2
Algoritmos de Visibilidade

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

2.1
Determinação de visibilidade

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

2.1.1
Hidden Surface Removal

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

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

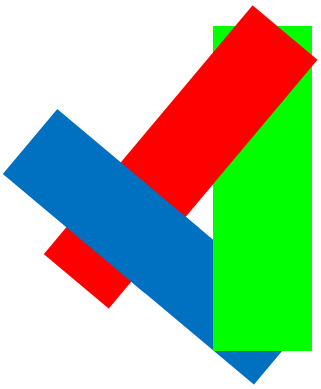

Figura 2.1: Problema com o algoritmo do pintor.

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

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

2.1.2 Culling

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

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

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

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

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

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

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

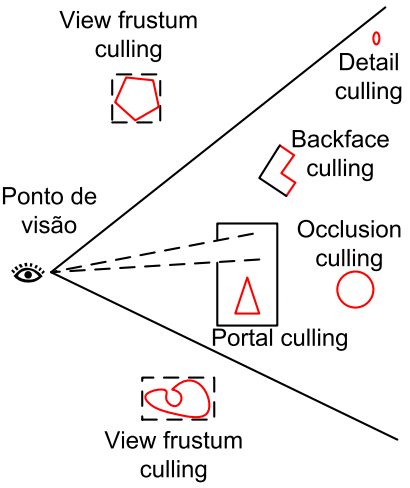

Figura 2.2: Técnicas de culling.

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

2.2 Classificação dos algoritmos de visibilidade

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

Exato X Conservativo X Aproximado

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

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

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

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

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

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

Cenas Dinâmicas X Cenas Estáticas

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

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

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

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

2.3 Pipeline dos algoritmos de visualização

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

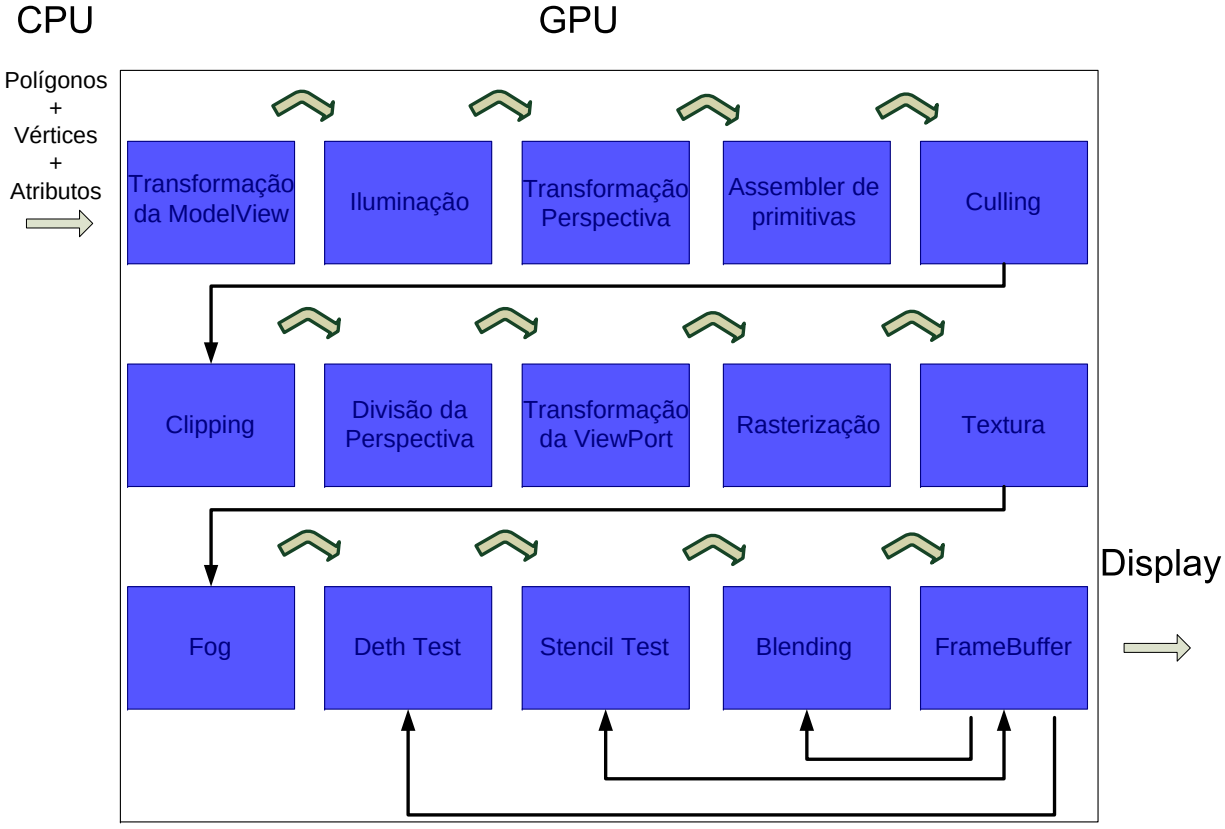

Figura 2.3: Pipeline do OpenGL.

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

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

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

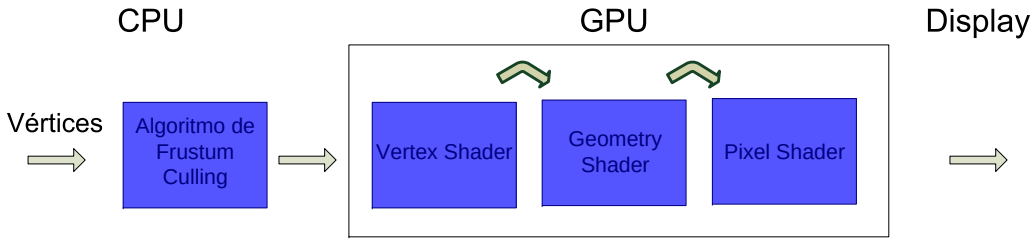

Figura 2.4: Frustum culling em CPU.

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

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

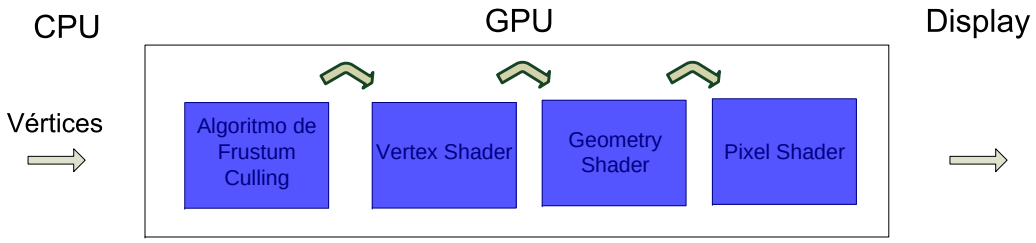

Figura 2.5: Frustum culling em GPU.

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

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

Tabela 2.1: Tabela comparativa de alguns algoritmos de visibilidade.