

Frustum Culling Híbrido em CPU e GPU para Visualização de Modelos Massivos em Tempo Real

Eduardo Telles Carlos
Tecgraf / DI / PUC-Rio
R. M. S. Vicente, 225
Rio de Janeiro, RJ, Brasil
etc@tecgraf.puc-rio.br

Alberto Raposo
Tecgraf / DI / PUC-Rio
R. M. S. Vicente, 225
Rio de Janeiro, RJ, Brasil
abraposo@tecgraf.puc-rio.br

Luciano Soares
Tecgraf / DI / PUC-Rio
R. M. S. Vicente, 225
Rio de Janeiro, RJ, Brasil
lpsoares@tecgraf.puc-rio.br

Resumo

Frustum culling é uma das principais técnicas utilizadas para a redução da complexidade geométrica de modelos 3D para sua visualização. Com o advento da GPGPU (General-Purpose computation on Graphics Processing) se abrem novas possibilidades de técnicas de frustum culling. A proposta desenvolvida nesta pesquisa é uma nova técnica que utiliza o poder computacional da GPU, resultando em um algoritmo híbrido que aproveitará o melhor da CPU e da GPU, tirando melhor proveito do tempo ocioso. Diversos algoritmos clássicos de frustum culling foram analisados e comparados com o algoritmo proposto, mostrando a eficiência do método proposto.

Abstract

Frustum culling is one of the main techniques used to reduce the geometric complexity of 3D models for visualization. GPGPU (General-Purpose computation on Graphics Processing) offers possibilities for new frustum culling techniques. The proposal of this research is a new technique that uses the GPU computational resources, resulting in a hybrid algorithm that takes advantage of best of CPU and GPU, better using idle time. Some classical frustum culling algorithms were analyzed and compared with the proposed method, showing its efficiency.

1. Introdução

Um dos problemas mais antigos da computação gráfica é a determinação de visibilidade. Inúmeros algoritmos foram desenvolvidos para viabilizar modelos cada vez mais detalhados em sistemas computacionais com recursos limitados. Dentre estes algoritmos, destaca-se o *frustum culling*, cujo papel é não processar objetos que não sejam visíveis

ao observador. Esse conjunto de algoritmos, muito comum em várias aplicações gráficas, vem sofrendo melhorias ao longo dos anos, a fim de otimizar 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. Com relação aos modelos massivos, necessita-se de algoritmos de alto desempenho, pois a quantidade de cálculos aumenta significativamente.

Diferentemente das CPUs (*Central Processing Units*), as GPUs (*Graphics Processing Units*) são processadores originalmente dedicados para cálculos gráficos e são altamente paralelizados. Nos últimos anos o poder de processamento das placas gráficas sofreu grande aumento, superando a *Lei de Moore*.

Atualmente as CPUs estão se paralelizando para conseguir aumentar seu poder de processamento. Porém 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*¹ das CPUs e GPU fornece uma ideia do poder de processamento das GPUS. Enquanto uma CPU Core 2 Extreme QX9650 chega a 48 *gigaflops*, uma GPU GeForce GTX 280 consegue 933 *gigaflops*.

A cada nova geração de processadores gráficos usados nas GPUs, novos recursos são incorporados aumentando o seu desempenho e tornando viável o processamento de cenas cada vez mais complexas.

A partir de 2001 as GPUs sofreram uma grande mudança, tornando alguns de seus estágios programáveis (antes tais estágios eram implementados diretamente em *hardware* não podendo ser reprogramados). Com isso, vários algoritmos novos foram desenvolvidos em diversas áreas. A *GPGPU (General-purpose computing on graphics*

¹*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.

processing units) é uma área recente que surgiu devido ao acesso dos estágios do *pipeline* das placas gráficas.

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 são desenvolvidos acompanhando 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 reduzem o nível de detalhes dos objetos gradativamente à medida que os objetos ficam mais afastados 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.

