Relatório

Trabalho Final INF2610 – Renderização em Tempo Real

Aluno André Mazal Krauss, PUC-Rio 2019.1

Professor Waldemar Celes

# Introdução – *Frustum Culling*

No presente relatório, apresento a proposta e implementação da solução desenvolvida como trabalho final para a disciplina INF2610 – Renderização em Tempo Real. Deveríamos escolher uma das técnicas de renderização mais sofisticadas vistas em sala, e a implementar utilizando programação em placa gráfica a partir do OpenGL. A técnica por mim escolhida é denominada, em inglês, de *Frustum Culling.*

A técnica de *Frustum Culling* é uma técnica de aceleração em que se propõe poupar a GPU de trabalho desnecessário e, assim, aumentar a performance da renderização. Isso é feito através da detecção, em CPU, de objetos que estão fora do volume de visão(*frustum*) e, portanto, não precisam ser renderizados, porque não teriam efeito algum sobre a imagem final. Assim, com uma quantidade reduzida de objetos processados por frame, reduz-se o tempo despendido pela GPU e a transferência de dados CPU-GPU, obtendo-se consequentemente uma renderização mais rápida. Para se obter o resultado desejado, uma série de procedimentos precisou ser implementada.

Primeiramente, é necessário escolher uma representação simplificada dos objetos para realizar a detecção. Caso tentássemos usar uma representação com malhas de triângulos, a detecção ficaria demasiadamente complexa e, possivelmente, tornaria a renderização mais lenta, o contrário do que se deseja. Para o presente trabalho, optamos por uma representação utilizando *AABBs*(*Axis Aligned Bounding Box*, ou caixa alinhada). Fez-se necessário, além de representar as AABBs em CPU, implementar os métodos para obtê-las a partir da geometria de um objeto genérico.

Com cada objeto sendo representado por uma AABB e por uma matriz Model, responsável por transformar a coordenadas do espaço do objeto para o espaço do mundo, podemos criar a função que avalia se dado objeto está dentro, fora, ou interceptando o frustum. Isso é feito a partir da matriz de ViewProjection e dos 6 planos que definem o frustum: utilizamos a ViewProjection e a Model para transformar os planos para mesmo espaço que nossas coordenadas AABB, e, então, selecionamos os pontos da AABB mais próximo e mais distante do plano. Podemos então calcular de que lado do plano estes pontos estão: sem ambos estiverem dentro, a AABB está dentro do frustum; se ambos estiverem fora, a AABB está fora; e se um estiver dentro e o outro fora, a AABB intercepta o plano.

Porém, somente realizar esta avaliação para cada objeto não é suficiente para obtermos bons resultados. Ainda que economizemos tempo na GPU, realizar o procedimento acima para cada objeto é trabalho demais à CPU. Para mitigar esse problema, devemos organizar os objetos em uma hierarquia, ou árvore, para que seja possível eliminar uma porção de objetos com poucas verificações. Para tal, a árvore deve ser construída de tal maneira que a AABB de cada nó contenha completamente as AABBs de seus filhos. Foi implementado um procedimento que, antes de começar a renderização, cria uma árvore binária que atende a essa propriedade, e procura minimizar o tamanho das AABBs ao mínimo necessário utilizando um *corte-mediano* (ou *median-cut*, em inglês).

Além disso, com todo o funcionamento básico explicado acima já funcionando, foram implementadas pequenas otimizações pontuais que eliminam cheques redundantes e exploram a coerência temporal da cena renderizada.

Por último, é evidente que se fez necessária a implementação dos *shaders*, por vértices e por fragmentos, para realizar a renderização em GPU. Porém, para o presente trabalho, os *shaders* foram simples adaptações dos shaders usados em trabalhos passados, implementado o modelo de iluminação de Phong com mapeamento de rugosidade e textura. Por isso, esse aspecto do trabalho será mais detalhado abaixo.

# Organização do Projeto

Para o desenvolvimento do projeto foi utilizada a pipeline gráfica com OpenGL e C++. O trabalho está, portanto, dividido em algumas classes implementadas em C++ e em dois shaders, o *vertex shader* e o *fragment shader*. Abaixo, detalho melhor as classes implementadas.

1. **GLWindowManager**: Classe responsável por gerenciar a janela OpenGL, e, portanto, por ditar o fluxo de todo o programa. Aqui estão implementados o loop de renderização, a inicialização da cena e o tratamento de input.
2. **BVHNode**: Classe abstrata, representa um nó genérico na árvore de hierarquia, denominada *Binding Volume Hierarchy*. Dela derivam duas outras classes, a AggregatorNode e a RenderObject, que devem ser nós com filhos e folhas, respectivamente. Nesta classe está implementado tudo o que há em comum a um nó da árvore, como o método *IsInsideFrustum,* que determina se o objeto está ou não dentro do frustum, e define métodos usados para percorrer a hierarquia, como o *CheckFrustumAndRender* e o *Render.* A implementação desses dois últimos é feita pelas classes herdeiras.
3. **RenderObject:** Classe que representa um objeto renderizável, ela herda de BVHNode. Ela implementa os métodos herdados: *Render*, que renderiza o objeto, e *CheckFrustumAndRender*, que renderiza o objeto ou não baseado no retorno do método *IsInsideFrustum.* Vale notar que, nesta classe, os extremos da AABB, Bmin e Bmax, são armazenados no espaço do objeto. Assim sendo, os *getters* *GetBmin* e *GetBmax* os devem transformar para o espaço do mundo antes de retornar. Além disso, essa classe também implementa a inicialização dos *Vertex Array Buffers* e *ElementArrayBuffer* para um objeto.
4. **AggregatorNode:** Classe que implementa um nó na *BVH* que não é um nó folha. É responsável por disparar as funções corretas em seus nós filhos, de acordo com o resultado do teste de visibilidade contra o *frustum.* Vale notar que esta classe representa os extremos da AABB, *Bmin* e *Bmax,* diretamente no espaço do mundo.

# Representação por AABBs