# 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, denominada BVH (*Bounding Volume*

*Hierarchy*) 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

Como explicado acima, para obter um desempenho satisfatório no método de *frustum culling*, é imperativo que cada objeto na cena seja representado por uma caixa envolvente (*bounding box*), que simplifique sua geometria. A caixa envolvente ideal deve ter um volume que cubra por inteiro o objeto, sem cobrir muito volume em excesso; além disso, idealmente seria rápido calculá-la a partir da geometria e também rápido obter sua interseção contra um plano. Na

prática, não temos uma caixa envolvente que seja a melhor em cada um desses requisitos, mas sim *tradeoffs* entre diferentes possibilidades. No caso do presente trabalho, a maior consideração para a escolha de AABBs foi facilidade de implementação. A AABB possibilita o uso de algoritmos simples para sua obtenção, para realizar o cheque contra o frustum de visão, e também para a união de duas AABBs, o que é útil para a construção de nós agregados na árvore hierárquica. Abaixo, detalho como funcionam esses procedimentos.

### Determinação da AABB de um RenderObject

Um RenderObject, sendo um objeto passível de ser renderizado em tela, armazena, dentre outros, uma malha de triângulos. O algoritmo descrito abaixo obtém uma AABB, no espaço do modelo, a partir desta malha.

Obs: a AABB é representada por dois campos glm::vec3, Bmax e Bmin. Eles armazenam, respectivamente, o ponto de valores máximo e de valores mínimos do objeto, efetivamente representando uma caixa, ou seis planos.

1. Inicializa Bmax como (MIN_VALUE, MIN_VALUE, MIN_VALUE) e Bmin como (MAX_VALUE, MAX_VALUE, MAX_VALUE).
2. Para cada vértice V na malha de triângulos do modelo:
    a. Para i = 0..2:
        i. Se V[i] > Bmax[i]:
            1. Bmax[i] = V[i]
        ii. Se V[i] < Bmin[i]:
            1. Bmin[i] = V[i]

Dessa maneira, obtemos os pontos Bmin e Bmax que determinam a bounding box. O algoritmo é O(n), onde n é o número de vértice no modelo.

Após a obtenção de Bmin e Bmax, podemos facilmente construir a LUT.

```
void BVHNode::UpdateLUT()
{
    //com Bmin e Bmax, constrói LUT para acesso rápido de Vmin, Vmax
    LUT[0] = glm::vec3(Bmin.x, Bmin.y, Bmin.z); // 0,0,0 -> min, min min
    LUT[1] = glm::vec3(Bmin.x, Bmin.y, Bmax.z); // 0,0,1 -> min, min, max
    LUT[2] = glm::vec3(Bmin.x, Bmax.y, Bmin.z); // 0,1,0 -> min,max,min
    LUT[3] = glm::vec3(Bmin.x, Bmax.y, Bmax.z); // 0,1,1 -> min,max,max
    LUT[4] = glm::vec3(Bmax.x, Bmin.y, Bmin.z); // 1,0,0 -> max,min,min
    LUT[5] = glm::vec3(Bmax.x, Bmin.y, Bmax.z); // 1,0,1 -> max,min,max
    LUT[6] = glm::vec3(Bmax.x, Bmax.y, Bmin.z); // 1,1,0 -> max,max,min
    LUT[7] = glm::vec3(Bmax.x, Bmax.y, Bmax.z); // 1,1,1 -> max,max,max
}
```

### Determinação da AABB de um Aggregate Node

Um nó agregado é definido como um nó com dois filhos, cada um possuindo uma AABB. Assim sendo, é trivial obter a AABB do nó agregado, tomando o mínimo de cada árvore em cada coordenada da Bmin, e o máximo para o Bmax. Ou seja:

```
//adjust bounds: a nova caixa pega os mins e maxs das dimensões das duas subcaixas
glm::vec3 LBmin = left->GetBmin();
glm::vec3 RBmin = right->GetBmin();

glm::vec3 LBmax = left->GetBmax();
glm::vec3 RBmax = right->GetBmax();

agg->Bmin = glm::vec3(glm::min(LBmin.x, RBmin.x), glm::min(LBmin.y, RBmin.y), glm::min(LBmin.z, RBmin.z));
agg->Bmax = glm::vec3(glm::max(LBmax.x, RBmax.x), glm::max(LBmax.y, RBmax.y), glm::max(LBmax.z, RBmax.z));
agg->UpdateLUT();
```

Novamente, uma vez que obtemos Bmin e Bmax, setamos a LUT.

