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.

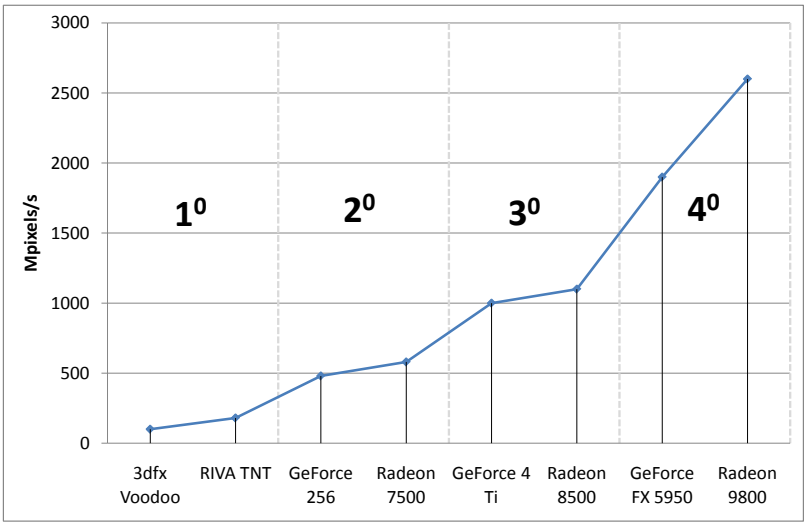

Figura 5.1: Evolução das placas gráficas até a quarta geração.

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.

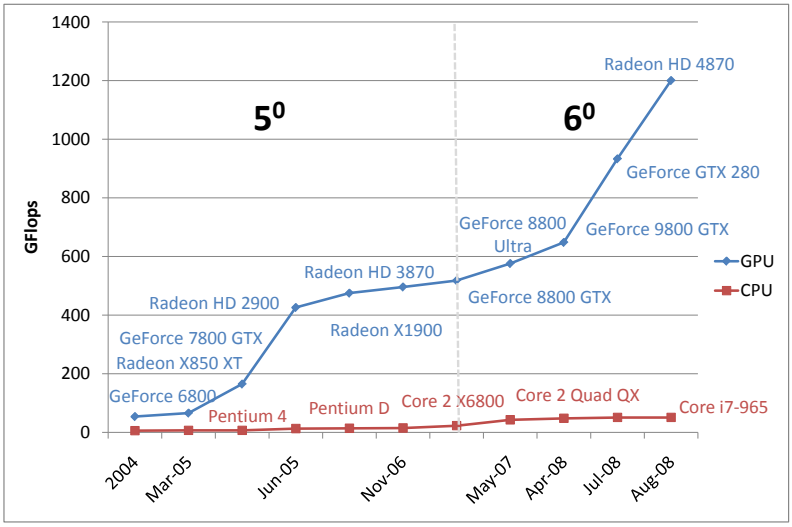

Figura 5.2: Evolução das placas gráficas a partir da quarta geração.

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.

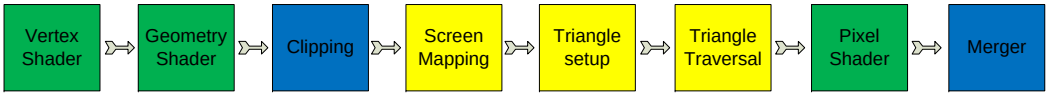

Figura 5.3: Pipeline das placas gráficas modernas.

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*.

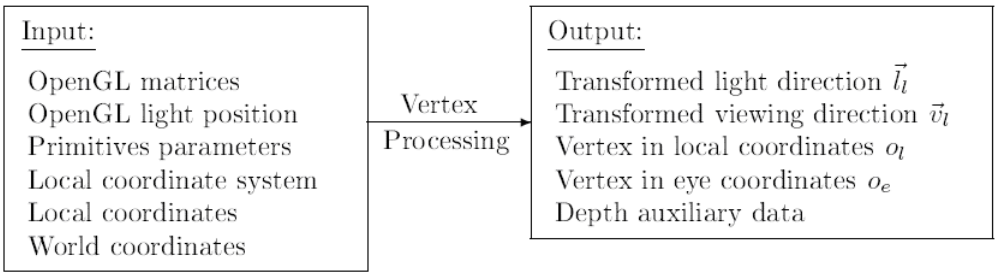

Figura 5.4: Variáveis do vertex shader das gpu primitives (extraída de [70]).

