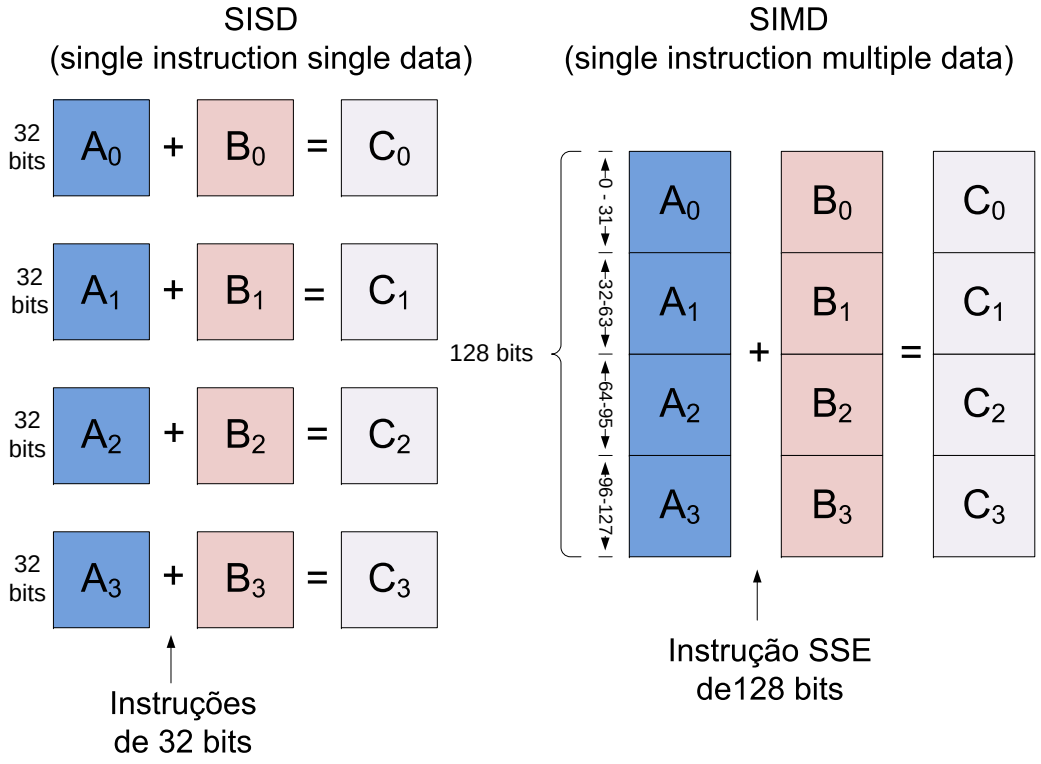


Figura 3.13: SISD X SIMD.

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.