### Determinação da posição da AABB relativa ao Frustum de visão

Após a obtenção da AABB no pré-processamento, devemos poder determinam se ela está dentro, fora, ou interceptando o frustum de visão.

1. Obter os 6 planos do frustum, no mesmo espaço em que está definida a LUT.[1]
2. Para cada plano:
    a. Escolher Pmin e Pmax, acessando a LUT com os sinais da normal do plano
    b. Calcular a distância com sinal (*signed distance)* de Pmin e Pmax até o plano[2]
    c. Se distância(Pmin) e distância(Pmax) < 0:
        i. Retornar FORA
    d. Senão, se distância(Pmin) < 0 e distância(Pmax) > 0:
        i. Setar flag INTERCEPTA
3. Se flag INTERCEPTA está setada:
    a. Retorna INTERCEPTA
4. Senão:
    a. Retorna DENTRO

Somente neste algoritmo percebemos a importância de usarmos uma AABB em vez de uma malha de triângulo para este procedimento: com ela, precisamos testar dois pontos somente, o que faz com que a operação tenha complexidade constante O(1). Se não usássemos simplificação alguma da geometria, seria necessário testar cada vértice, o que faria com que a operação tivesse custo linear no número de vértices [O(n)] e, nesse caso, provavelmente seria mais rápido simplesmente renderizar o objeto.

## *Bounding Volume Hierarchy*(BVH)

Ainda que seja possível determinar, para cada objeto, se ele está ou não dentro do frustum de visão, isso ainda não é o suficiente para se obter um bom resultado. O grande problema é que o número de teste que teríamos que realizar é linear com o número de objetos [O(n)]. Por outro lado, caso quiséssemos renderizar todos os objetos, não utilizando frustum culling, a complexidade total também seria linear; assim sendo, é possível que seja melhor simplesmente renderizar tudo.

Para sanar este problema e melhorar o desempenho, introduzimos a hierarquia de volumes envolvente, que reduz o número de checagens, no pior caso, de O(n) para O(log(n)). Creio que o este funcionamento é melhor explicado diretamente pelo algoritmo que percorre a hierarquia e a renderiza.

### Renderização da BVH

A renderização da BVH é um procedimento recursivo, começando do nó raiz. Na implementação do programa, esse procedimento recursivo de checar o frustum e potencialmente renderizar o nó é implementado pelos métodos *CheckFrustumAndRender*, que é sobrescrito nas classes RenderObject e AggregateNode

