Diego Cordeiro Barboza

IMPLEMENTAÇÃO DO RAY TRACING ACELERADO POR UMA OCTREE EM GPU

Dissertação apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Mestre. Área de Concentração: Computação Visual.

Orientador: Prof. Dr. Esteban Walter Gonzalez Clua

Niterói

2011

Ficha Catalográfica elaborada pela Biblioteca da Escola de Engenharia e Instituto de Computação da UFF

B238 Barboza, Diego Cordeiro

Implementação do ray tracing acelerado por uma octree em GPU/ Diego Cordeiro Barboza. – Niterói, RJ : [s.n.], 2011.
58 f.

Dissertação (Mestrado em Computação) - Universidade Federal Fluminense, 2011.
Orientador: Esteban Walter Gonzalez Clua.

1. Processamento de imagem. 2. Computação visual. 3. Unidade de processamento gráfico. 4. Árvore octária. 5. Traçado de raios. 6. Estrutura de dados. I. Título.

CDD 006.37

DIEGO CORDEIRO BARBOZA

IMPLEMENTAÇÃO DO RAY TRACING ACELERADO POR UMA OCTREE EM GPU

Dissertação apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Mestre. Área de Concentração: Computação Visual.

Aprovada em Agosto de 2011.

BANCA EXAMINADORA

_____

Prof. Dr. Esteban Walter Gonzalez Clua – Orientador
Universidade Federal Fluminense

_____

Prof. Dr. Leandro Augusto Frata Fernandes
Universidade Federal Fluminense

_____

Prof. Dr. Ricardo Farias
Universidade Federal do Rio de Janeiro

Niterói
2011

À minha família e a todos que sempre estiveram do meu lado, nos melhores e piores momentos.

AGRADECIMENTOS

Agradeço à minha família, em especial à minha mãe e minha irmã, pela motivação e confiança que depositaram em mim.

À Adrian, por todos estes anos ao meu lado e pela compreensão e apoio que sempre me deu.

Ao professor Esteban Clua, pelo auxílio e incentivo dado durante todo o mestrado.

Aos amigos que fiz na UFF pelas conversas de corredor, pela troca de conhecimento e pela amizade verdadeira.

Ao Instituto de Computação pela oportunidade de cursar o mestrado e pela infra-estrutura oferecida.

“Não entre em pânico.”

Douglas Adams

RESUMO

Ray tracing é uma técnica amplamente empregada na geração de imagens computadorizadas com alto grau de fidelidade e realismo. No entanto, esta técnica apresenta um custo computacional bastante elevado, devido principalmente aos cálculos de interseção que devem ser realizados pelo algoritmo. Por outro lado, o *ray tracing* é um algoritmo altamente paralelizável, visto que o cálculo de um raio de luz qualquer é independente dos demais. Assim, a implementação do *ray tracing* em GPU é um processo natural.

Estruturas de dados podem ser empregadas para reduzir a carga de processamento do *ray tracing*, minimizando o número de cálculos de interseção realizados. Em trabalhos recentes, estas estruturas têm sido levadas à GPU a fim de acelerar buscas em algoritmos paralelos diversos.

Neste trabalho, é proposta uma implementação do algoritmo de *ray tracing* em GPU utilizando uma *octree* como estrutura de aceleração. Todos os passos para a execução do *ray tracing* em paralelo ocorrem diretamente na GPU, portanto elimina-se o gargalo de comunicação entre a CPU e a GPU. O trabalho também se propõe a estudar qual a melhor representação de uma *octree* na GPU, realizando um estudo comparativo entre duas implementações distintas.

Palavras-chave: Estruturas de dados; Octree; Ray tracing; GPU.

ABSTRACT

Ray tracing is a largely employed technique for generating computer images with high fidelity and realism. However, this technique is very costly, mainly because of the intersection calculations that are made by the algorithm. Still, the ray tracing is a highly parallelizable algorithm, bacause the calculations for a single light ray are independent from the computation of the others. This way, the implementation of the ray tracing on the GPU is a natural process.

Data structures can be employed for reducing the processing load of the ray tracing algorithm, minimizing the number of intersection calculations to be done. In recent works, these structures are being ported to the GPU, so they can be used to accelerate various parallel algorithms.

In this work, it is proposed a GPU ray tracing implementation using an octree as acceleration structure. Every single step of the algorithm runs in parallel on the GPU, so the communication bottleneck between the GPU and the CPU is eliminated. This work also proposes the study of which is the best way to represent the octree on the GPU, by comparing two different implementations.