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*.

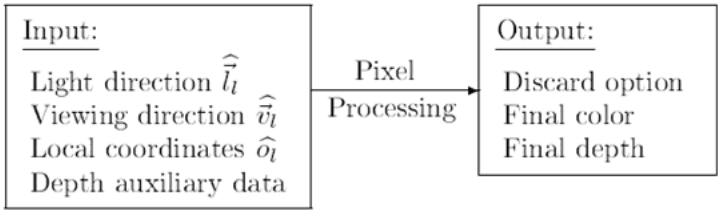

Figura 5.5: Variáveis do pixel shader das gpu primitives (extraída de [70]).

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.

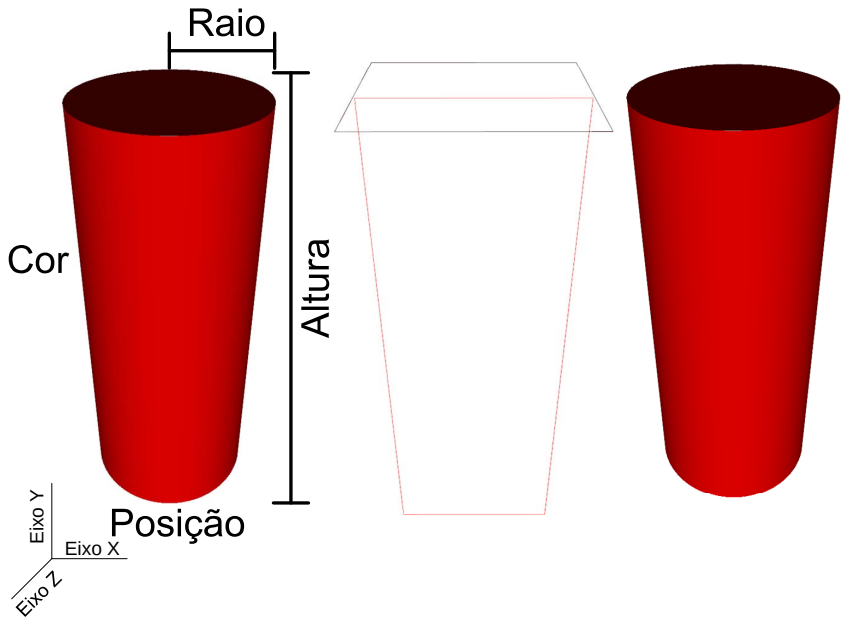

Figura 5.6: Cilindro Gpu primitives.

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.*

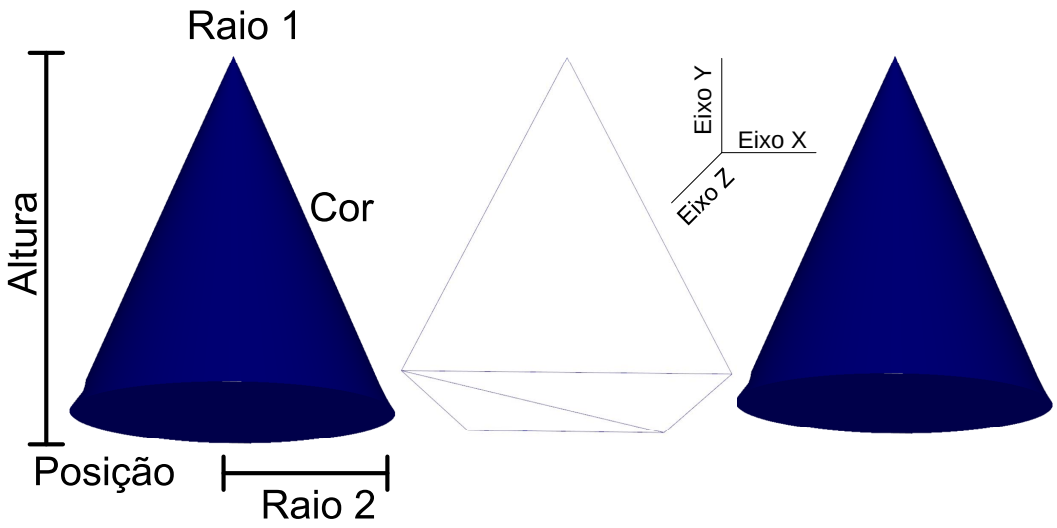

Figura 5.7: Cone GPU primitives.