1. Obter o resultado da checagem da AABB do nó corrente contra o frustum:

---

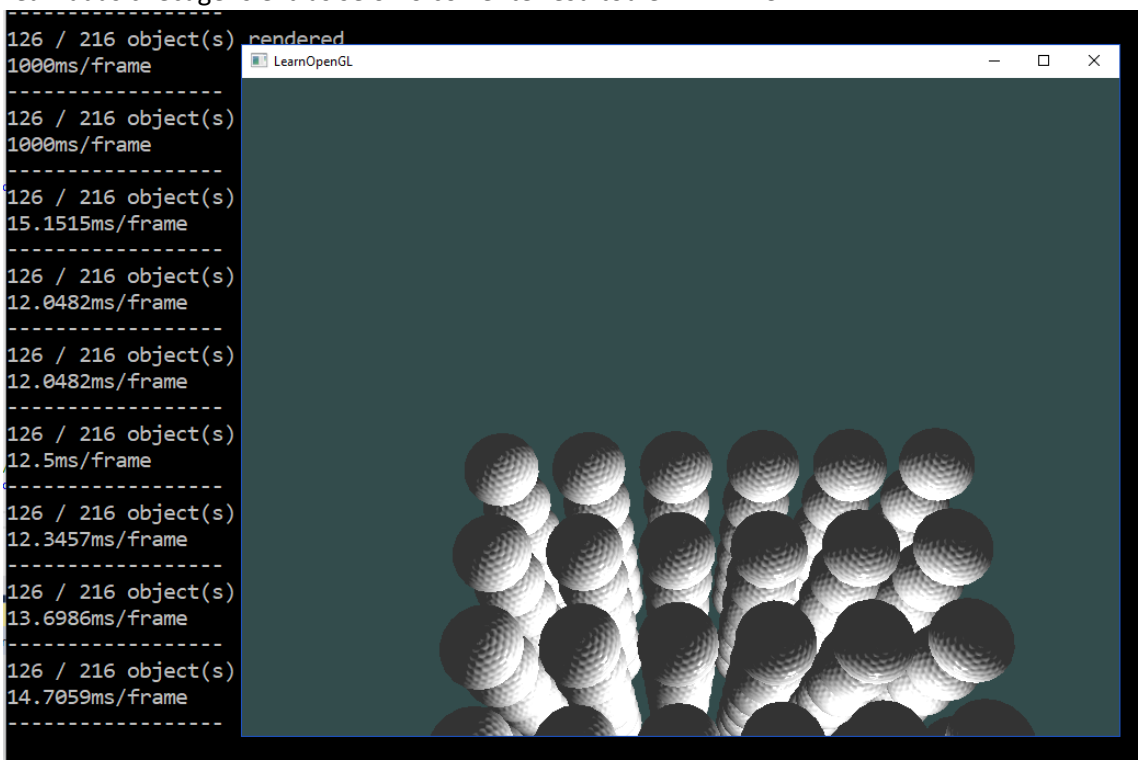
[1] Esse espaço não é sempre o mesmo. Para um AggregateNode, este espaço é o do mundo, e para um RenderObject, este espaço é o do objeto

[2] Essa é uma operação linear, melhor definida aqui: http://mathworld.wolfram.com/Point-PlaneDistance.html

a. Se o resultado for DENTRO:
    i. Renderiza o próprio nó, se for uma folha (RenderObject)
    ii. Renderiza os nós filhos, sem checar novamente, se for um nó não-folha (AggregateNode)
b. Se o resultado for FORA:
    i. Retorna
c. Se o resultado for INTERCEPTA
    i. Renderiza o próprio nó, se for uma folha (RenderObject)
    ii. Chama *CheckFrustumAndRender* recursivamente para os nós filhos