Keywords: Data strucutres; Ocree; Ray tracing; GPU

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

CPU: Central Processing Units

CUDA: Computer Unified Device Architecture

PBO: Pixel Buffer Object

GPGPU: General-Purpose computation on Graphics Processing Units

GPU: Graphics Processing Unit

GLOSSÁRIO

Ray tracing: Algoritmo de iluminação por traçado de raios

Octree: Árvore em que, a cada nível, os nós são subdividos em oito partes

Quadtree: Árvore em, a cada nível, os nós são subdividos em quatro partes

Kernel: Programa executado na placa gráfica de propósito geral

OpenGL: Biblioteca gráfica

Shader de fragmento: Programa que substitui o *pipeline* tradicional no tratamento de fragmentos

Shader de vértice: Programa que substitui o *pipeline* tradicional no tratamento de vértices

Tabela hash: Uma estrutura de dados que armazena objetos em posições de memória dada por uma função de dispersão

Função de hash: Função de dispersão usada para mapear dados em uma tabela *hash*

Voxel: *Volumetric Pixel*. Usado aqui como sinônimo de octante

Pixel: Menor ponto que forma uma imagem digital

SUMÁRIO

CAPÍTULO 1 – INTRODUÇÃO

O algoritmo de iluminação por *ray tracing* é uma técnica bastante popular para a geração de imagens computadorizadas, graças à alta fidelidade visual dos resultados obtidos. Filmes e aplicações não-interativas, como alguns sistemas de visualização médica, podem empregar o *ray tracing* na geração imagens de alta qualidade e complexidade. No entanto, o *ray tracing* por ser um modelo baseado em iluminação pixel a pixel e recursivo, é um algoritmo de alto custo computacional, o que torna difícil seu uso em aplicações em tempo real, como jogos digitais.

O *ray tracing* simula o comportamento da luz no seu processo de geração de imagens. Tipicamente, utiliza-se o *ray tracing* inverso, que é construído ao se disparar raios a partir do ponto de visualização e seguir estes raios através da cena até um ponto de luz. Para cada um dos muitos raios que devem ser disparados para representar a região de visualização do espectador, deve-se calcular a interseção com todas as primitivas que compõem a cena, a fim de determinar a cor final de cada ponto. Para imagens com altas resoluções e cenas complexas, com alto número de primitivas, a aplicação do algoritmo em tempo real é impraticável.

O algoritmo de *ray tracing* é intrinsecamente paralelizável, uma vez que a computação de um raio de luz qualquer que atravesse a cena não possui qualquer dependência com os demais. Assim, pode-se utilizar os recursos de programação das placas gráficas atuais para executar o algoritmo em paralelo, com tantos raios sendo processados simultaneamente quanto o número de processadores disponíveis.

No entanto, o processamento individual de cada raio ainda pode ser uma tarefa muito custosa. Para tentar minimizar este problema, estruturas espaciais podem ser empregadas a fim de reduzir o número de cálculos de interseção para cada raio. Sem uma estrutura deste tipo, todo raio deve ser testado contra todas as primitivas da cena. Com a aplicação de uma estrutura de dados que divida a cena espacialmente, pode-se primeiro percorrer a estrutura e determinar um subconjunto de primitivas que serão testados.

Uma das formas mais simples de se estruturar a cena é através da utilização de volumes envolventes. Cada subconjunto de primitivas deve ser envolvido por um volume, tal como uma caixa ou uma esfera, por exemplo. O algoritmo de *ray tracing* passa por dois passos, neste caso. Primeiro é preciso determinar com quais destes volumes houve interseção,

para, a seguir, testar a interseção dos raios com as primitivas contidas nestes volumes. Esta, abordagem permite eliminar um grande número de cálculos de interseções, pois apenas os raios que tiverem colisão com os volumes envolventes serão testados com as primitivas e, mesmo neste caso, cada raio também será testado com um número inferior de objetos.

Outra abordagem é a construção de uma árvore para subdivisão espacial da cena mantendo uma estrutura hierárquica. O próprio método descrito anteriormente pode ser utilizado de forma hierárquica, onde cada subconjunto de volumes envolventes pertence a um volume maior, contido em um nível acima na hierarquia até a raiz.

As *octrees* são estruturas hierárquicas em árvore onde cada nó interno contém oito filhos e cada um destes filhos é subdivido novamente em oito partes, até que se alcance uma folha. A busca em uma *octree* é feita a partir da raiz e, para cada nó, determina-se quais de seus filhos pertencem à região de busca. Os nós pertencentes a esta região são enfileirados, enquanto os demais são descartados. A busca prossegue nos nós enfileirados até que se alcance uma folha. Quando uma folha for alcançada, apenas um pequeno subconjunto de primitivas terá restado, reduzindo o número total de cálculos de interseção.

Este trabalho propõe a utilização desta estrutura de *octree* na GPU (Graphics Processing Unit), permitindo processar o *ray tracing* em paralelo ao mesmo tempo em que se aproveita de uma estrutura de particionamento do espaço a fim de reduzir o número total de cálculos de interseção. Este trabalho estuda a melhor maneira de se representar os dados necessários na GPU e sua aplicação no algoritmo de *ray tracing* em paralelo.

1.1 MOTIVAÇÃO

[WHITTED, 1980] observa que cerca de 75% ou mais do tempo gasto na geração de imagens por *ray tracing* é gasto com cálculos de interseção dos raios com os objetos da cena. Ele sugere que melhorias na maneira com que os cálculos de interseção são executados e a possível paralelização deste problema podem trazer um grande ganho de desempenho ao algoritmo.

Existem diversos trabalhos que exploram o algoritmo de *ray tracing* em paralelo. Entre eles, [PURCELL, 2002] apresenta uma discussão sobre o modelo de *ray tracing* para GPU e uma comparação entre este tipo de implementação e a implementação em CPU, levando em conta questões de tráfego de dados e poder de processamento dos dispositivos. [WALD e SLUSALLEK, 2001] apresenta um levantamento sobre o estado da arte do *ray tracing*, destacando a implementação em paralelo em *clusters* de computadores.