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*.

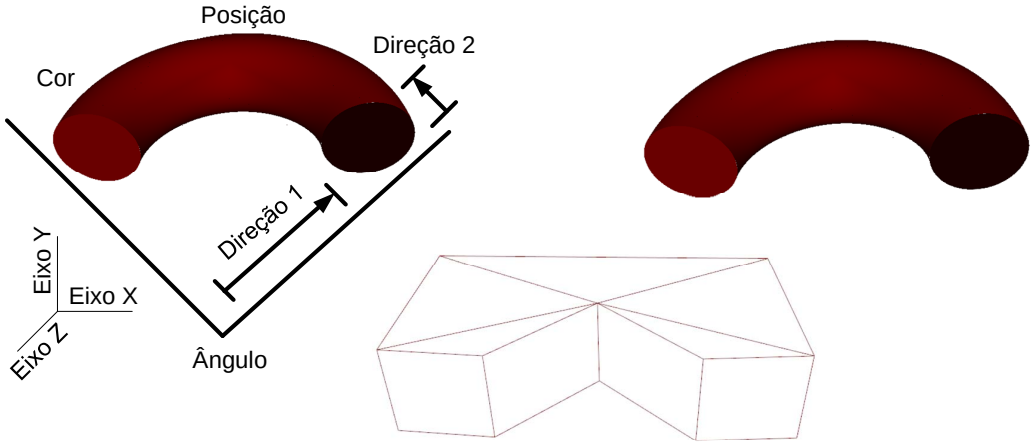

Figura 5.8: Torus slice GPU primitives.

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.

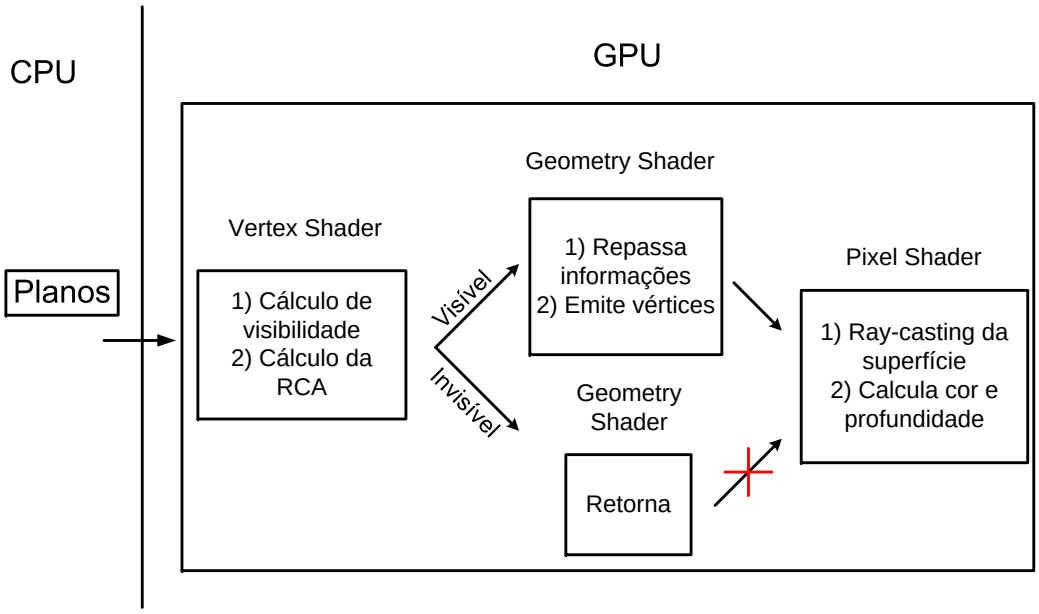

Figura 5.9: Frustum culling junto com GPU primitives.

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.