O primeiro objetivo deste pesquisa foi fazer uma investigação do estado da arte dos algoritmos 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 desenvolvimentos deste tipo de algoritmo em CPU e as GPUs estão se tornando cada vez mais poderosas e acessíveis para programação, um segundo objetivo desta pesquisa é desenvolver esse tipo de algoritmo em GPU e compará-lo com o que foi obtido em CPU, analisando os pontos positivos e negativos.

Outro objetivo é desenvolver o descarte de objetos paramétricos diretamente em GPU. Tais objetos são conhecidos como *gpu primitives* [17]. 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 processadas diretamente na GPU, o cálculo de descarte será inserido neste *pipeline* de duas maneiras diferentes e comparado com a implementação da CPU.

Como principal resultado desta pesquisa está o desenvolvimento de um *frustum culling* híbrido utilizando a GPU em momentos oportunos para determinar quais os objetos estão visíveis pela câmera em conjunto com a CPU. O algoritmo de *frustum culling* quando tratado apenas em CPU com modelos massivos, pode apresentar gargalos diminuindo consideravelmente o desempenho da aplicação. Esta parte do trabalho tentará identificar os momentos oportunos para uso da GPU e melhorar o desempenho para que a interação não seja afetada. Mais detalhes serão apresentados na Seção 5.

2. Trabalhos relacionados

Os trabalhos relacionados a este artigo 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

o trabalho desenvolvido por Assarsson e Moller, que desenvolveram algoritmos que aceleram os cálculos de descarte [1] e técnicas de *frustum culling* em máquinas multiprocessadas. Thrane [11] removeu a necessidade da utilização de uma pilha para realizar o percurso da hierarquia. Mais recentemente [13] implementou uma nova forma de fazer descarte de objetos sem a necessidade de fazer extração de matrizes e cálculo de planos a cada quadro (*frame*). Além dos trabalhos citados, existem outros trabalhos relacionados a este artigo que serão citados e discutidos com mais detalhes na Seção 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 reaproveitada. Um exemplo é o trabalho de Thrane [11], que desenvolveu um percurso de hierarquia em GPU a priori para acelerar algoritmos de *ray tracing*.

3. Frustum Culling em CPU

O objetivo do algoritmo de *frustum culling* é descartar a maior parte dos objetos que estão fora do volume de visão, evitando que dados geométricos sejam processados desnecessariamente. O teste de descarte pode resultar em três situações:

1. Objeto totalmente dentro do volume de visão;
2. Objeto interceptando o volume de visão;
3. Objeto totalmente fora do volume de visão.

De acordo com [10], 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.

Uma das maneiras de identificar se um polígono está visível no volume de visão é calculando os seis planos que o compõem (*near, far, left, right, top, down*) a cada quadro que a câmera muda a posição ou orientação. Os cálculos para obtenção dos planos utilizando *OpenGL* e *DirectX* podem ser encontrados em [7]. 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:

$A * p_x + B * p_y + C * p_z + D \leq 0$, onde (p_x, p_y, p_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.

Para que um objeto qualquer seja considerado não visível, todos os seus vértices devem estar fora do volume de visão. 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*). O princípio desta técnica 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 outros tipos de volumes envolventes como cilindros, prismas [14], elipsóides, *k-dops*, entre outros. Elevando a complexidade do volume envolvente, o ajuste com a geometria do objeto melhora, porém aumenta o custo dos testes [3, 18].

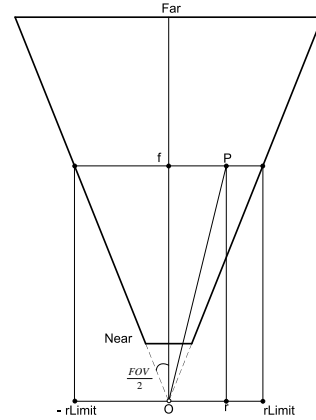
Fazer testes utilizando volumes envolventes ao invés da própria geometria pode ser vantajoso. Porém ainda é necessário testar todos os volumes envolventes individualmente para determinar se estão visíveis, podendo transformar-se no gargalo da aplicação dependendo da cena. Clark [2] utilizou a ideia de divisão do espaço 3-D em uma árvore para determinar a visibilidade de superfícies. Neste trabalho, a construção das hierarquias utilizou AABB (*Axis-Aligned Bounding Box*) 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 nos testes realizados.