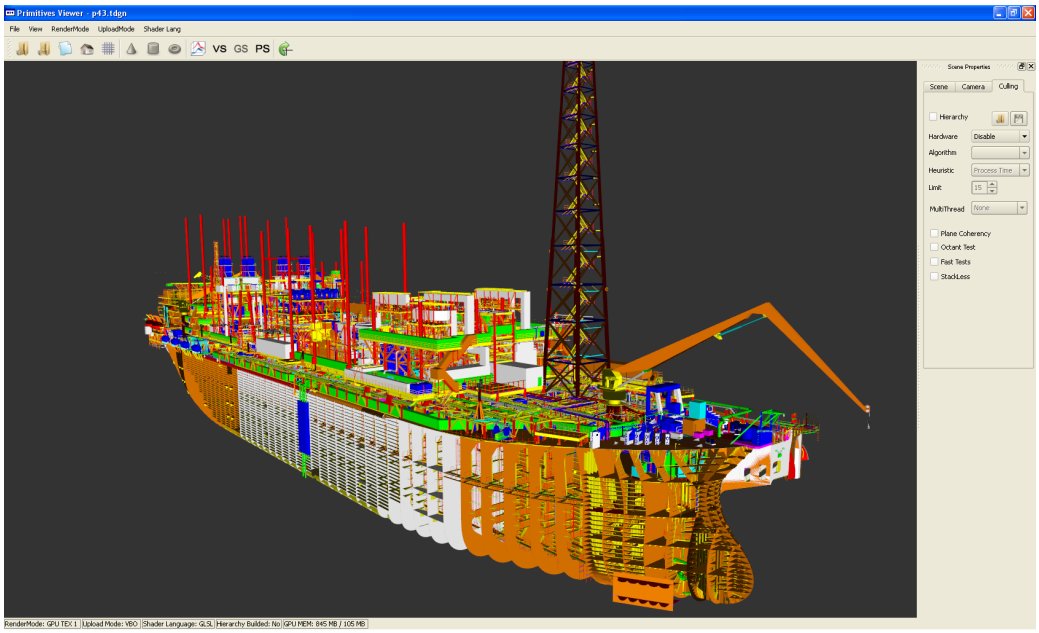


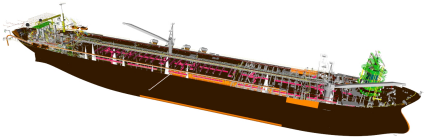
Figura 3.15: Aplicação desenvolvida para os testes.

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.

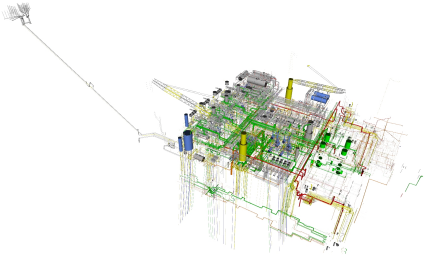
Os passos para a renderização dos dados paramétricos de forma eficiente podem ser encontrados na Seção 5.2. Além dos objetos paramétricos, este arquivo também contém dados de malha. A Tabela 3.1 traz as principais informações sobre os modelos TDGN utilizados como testes.

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

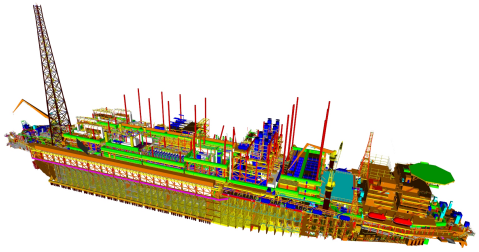
[4]shaders - programas desenvolvidos para serem rodados diretamente na placa gráfica.



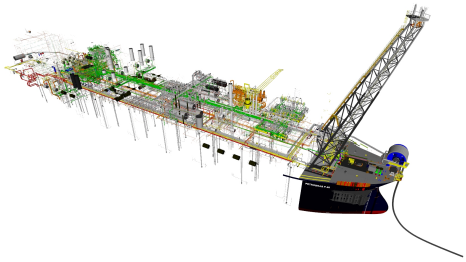
3.16(a): P-38.



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.

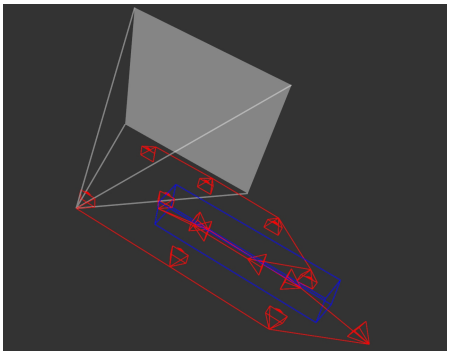
Para fazer os testes dos algoritmos foram construídos caminhos de câmera ao longo dos modelos como pode ser visto na Figura 3.17, onde a caixa azul representa o modelo, a linha vermelha o caminho de câmera feito, a representação de câmera em vermelho ilustra alguns dos *frames* chave para realização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

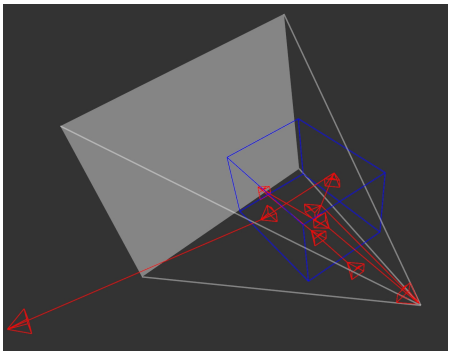
Todos os testes realizados nesta dissertação foram executados três vezes e utilizou-se a média, a fim de obter um valor mais próximo da realidade.