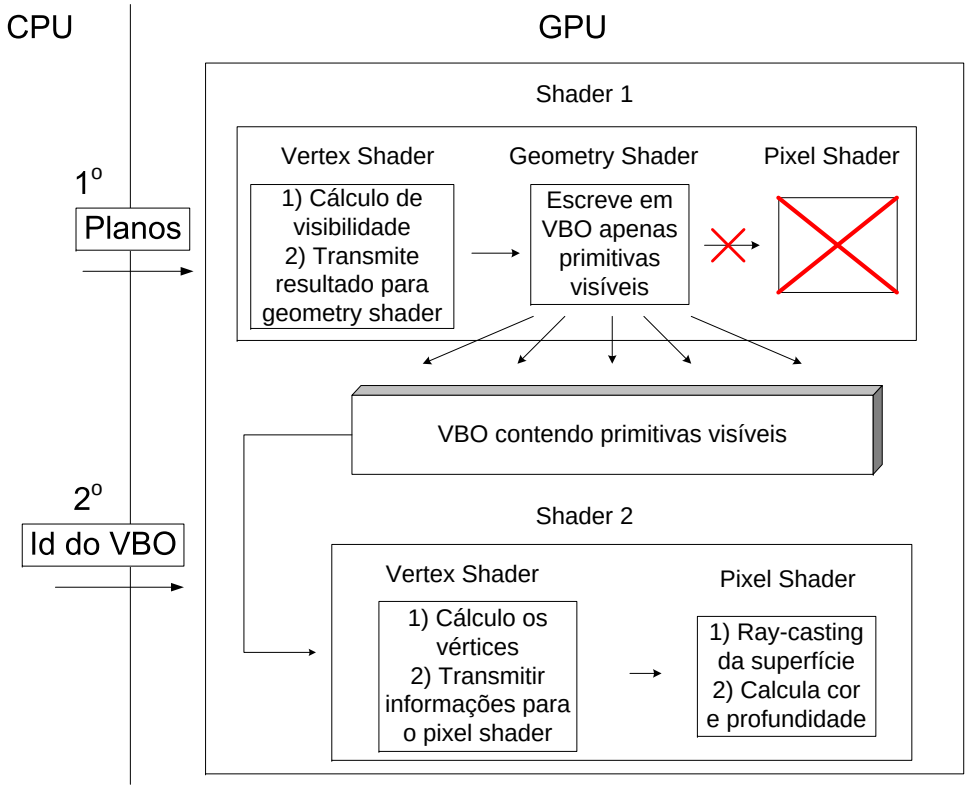

Figura 5.10: Frustum culling em shader separado.

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

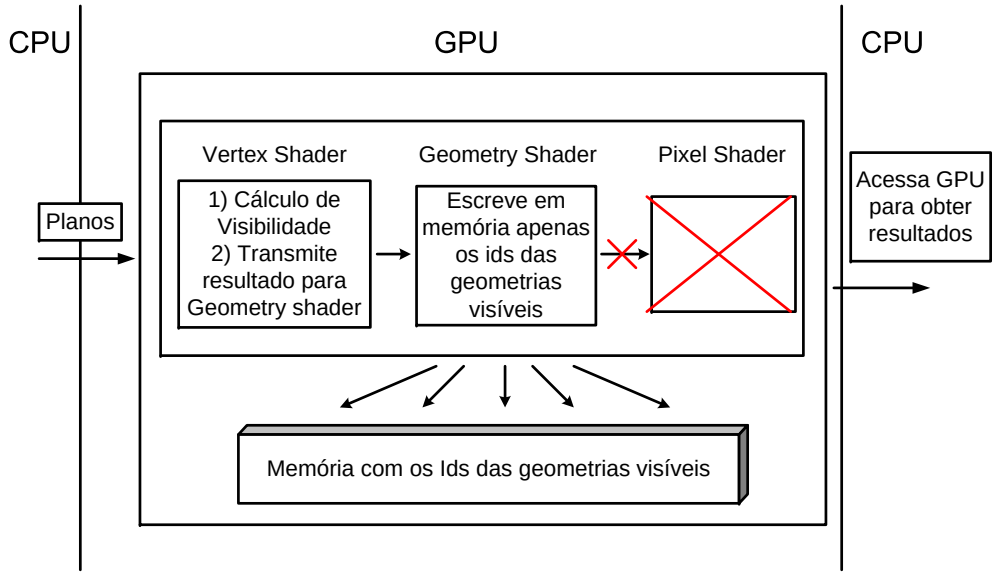

Figura 5.11: Frustum culling em modelos genéricos.