3.1 Otimizações

Nesta seção serão levantadas várias otimizações em cima do algoritmo de *frustum culling* visando obter seu estado da arte em CPU.

A primeira otimização se refere ao modo pelo qual são feitos os cálculos para determinação de visibilidade. 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.* [13], que ficou conhecida como radar. Mais conservativo que o algoritmo que utiliza planos, o princípio 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 de corte próximo e distante (*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. Esses valores só precisam ser inicializados na construção da câmera ou quando forem modificados, o que não é muito comum. A Figura 1 ilustra a classificação de um ponto *P* contra um sistema de radar em 2-D.



$$r factor = \tan\left(\frac{FOV}{2}\right) = \frac{ladooposto}{ladoadjacente} = \frac{rLimit}{f}$$

$$r factor = \frac{rLimit}{f}$$

$$rLimit = r factor * f$$

Figura 1. Classificação de um ponto P contra o frustum.

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.

Outra otimização 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 [8, 6]. 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 2 ilustra a identificação dos vértices *n* e *p* em dois casos.

Caso a distância entre o vértice *n* e plano for positiva, 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 a caixa está interceptando o plano, o contrário acontece: a distância do plano para o vértice *n* é negativa e para o vértice

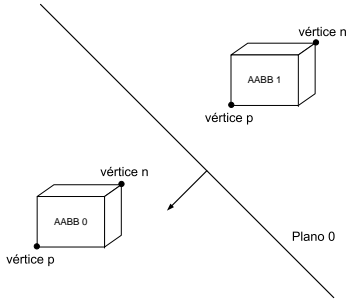


Figura 2. Classificação dos vértices n e p.

p é positiva. Esta otimização só pode ser feita em AABs [1].

Assarsson e Möller [1] 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 seu centro. Uma vez identificada a sua localização é suficiente testar apenas os três planos externos para determinar se a esfera envolvente está visível. O mesmo princípio pode ser aplicado a outros volumes envolventes [1].
- *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.

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

- *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 complexos 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 mostra a AABB 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 [4].

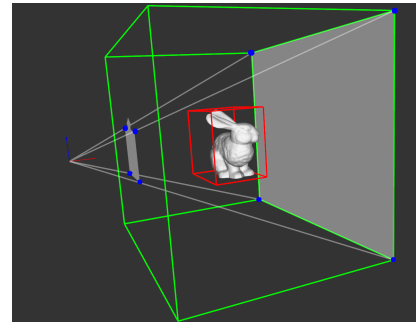


Figura 3. Volume envolvente do frustum.

O conjunto de otimizações utilizadas para determinar o melhor algoritmo conseguido em CPU, assim como os em GPU, serão apresentados na Seção 6.

4. Frustum Culling em GPU

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

4.1 Frustum culling nas GPU primitives

GPU primitives [17] é 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 no *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.

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 no bom desempenho do teste contra apenas dois vértices da AABB feito em CPU. As etapas dos cálculos executados pela GPU podem ser vistas na Figura 4.

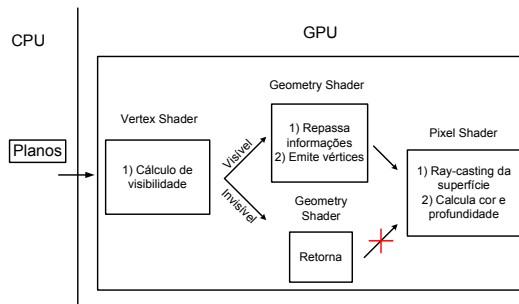


Figura 4. Frustum culling com GPU primitives.

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.

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 síntese das primitivas. A Figura 5 mostra o esquema do algoritmo.