[HORN *et al*, 2007] e [REVELLES *et al*, 2000] apresentam estudos sobre a aplicação de estruturas de aceleração no *ray tracing* e sobre o percorrimento eficiente de uma *octree*.

Combinando os conceitos acima, este trabalho visa descrever a cena a ser visualizada em uma estrutura de dados hierárquica que permita uma grande redução no cálculo de interseções de cada raio. Ao mesmo tempo, cada raio pode ser trabalhado em paralelo, utilizando o *hardware* gráfico programável disponível atualmente no mercado.

1.2 CONTRIBUIÇÃO

Este trabalho tem como principal contribuição a modelagem dos dados para a execução do algoritmo *de ray tracing* em paralelo, utilizando *hardware* gráfico atual. Emprega-se o conceito de *octrees* para particionar o espaço e reduzir o número total de cálculos de interseção dos raios com as primitivas da cena, com conseqüente diminuição no tempo gasto nesta tarefa pelo algoritmo de *ray tracing*.

A *octree* é construída em memória de uma maneira otimizada, visando o menor custo para tornar possível sua utilização em placas gráficas com recursos limitados. Para isto, este trabalho propõe uma representação dos dados da *octree* que trata nós internos e folhas de maneira diferente, minimizando o desperdício de memória.

Este trabalho também contribui ao apresentar um estudo comparativo entre duas diferentes representações da *octree* em GPU, investigando os ganhos de desempenho ao se utilizar tabelas *hash* ou listas lineares para esta representação.

1.3 ESTRUTURA DA DISSERTAÇÃO

Este trabalho está estruturado da seguinte forma: O Capítulo 2 apresenta uma revisão bibliográfica sobre *octrees* e sua utilização em algoritmos paralelos em placas gráficas. Ainda é apresentado um estudo sobre o algoritmo de *ray tracing*. O Capítulo 3 descreve como os dados foram representados para o uso na GPU e o Capítulo 4 descreve o funcionamento do *ray tracing* juntamente com a *octree*. O Capítulo 5 descreve os testes elaborados e apresenta os resultados encontrados. No Capítulo 6 são apresentadas as conclusões e sugestões de trabalhos futuros sobre o tema.

CAPÍTULO 2 – REVISÃO BIBLIOGRÁFICA

Este capítulo apresenta uma revisão bibliográfica relativas à utilização de *octrees* para o particionamento espacial de uma cena em três dimensões (seção 2.1) e ao algoritmo de iluminação por *ray tracing* (seção 2.2).

2.1 OCTREES

Uma *octree* é uma estrutura de dados que representa a divisão de um espaço tridimensional em uma estrutura hierárquica, onde cada elemento é representado por um cubo e pode ter entre zero e oito filhos localizados em um nível imediatamente abaixo do seu [CASTRO, 2008].

A *octree* é construída a partir da sua raiz. Uma caixa envolvente é criada ao redor da cena e a raiz é dividida em oito nós. Para cada nó resultante, verifica-se se há interseção entre ele e as primitivas contidas no pai. Em caso positivo, este novo nó é um nó interno que será subdivido novamente no próximo nível até que se alcance o nível máximo da árvore ou ele contenha o número máximo desejado de primitivas. Caso contrário, trata-se de um nó vazio que não será mais processado. A Figura 1 apresenta uma visão esquemática da construção de uma *octree* e na Figura 2 é apresentado um exemplo real de árvore criada a partir de um modelo.

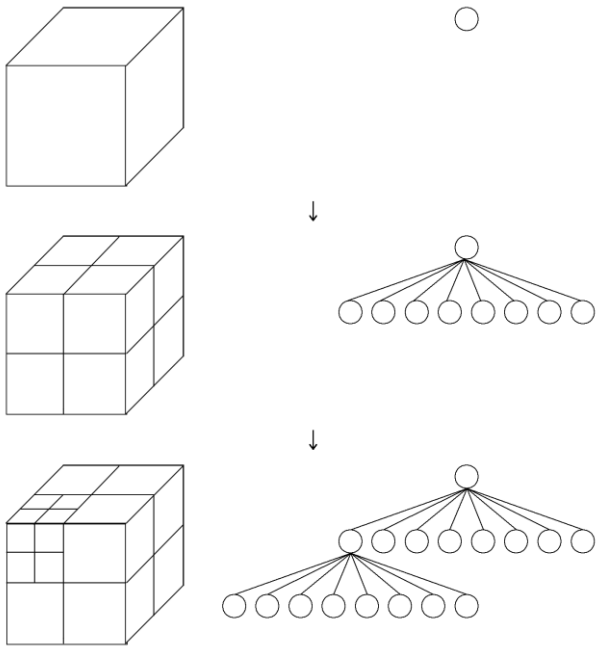

Figura 1: Visão esquemática da construção da *octree* [MADEIRA, 2010].