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.

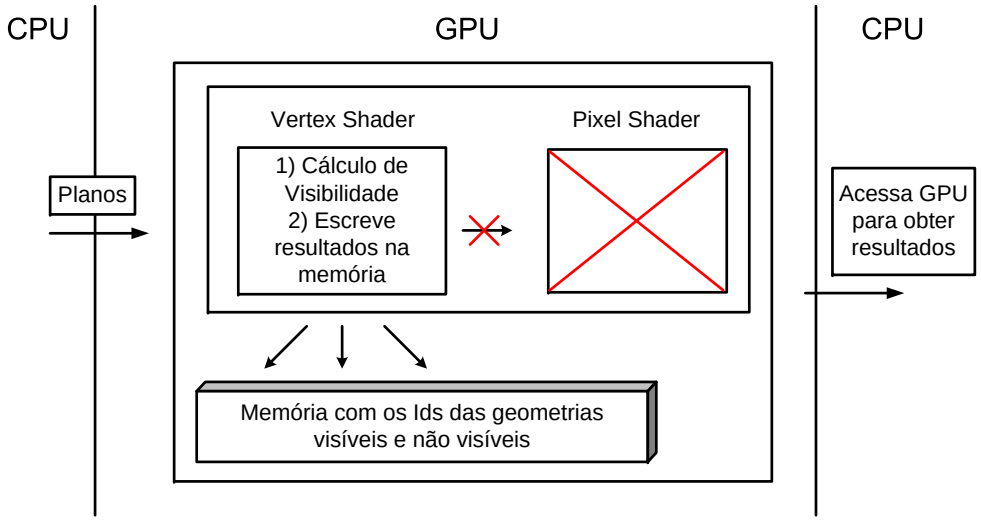

Figura 5.12: Frustum culling em modelos genéricos.

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.

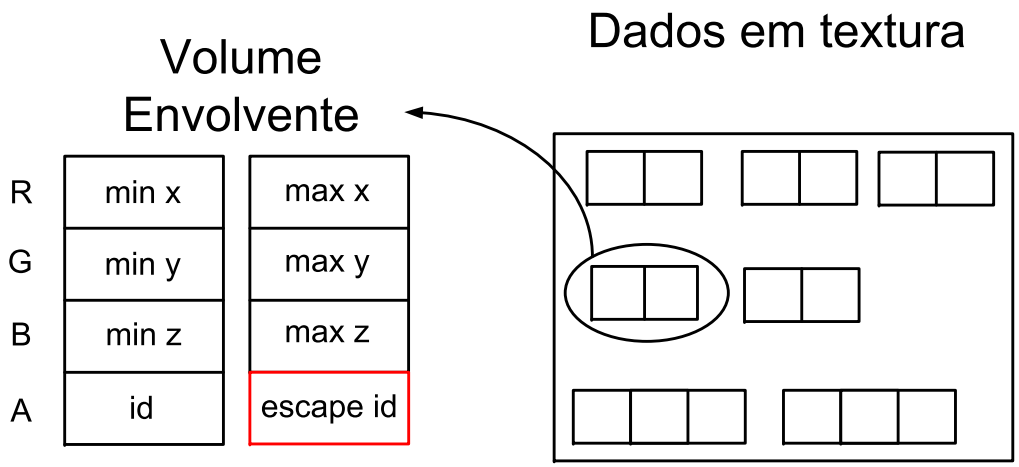

Figura 5.13: Memória utilizada no percurso em GPU.