Primeiramente o *shader* que vai realizar os cálculos de

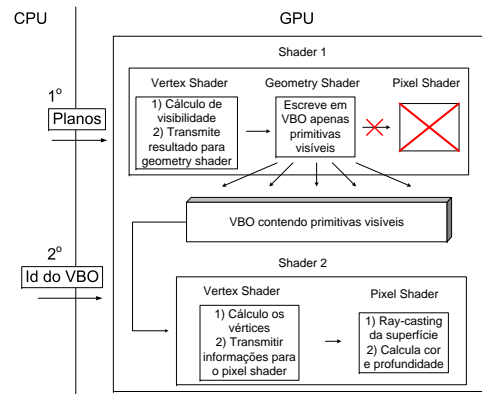


Figura 5. Frustum culling em dois shaders.

frustum culling, representado pelo número 1 na Figura 5, 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.

4.2 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. Essa limitação também ocorre no caso das *gpu primitives*, porém no máximo quatorze vértices precisam ser emitidos, no caso mais complexo 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 quadro é muito custoso. Um possível esquema de tratamento de modelos genéricos é visto na Figura 6.

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 a quantidade de resultados que serão levados para a CPU.

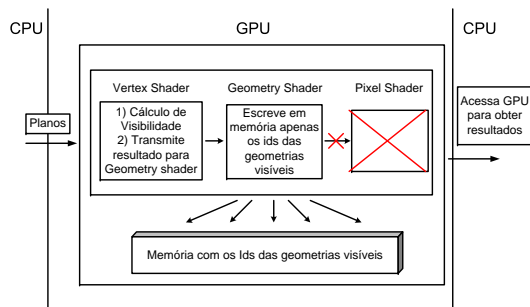


Figura 6. Frustum culling em modelos genéricos usando o geometry shader.

Outra possível abordagem é eliminar o estágio de geometria e escrever todos os resultados (volumes envolventes visíveis e não visíveis diferenciados por sinal) na memória da GPU e depois, em CPU, separar os volumes visíveis. Este esquema está ilustrado na Figura 7.

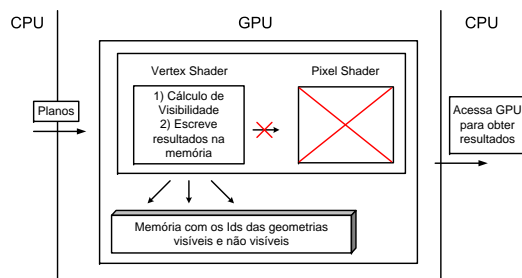


Figura 7. Frustum culling em modelos genéricos sem utilizar o geometry shader.

5. 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 trivial e em alguns casos não produz melhores resultados. Este trabalho visou até agora mostrar o estado da arte do *frustum culling* em CPU e maneiras de torná-lo portátil para a GPU. Esta seção propõe algumas heurísticas para determinar o momento ideal, onde a CPU possui pior desempenho, para utilização da GPU a fim de acelerar o algoritmo de determinação dos objetos visíveis.

O estado da arte do algoritmo de *frustum culling* em CPU obtém boa performance em cenas pequenas e médias como será visto na Seção 6. 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, já 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 bom desempenho 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 a ponto 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.

5.1 Identificação do momento ideal

A identificação do momento para a utilização da GPU é crucial para melhorar o desempenho da aplicação, pois se não for bem escolhido pode diminuir a velocidade da aplicação. 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. Na medida em que os cálculos vão crescendo, a única saída da CPU para acelerar o desempenho 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.

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.

Tendo os momentos de transição dos cálculos da CPU para a GPU identificados, falta converter o algoritmo para o formato vetorial a partir dos resultados obtidos na CPU e tirar proveito do poder de processamento da GPU.

5.2 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 o desempenho 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 proposta é processar apenas uma parte dos nós em paralelo, uma vez que se todos os nós forem processados, o desempenho global vai diminuir. 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 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 poder gerar 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: se em um determinado quadro 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 8 mostra o esquema do algoritmo.