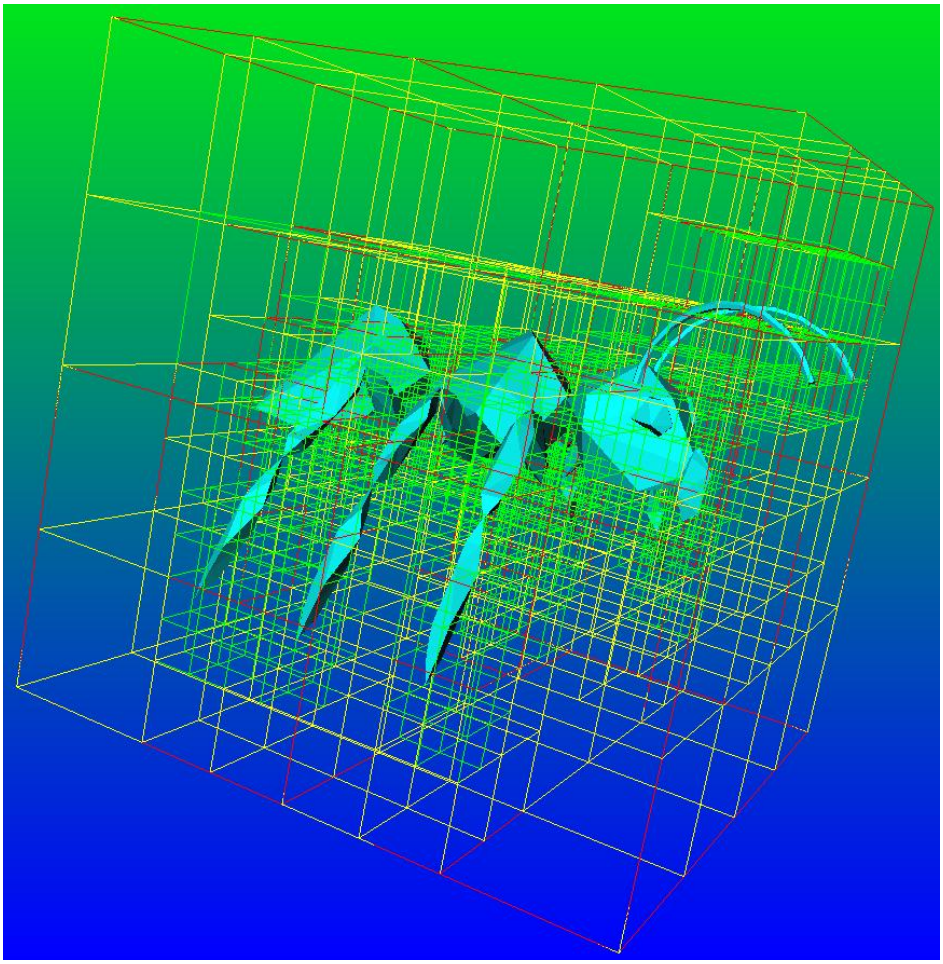

Figura 2: *Octree* criada a partir de um modelo 3d.

Existem várias formas de representar a relação entre os nós de uma *octree* para a construção da estrutura de dados:

- A representação por ponteiros armazena no pai os ponteiros para todos os seus filhos e a navegação é feita de forma direta. Há ainda a opção onde cada nó possui apenas o ponteiro para o primeiro filho e seu irmão à direita. A navegação pelos filhos de um nó é feita através deste primeiro ponteiro e depois pela seqüência de irmãos;

- A *octree* linear é armazenada em um vetor e o acesso aos filhos de um nó é feito de forma direta, através do índice dado por (8 * *índiceDoPai* + *índiceDoFilho*). A desvantagem clara deste método é o alto consumo de memória, visto que árvores não-cheias irão apresentar um grande número de posições vazias. Além disto, a construção do vetor tem seu tamanho condicionado ao conhecimento prévio do nível máximo da árvore;

- A *octree* representada em uma tabela *hash* permite acessar os filhos de um nó em tempo constante a partir de cálculos feitos sobre o índice do pai. Também é possível acessar um nó qualquer baseado em sua posição no espaço ou na hierarquia na *octree*;

- Outras possibilidades incluem a criação de uma árvore representada por ponteiros onde o espaço é dividido de maneira não uniforme, visando reduzir o número de sub-árvores vazias, bem como a representação de *octrees* em texturas para possibilitar seu uso nas primeiras gerações de *hardware* gráfico programável, onde aplicações de propósito geral na GPU ainda estavam presas a programa de *shaders* de vértices e fragmentos.

Devido à natureza das placas gráficas atuais, onde recursos de memória são limitados e alocação dinâmica de memória está presente apenas em modelos mais novos, a escolha pela representação da *octree* em uma estrutura de tamanho fixo e sem o uso de ponteiros é a melhor opção.

[MADEIRA, 2010] propõe uma estrutura de representação de *octrees* em uma tabela *hash* para utilização em GPU. Nesta proposta, a chave de cada nó é definida por um código binário (Código de Morton) obtido a partir da posição geométrica do nó dentro da *octree* e o acesso aos filhos de um nó é feito através de cálculos feitos sobre o índice do pai. O acesso a um nó qualquer é feito em tempo constante, dado seu índice, porém o desempenho da recuperação é dependente da função de dispersão da tabela *hash* empregada.