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

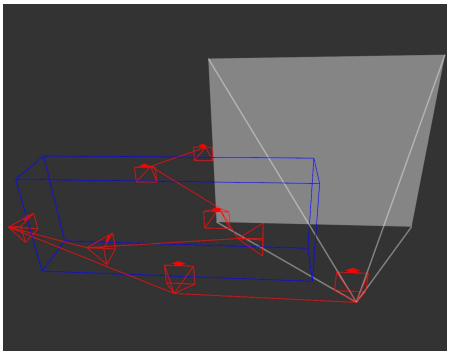
Tabela 3.2: Modelos com malhas triangulares.



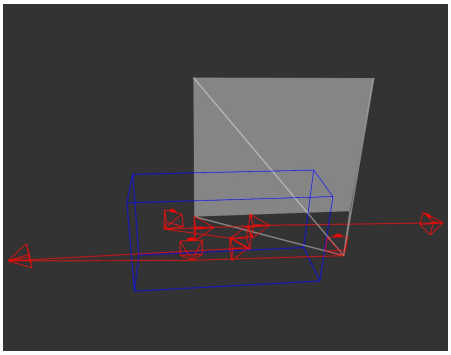
3.17(a): P-38 (80 segundos).



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.

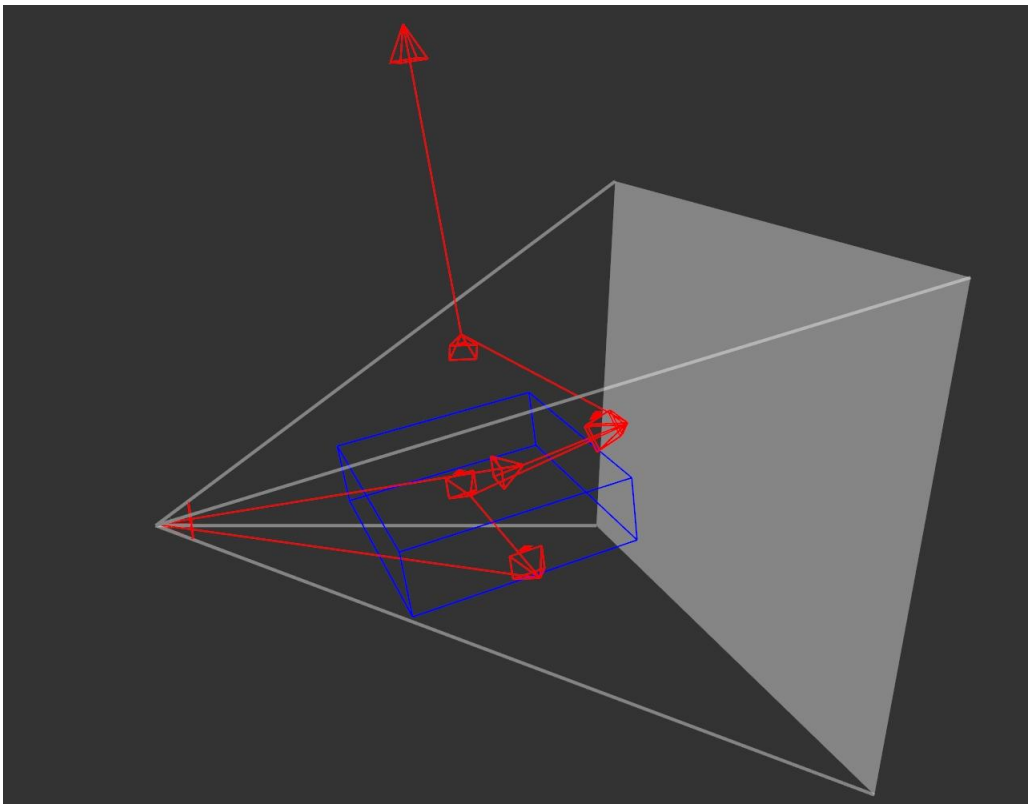


Figura 3.18: Caminho de câmera pelo Boeing (73 segundos).