Na Figura 8 os nós 2, 6, 7 e 10 foram os últimos nós classificados como visíveis, ou seja, suas geometrias participam

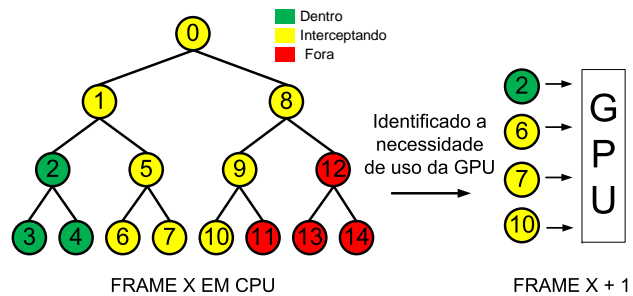


Figura 8. Esquema do algoritmo de frustum culling híbrido.

da imagem final. Uma vez que os nós estejam sendo processados em GPU, os resultados emitidos se tornarão a entrada do quadro seguinte. Desta forma é mantida a coerência temporal entre quadro, diminuindo 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 negativo, ou seja, fora do volume de visão, o nó pai correspondente deve ser adicionado como *input* de processamento no próximo quadro. Por exemplo, se no quadro x o nó 10 for processado em GPU e seu resultado der fora do volume de visão, no quadro $x + 1$, 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 quadro e nos quadros 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 (Figura 9).

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 quadros 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.

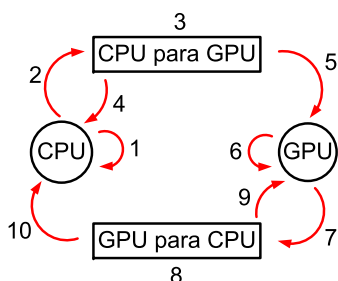


Figura 9. Possíveis estados do frustum culling híbrido.

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 quadros 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 quadro 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.

6. Resultados

Para a realização dos testes foi utilizada uma máquina 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 GLSL³ para os *shaders*. Também foram utilizadas bibliotecas auxiliares como OpenGL para a renderização, Qt 4.4.1 [16] para a interface gráfica, libQGLViewer 2.3.1 [15] para auxiliar o desenvolvimento do visualizador e libglsl 1.0.0 [5] para os *shaders*.

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

³GLSL (OpenGL Shading Language) - linguagem de programação de alto nível criada pela OpenGL ARB (Architecture Review Board) destinada a programação do *pipeline* das placas gráficas.

um formato que faz a ligação entre ferramentas CAD e os módulos de visualização 3D desenvolvido no Tecgraf / PUC-Rio. Seus dados são provenientes de partes do arquivo DGN (DesiGN file) lido na íntegra pelo programa MicroStation [9]. A Figura 10 ilustra parte de algumas plataformas de petróleo utilizadas nos testes.

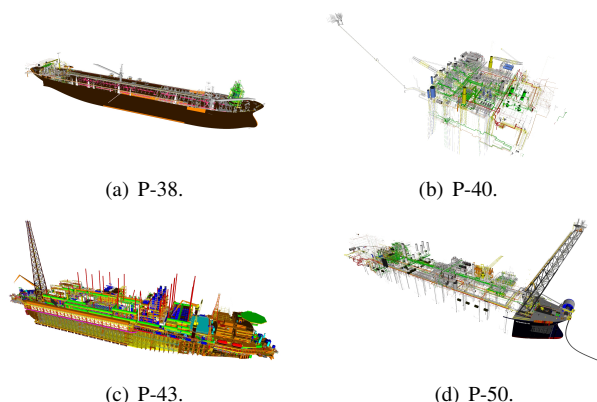


Figura 10. Modelos paramétricos.

O formato utilizado nos modelos com malhas poligonais foi o OBJ [12]. Desenvolvido pela Wavefront Technologies, este formato foi escolhido por ser de simples interpretação e por um dos modelos utilizados neste trabalho, Boeing 777, já estar neste formato. A Tabela 1 traz informações dos modelos utilizados.

Modelos	Objetos	Tamanho
P-38	13×10^4	1.94 GB
P-40	26×10^4	1.34 GB
P-43	28×10^4	1.02 GB
P-50	71×10^4	1.14 GB
Boeing 777	71×10^4	14.00 GB

Tabela 1. Informações dos modelos de teste.