Para realizar uma busca na tabela *hash*, deve-se aplicar uma função de dispersão que transforma a chave em um índice na tabela. Em um cenário ideal, cada chave é mapeada para um único índice, permitindo a busca em tempo *O(1)*. Caso aconteça uma colisão na criação da tabela, uma nova função de *hash* é aplicada para determinar onde o dado será armazenado. Esta função de *rehash* é aplicada na busca até que se encontre o valor equivalente à chave buscada. Assim, no pior caso o tempo de busca é *O(n)*.

2.2 RAY TRACING

O conceito do algoritmo de iluminação por *ray tracing* foi introduzido no final da década de 1960 por [APPEL, 1968]. A geração de imagens é reduzida a encontrar interseções entre raios disparados a partir de um ponto de visão com os objetos da cena e ao cálculo de iluminação nestes pontos [KUCHKUDA, 1988].

Para determinar a cor resultante da cada pixel na área de visualização, é preciso aplicar um modelo de iluminação nos pontos de interseção de cada raio. Este modelo é flexível o bastante para adicionar ou remover componentes conforme a necessidade.

Um dos modelos mais básicos de iluminação é o modelo de Phong [PHONG, 1975], descrito pela seguinte equação:

$$I = I_a + K_d \sum_{j=1}^{j=ls} (\vec{N} * \overrightarrow{L_j}) + K_s \sum_{j=1}^{j=ls} (\vec{N} * \overrightarrow{L'_j})^n, \text{ onde}$$

I = Intensidade da luz refletida (a cor do ponto amostrado na grade de visualização);

I_a = Intensidade de reflexão da luz ambiente;

K_d = Coeficiente de reflexão difusa;

$\vec{N}$ = Vetor unitário normal da superfície;

$\overrightarrow{L_j}$ = Vetor unitário de direção da j-ésima fonte de luz;

K_s = Coeficiente de reflexão especular;

$\overrightarrow{L'_j}$ = Vetor unitário na direção da bissetriz entre a fonte de luz e o observador;

n = Expoente representante do polimento da superfície.

Esta formulação foi estendida em diversos outros trabalhos a fim de acrescentar novos componentes ao algoritmo de iluminação. O modelo apresentado em [HALL e GREENBERG, 1983] estende a proposta de [WHITTED, 1980] e apresenta a seguinte equação para o cálculo de iluminação, combinando os componentes de reflexão difusa, especular e de transmissão tanto do objeto quanto globais:

$$I = K_d \sum_{j=1}^{l} (\vec{N} * \vec{L}) R_d I_j + Ks \sum_{j=1}^{j=l} (\vec{N} * \vec{H})^n R_f I_j + Ks \sum_{j=1}^{j=l} (\vec{N} * \overrightarrow{H'})^n T_f I_j + I_a R_d + K_s R_f I_r F_r^{dr} + K_s T_f I_t F_t^{dt}, \text{ onde}$$

I = Intensidade da luz refletida (a cor do ponto amostrado na grade de visualização);

K_d = Coeficiente de reflexão difusa;

l = Número de fontes de luz;

$\vec{N}$ = Vetor normal da superfície;

$\vec{L}$ = Vetor de direção da fonte de luz;

R_d = Curva difusa de reflectância do material baseada no comprimento de onda;

I_j = Intensidade da j-ésima fonte de luz;

K_s = Coeficiente de reflexão especular;

$\vec{H}$ = Vetor de direção espelhada baseado no raio refletido;

R_f = Curva de difração de Fresnel do material baseada no comprimento de onda;

$\overrightarrow{H'}$ = Vetor de direção espelhada baseado no raio transmitido;

n = Expoente representante do polimento da superfície.

T_f = Curva de transmissividade do material baseada no comprimento de onda;

I_a = Intensidade da luz ambiente;

I_r = Intensidade do raio refletido;

F_r = Transmitância por unidade do material refletido;

dr = Distância percorrida pelo raio refletido;

I_t = Intensidade do raio transmitido;

F_t = Transmitância por unidade do material transmitido.

dt = Distância percorrida pelo raio transmitido;

Uma vez que no ambiente existem infinitos raios de luz sendo emitidos por fontes luminosas, uma simulação deste tipo torna-se inviável. Habitualmente, em sistemas computacionais utiliza-se o *ray tracing* inverso, simulando o percurso dos raios de luz a partir do ponto de vista do usuário em direção à cena visualizada. A Figura 3 exemplifica este comportamento, onde:

O = Observador;

P = Pixel amostrado na área de visualização;

F = Fonte de luz;

S = Raio de sombra;

R = Raios refletidos;

T = Raio transmitido.