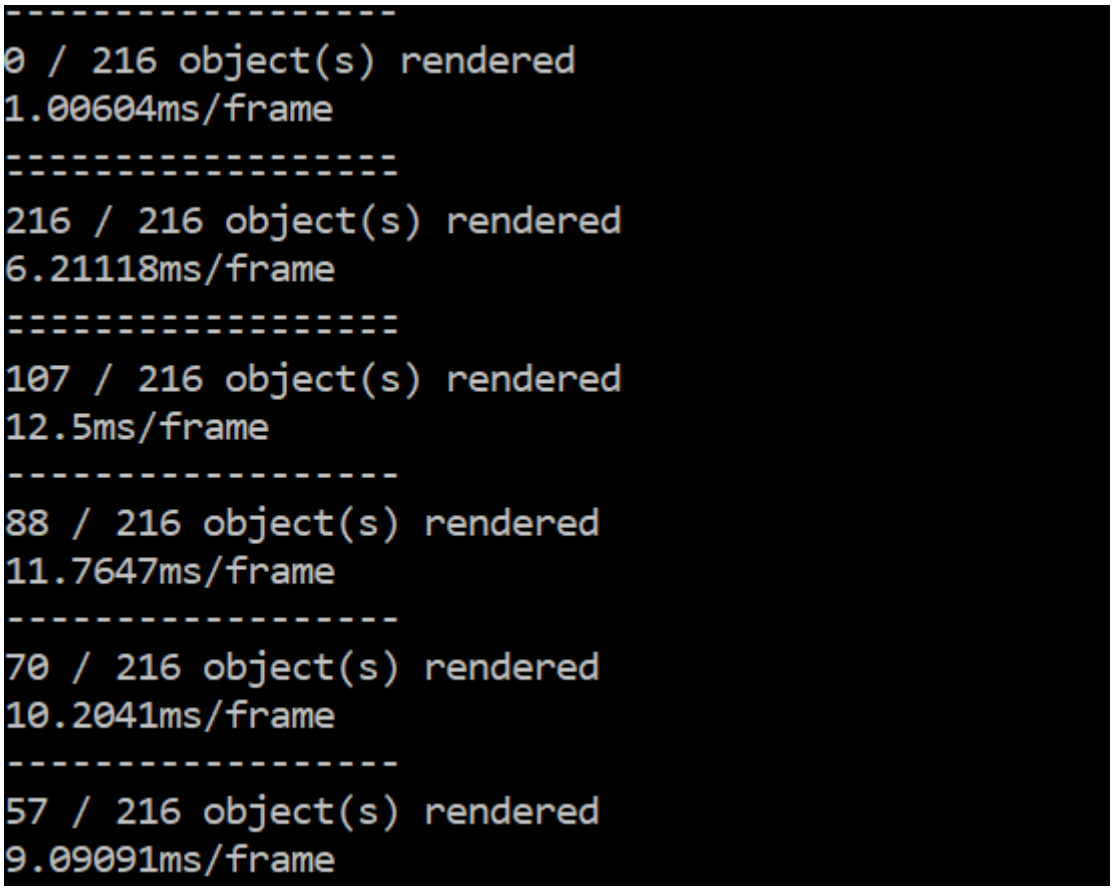
Dessa maneira, o número de checagens necessárias cai drasticamente. Por exemplo, nas situação do nó raiz estar totalmente fora ou totalmente dentro do frustum, realizamos a checagem somente uma vez. De forma geral, em vez de eliminarmos somente um objeto por checagem, podemos eliminar até 2^(n-i) – 1, onde n é a profundidade máxima da árvore e i é o andar atual. Além disso, foi implementado uma otimização que aproveita a esperada coerência temporal dos objetos. Quando um objeto é eliminado por um plano, este plano ganha a prioridade para as próximas checagens sobre este mesmo objeto. Isso acelera significativamente a renderização quando não a cena permanece estática por qualquer período de tempo.

Vale notar que é possível ter uma noção da quantidade de checagens realizadas apertando a tecla ‘0’ e vendo o arquivo out.txt, usado para debug. Ele registra todo o caminho percorrido até as folhas da árvore, onde 0 indica DENTRO, 1 indica FORA, e 2 indica INTERCEPTA. É importante frisar que, para este debug, a árvore é percorrida sempre até o final e as checagens são realizadas para cada nó, mas o mesmo não ocorre na a renderização normal, onde só são realizadas checagens extras se o nó corrente resultou em INTERCEPTA.




Somente 126 / 216 objetos são passados à GPU para renderização

## Resultados




No exemplo desta cena, podemos ver casos em que o algoritmo está funcionando bem, e casos em que ainda deveriam ser feitas melhorias. Nos dois primeiros, vemos uma grande diferença de tempo entre renderizar todos os objetos e renderizar nenhum. Nesse caso, todos os objetos estavam confortavelmente dentro/fora do frustum, e era gasto pouco tempo com a checagem de visibilidade. Já nos demais casos, vemos claramente como, em casos intermediários, gasta-se muito tempo com a checagem, e poderia ser melhor simplesmente renderizar todos os objetos da cena. Proponho duas melhorias: repensar a construção da árvore de hierarquia, procurando dividir espacialmente os objetos de forma melhor, e possivelmente fazendo ela ser uma árvore não binária; e implementar flags para descartar, nos nós filhos, checagens contra planos que já tenham sido marcados como DENTRO nos nós pais.