Para fazer os testes dos algoritmos foram construídos caminhos de câmera ao longo dos modelos tentando explorar o maior número de situações possíveis em uma interação, alternando o número de colisões entre o *frustum* e os volumes envolventes. 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.

O esquema do melhor algoritmo de *frustum culling* implementado em CPU para a maioria dos modelos, pode ser visto na Figura 11.

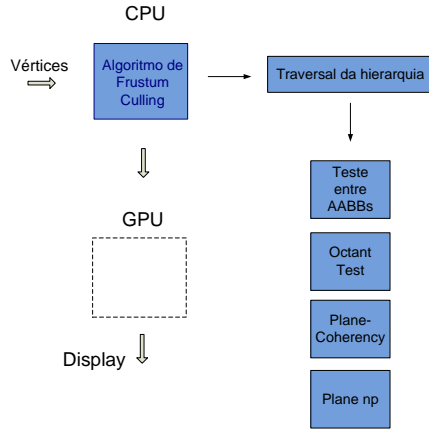


Figura 11. Pipeline final em CPU.

Em todos os casos de testes foi utilizada a mesma hierarquia de volumes envolventes para cada modelo intercalando as otimizações buscando sempre o maior desempenho. A primeira otimização da Figura 11 converte o *frustum* de visão em uma AABB e realiza testes entre duas AABB para determinar se o objeto está 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. As otimizações de *octant test* e *plane-coherency*, desenvolvidas por Assarsson e Möller [1], também aumentaram o desempenho do algoritmo e as outras duas, *masking* e *TR coherency* não se mostraram vantajosas na maioria dos casos. Caso não ocorra descarte em nenhuma dessas otimizações é feito o teste utilizando apenas os dois vértices do volume envolvente (*plane-np*).

A comparação entre o melhor algoritmo conseguido em CPU com os desenvolvidos para GPU, foram eliminadas as hierarquias de ambos, porém mantendo as otimizações na CPU. Na GPU foi implementado apenas o descarte utilizando a otimização de *plane n-p* para os modelos paramétricos e o com malha triangular. A Tabela 2 mostra a comparação de médias de *fps* conseguidas em cada modelo.

Método	P-38	P-40	P-43	P-50	Boeing
Melhor CPU	237	61	32	27	36
GPU c/ Geom. Shader (fig.6)	1350	481	273	362	240
GPU s/ Geom. Shader (fig.7)	1233	435	231	306	241

Tabela 2. Algoritmos em CPU e GPU

A ampla vantagem do algoritmo puramente em GPU

frente ao melhor 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. Para o modelo contendo malha triangular (Boeing 777), os resultados foram bem próximos. O gargalo das duas aplicações se encontra na necessidade de trazer os resultados da GPU para a CPU.

Para os modelos paramétricos, foram propostos dois algoritmos diferentes quanto ao número de *shaders* a serem utilizados. A primeira executa os cálculos no mesmo *shader* de renderização das *GPU primitives* e a outra utiliza um *shader* a parte. Como pode ser visto na Tabela 3, o desempenho melhora quando a técnica do *shader* separado é utilizada.

Método	P-38	P-40	P-43	P-50
1 shader (fig. 4)	351	137	106	89
2 shaders (fig. 5)	934	361	170	133

Tabela 3. Algoritmos em CPU e GPU

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.

Para que o *frustum culling* híbrido funcione bem, os gargalos em CPU e o momento de saída da GPU devem ser bem estimados. Segue abaixo o resultado da utilização de cada uma das heurísticas propostas anteriormente.

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.

A Tabela 4 compara o melhor algoritmo conseguido em CPU com a hierarquia com a abordagem híbrida, utilizado a heurística 4 acima.

Método	FPS Mínimo	FPS Máximo	Média do Caminho
CPU na P-38	2368	7241	5494
Híbrido na P-38	2756	7301	5615
CPU na P-40	320	7263	3143
Híbrido na P-40	596	7266	3490
CPU na P-43	358	7334	3265
Híbrido na P-43	527	7270	3627
CPU na P-50	311	7256	3022
Híbrido na P-50	578	7241	3505
CPU no Boeing	96	7259	2605
Híbrido no Boeing	280	7300	4629

Tabela 4. Comparação dos resultados entre melhor algoritmo em CPU e Híbrido.