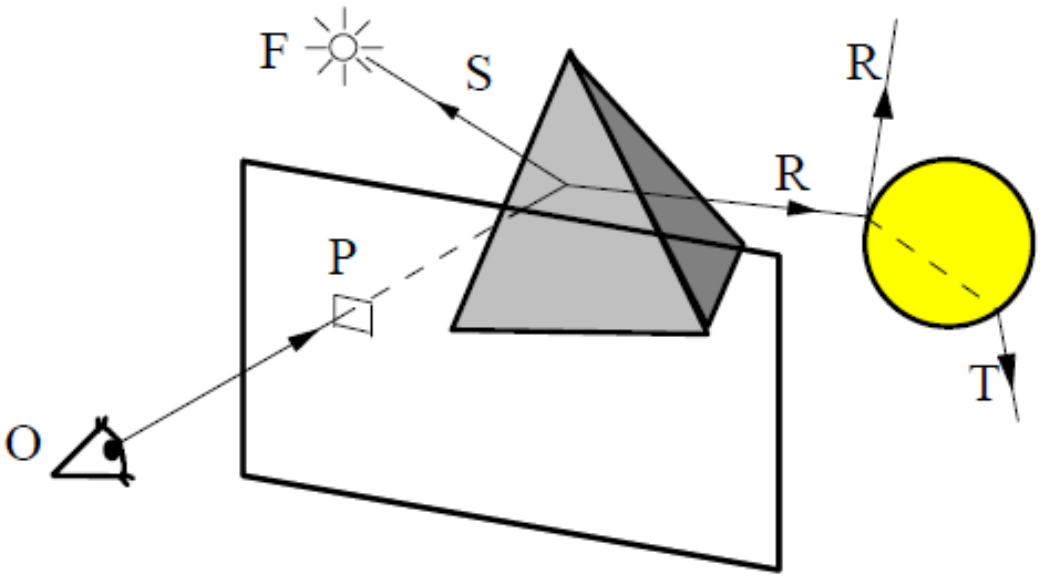

Figura 3: Visão esquemática dos componentes do *ray tracing* [SANTOS, 1994].

Conforme observado por [WHITTED, 1980], cerca de 75% do tempo de execução de seu algoritmo de *ray tracing* é gasto com o cálculo de interseção entre os raios e os objetos da cena. Este alto custo computacional é devido ao fato de que no *ray tracing* exaustivo todo raio

disparado em direção à cena deve ser testado com todos os objetos. Neste sistema, um acréscimo tanto no número de raios quanto no número de objetos é altamente custoso.

No *ray tracing* exaustivo, sem a utilização de uma estrutura de dados para acelerar o percorrimento da cena, a quantidade mínima *n* de cálculos de interseção é dada por $n = p * l * a$, onde *p* é a quantidade de primitivas da cena, *l* é a largura e *a* é a altura da janela de visualização. Esta quantidade é pertinente somente aos raios primários, portanto um número ainda maior de cálculos de interseções será feito conforme os raios secundários forem criados durante a execução do algoritmo.

Para reduzir este custo, empregam-se estruturas de dados para dividir os objetos em subconjuntos e assim reduzir o número de cálculos de interseção para cada raio. A seção anterior (2.1 – Octrees) descreve uma estrutura de dados hierárquica em árvore, empregada neste trabalho.

CAPÍTULO 3 – MODELAGEM DE DADOS

Este capítulo apresentada a modelagem dos dados elaborada para a execução do ray *tracing* em paralelo na GPU utilizando a *octree* como estrutura de aceleração. Esta modelagem tem como base a proposta apresentada por [MADEIRA, 2010], porém diversas adaptações e modificações tiveram que ser feitas para permitir seu emprego junto ao algoritmo de *ray tracing*.

Uma cena é descrita por um conjunto de objetos geométricos que podem ser visualizados por uma câmera (representando o observador) através da interação de sua geometria e das fontes de luz do ambiente. Neste trabalho, os objetos da cena podem ser representados de duas formas: através de malhas poligonais ou nuvens de pontos.

3.1 MALHA POLIGONAL

Uma malha poligonal é construída a partir de um conjunto de faces triangulares compostas por vértices no espaço tridimensional. Cada vértice é representado por um vetor com coordenadas XYZ e cada face possui três índices, indicando quais vértices a formam. Um mesmo vértice pode ser compartilhado por diversas faces vizinhas, assim, esta modelagem permite reduzir o consumo de memória para representação da cena. A Figura 4 ilustra esta relação entre vértices e faces. Nesta figura, são descritas três faces como exemplo: F0 = {V0, V1, V2}; F1 = {V2, V3, V4}; e F2 = {V3, V4, V5}. Como se pode observar, as faces F0 e F1 compartilham um vértice (V2), enquanto as faces F1 e F2 compartilham dois vértices (V3 e V4).

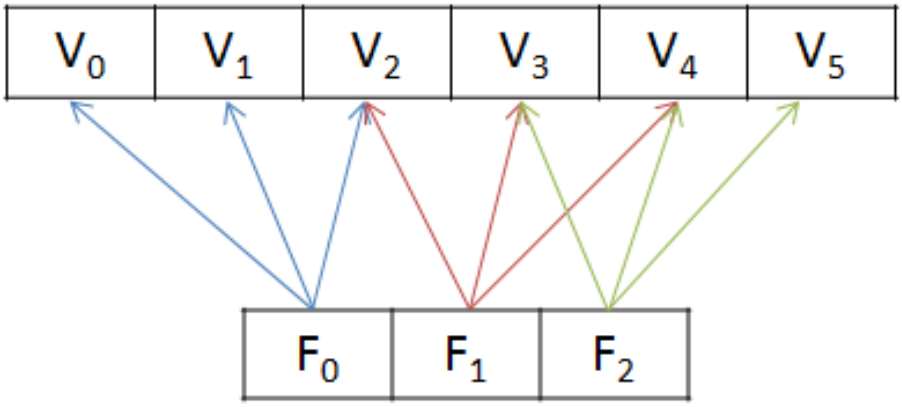

Figura 4: Ligação entre a lista de faces e de vértices.