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

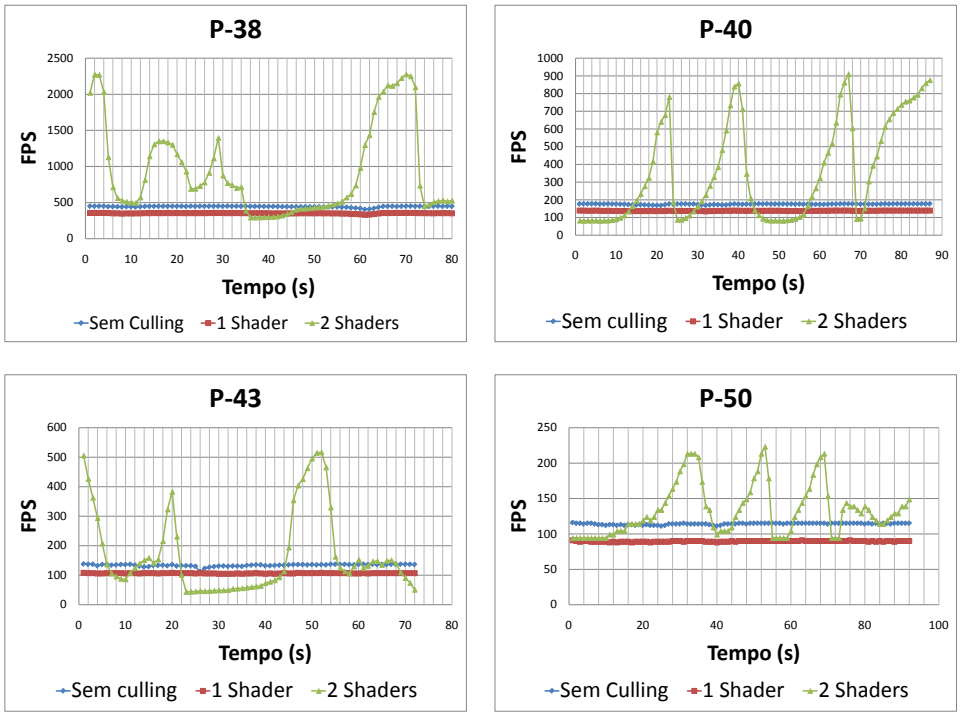

Figura 5.14: Frustum culling nas GPU primitives.

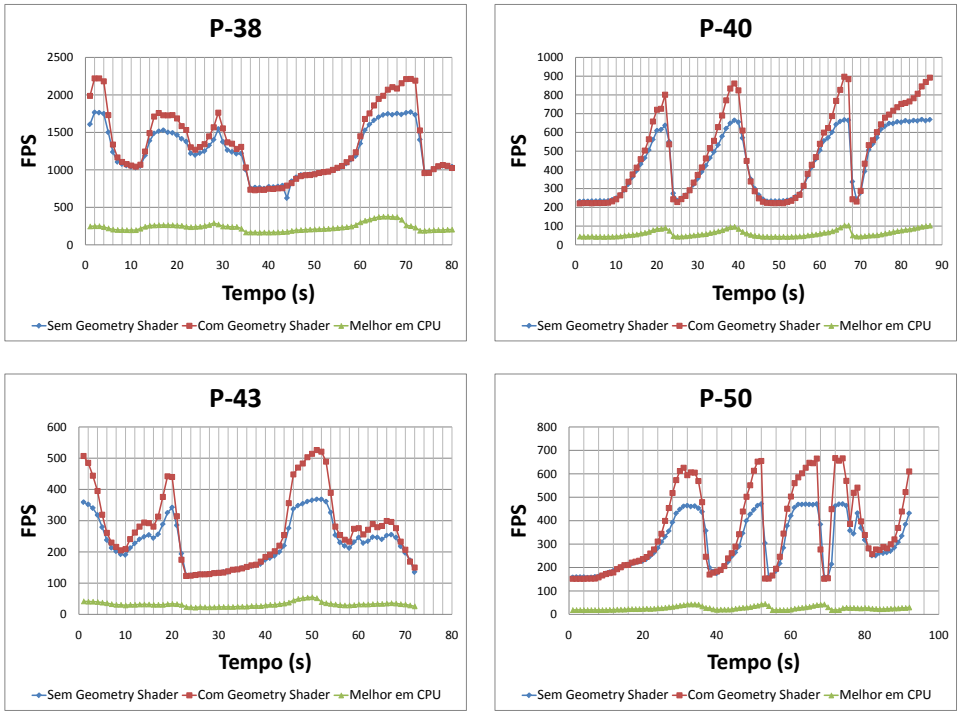

Figura 5.15: Frustum culling em GPU para modelos genéricos.

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.