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, o que indica sua viabilidade para modelos mais complexos.

7. Conclusão

Sistemas híbridos de *frustum culling* apresentam uma melhoria significativa em relação às técnicas convencionais. Embora recursos da GPU que seriam usados para o processamento da imagem sejam comprometidos para o cálculo do *frustum culling* o ganho de velocidade pelo descarte de geometrias supera as perdas em cenas complexas. Já com o sistema híbrido, este consumo de CPU e GPU é automaticamente calibrado para a complexidade da cena, fazendo o algoritmo ser adaptativo para a cena que se esteja visualizando independente do ponto de vista.

Uma possível forma de contornar as limitações de *output* do estágio de *geometry shader* da placa gráfica seria escrever os resultados em textura no estágio de *pixel shader*. Uma questão ainda em aberto é determinar uma heurística ideal de entrada, permanência e saída da GPU para modelos diversos.

A estrutura híbrida implementada poderá ser testada em outros algoritmos como, por exemplo, na detecção de colisão, visando ganho de performance.

8. Agradecimentos

O Tecgraf é um grupo prioritariamente financiado pela Petrobras. Alberto Raposo é financiado parcialmente pelo CNPq (processo 472967/2007-0) e pela FAPERJ (Bolsa Jovem Cientista do Nosso Estado).

Referências

- [1] U. Assarsson and T. Moller. Optimized view frustum culling algorithms for bounding boxes. *JOURNAL OF GRAPHICS TOOLS*, 5(1):9–22, 2000.
- [2] J. H. Clark. Hierarchical geometric models for visible-surface algorithms. In *SIGGRAPH '76: Proceedings of the 3rd annual conference on Computer graphics and interactive techniques*, pages 267–267, New York, NY, USA, 1976. ACM.
- [3] M. H. da Silva. 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.
- [4] F. Dunn and I. Parberry. *3D Math primer for Graphics and Game Development*. Wordware Publishing Inc, 2002.
- [5] GlsI. <http://www.clockworkcoders.com/oglsI/downloads.html>, visitado em 23/01/09.
- [6] N. Greene. Detecting intersection of a rectangular solid and a convex polyhedron. *Academic Press Graphics Gems Series*, pages 74–82, 1994.
- [7] G. Gribb and K. Hartmann. 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.
- [8] E. Haines and J. Wallace. Shaft culling for efficient ray-traced radiosity. In *Eurographics Workshop on Rendering*, May 1991.
- [9] Microstation. <http://www.bentley.com/>, visitado em 23/01/09.
- [10] T. Möller and E. Haines. *Real-time rendering*. A. K. Peters, Ltd., Natick, MA, USA, 1999.
- [11] L. O. S. Niels Thrane. A comparison of acceleration structures for gpu assisted ray tracing. Master's thesis, University of Aarhus, August 2005.
- [12] Obj. Especificação do arquivo, <http://local.wasp.uwa.edu.au/pbourke/dataformats/obj/>, visitado em 29/01/09.
- [13] K. Pallister. *Game Programming Gems*, volume 5, pages 65–76. Charles River Media, Rockland, February 2005.
- [14] J. Ponce and O. Faugeras. An object centered hierarchical representation for 3d objects: the prism tree. *Comput. Vision Graph. Image Process.*, 38(1):1–28, 1987.
- [15] libqglviewer. <http://www.libqglviewer.com/>, visitado em 23/01/09.
- [16] Qt. <http://www.qtsoftware.com/>, visitado em 23/01/09.
- [17] R. Toledo. *Interactive Visualization of Massive Data using Programmable Graphics Cards*. PhD thesis, Loria, INRIA institute, 2007.
- [18] J. Verth and L. Bishop. *Essential mathematics for games and interactive applications*. Morgan Kaufmann, San Francisco, 2004.