O compartilhamento de vértices é uma abordagem particularmente importante quando se está trabalhando com dados em GPU, uma vez que estes dispositivos em geral possuem

uma quantidade de memória bastante restrita se comparada à memória principal do sistema. No exemplo anterior, três faces foram representadas por um total de seis vértices, ao contrário de nove que seria o usual. Isto representou uma economia de 1/3 de memória. Esta economia, de fato, é dependente da quantidade de vértices compartilhados entre as faces que compõem a malha tridimensional que se deseja exibir, algo que está intimamente ligado à maneira que o objeto foi modelado.

A representação da cena como um conjunto de faces apresenta um problema na construção da *octree*. Uma mesma face pode pertencer a um ou mais nós folha da *octree*, aumentando assim, a contagem total de polígonos contidos na árvore. Na sua implementação do algoritmo de busca, [MADEIRA, 2010] utiliza o baricentro do triângulo para determinar se ele está ou não contido em um nó da *octree*. Esta solução faz com que um triângulo sempre pertença a somente um nó.

Por se tratar de um sistema onde se pretende apenas determinar a existência ou não de um triângulo em uma determinada coordenada no espaço, sem qualquer menção a hierarquia, esta abordagem pode ser empregada sem problemas. No entanto, para o algoritmo de *ray tracing*, a decomposição hierárquica da cena é de grande importância. Ao se utilizar somente o baricentro do triângulo, descarta-se sua presença em nós adjacentes, fazendo com que raios que deveriam colidir com o objeto não o façam. A Figura 5 ilustra um caso de um triângulo armazenado em uma *quadtree*. Caso apenas o baricentro *C* fosse levado em conta, o triângulo composto pelos vértices {V1, V2, V3} pertenceria somente ao quadrante *Q3* da *octree* e nenhuma interseção seria encontrada quando o raio *R1* percorresse a cena.

Uma solução trivial seria considerar os vértices, ao invés do baricentro. Esta alternativa, no entanto, ainda levaria a resultados incorretos. Na figura, os vértices {V1, V2, V3} estão contidos, respectivamente, nos quadrantes {Q1, Q3, Q4}. No entanto, é possível observar que o triângulo também pertence ao quadrante *Q2*. Em qualquer situação onde um raio passasse somente por este quadrante, nenhuma interseção seria encontrada.

A solução, neste caso, é verificar se cada triângulo está contido, ainda que parcialmente, em cada nó da árvore [AKENINE-MOLLER, 2001]. Em caso positivo, o triângulo é adicionado à lista de faces do nó. Isto, porém, faz com que um mesmo triângulo possa aparecer replicado em vários nós da árvore, de certa forma aumentando o número total de faces da cena. No exemplo, um mesmo triângulo estaria presente em quatro nós.

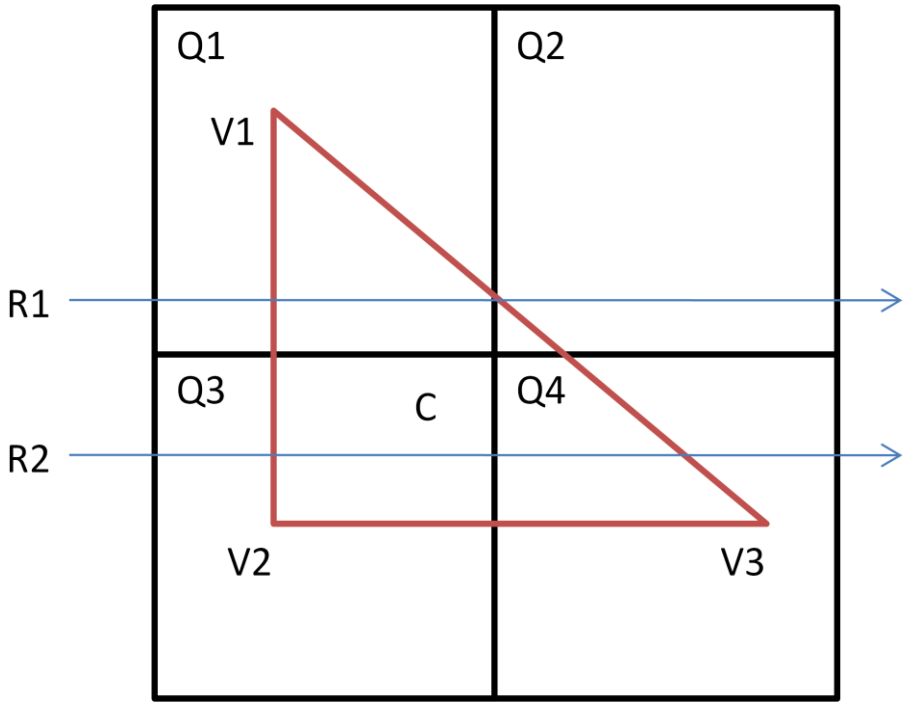

Figura 5: Representação de um triângulo armazenado em uma quadtree. Resultados incorretos são obtidos caso apenas o baricentro seja levado em conta para o cálculo de interseção.

3.2 NUVEM DE PONTOS

Uma nuvem de pontos é constituída por um conjunto de vértices com coordenadas XYZ no espaço 3D. Esta representação reduz a quantidade de dados armazenados na memória, uma vez que cada ponto passa a ser tratado individualmente e não mais como parte de uma face. Assim, apenas uma única lista de pontos (Figura 6) é necessária para representar toda a cena.

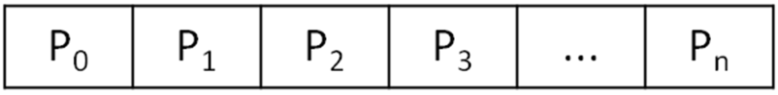

Figura 6: Lista de pontos utilizados na nuvem de pontos.

Para a construção da *octree*, considera-se que cada ponto representa uma esfera de raio *R* no espaço e verifica-se a interseção entre estas esferas e os nós da árvore. Neste caso, ainda existe o problema de uma mesma primitiva estar contida em mais de um nó-folha da *octree*. No entanto, estes dados podem ser tratados durante sua geração para impedir que isto ocorra. Para isto, a nuvem de pontos é gerada dentro de uma grade regular de três dimensões cujo tamanho *T* de cada *voxel* é sempre maior ou igual a 2*R* e o ponto contido no *voxel* está sempre centralizado (Figura 7). Por conseqüência, na construção da *octree* o tamanho de um nó folha não-vazio será igual a *T*.

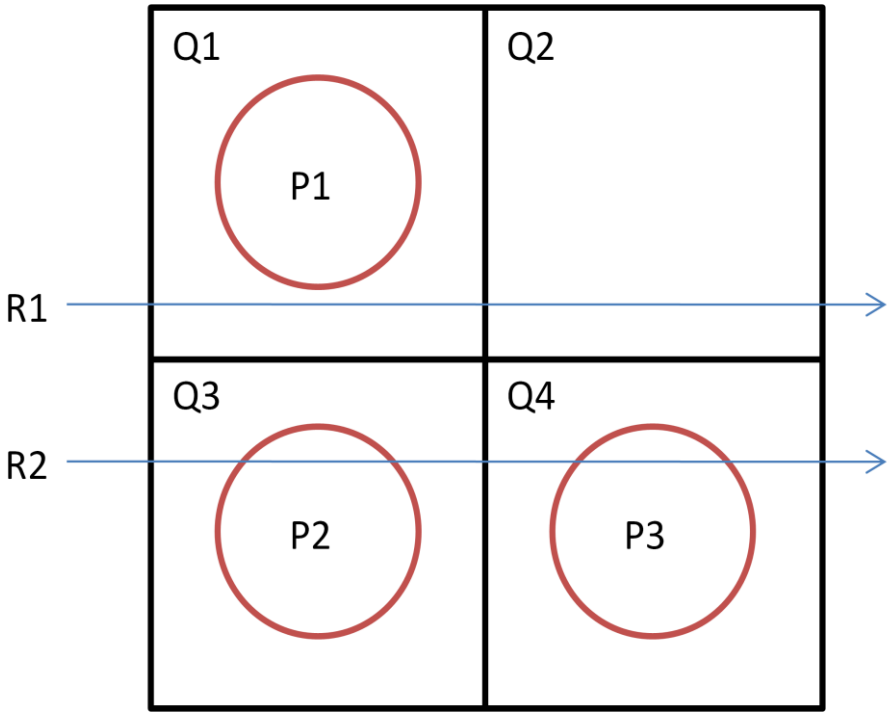

Figura 7: Representação da cena como uma nuvem de pontos em uma *quadtree*. Cada nó da árvore armazena um único ponto centralizado.

Esta construção foi proposta para fins de teste de desempenho da *octree*, eliminando-se ao máximo os problemas diversos que possam mascarar o ganho do emprego desta estrutura em função de problemas de representação.

Algumas implementações de *ray tracing* com nuvens de pontos estão disponíveis em [LINSEN *et al.*, 2007] e [DEUL *et al*, 2010].

3.3 REPRESENTAÇÃO DA CENA

Neste trabalho, deseja-se representar a cena em uma *octree* a fim de reduzir o número total de cálculos de interseção feitos por cada raio que percorre a cena. Nesta árvore são executadas duas operações básicas: a construção (realizada previamente, junto ao carregamento da cena) e a consulta (realizada enquanto um raio percorre a cena). As seções a seguir descrevem como estas operações são realizadas e como os dados são representados na *octree*.

3.3.1 CONSTRUÇÃO DA OCTREE

A construção da *octree* é feita através de um único passo recursivo: a partir de um conjunto de dados da cena, verifica-se se ele está inserido no nó atual da árvore. Em caso positivo, este nó é subdividido em oito filhos e para cada filho resultante verifica-se se há interseção com a cena. O algoritmo pára ao se atingir o nível máximo pré-determinado para a

árvore, o número máximo de primitivas por folha ou ao não se encontrar interseções entre o nó corrente e as primitivas.

Um nó que não possua filhos e tenha uma lista de primitivas não-vazia é chamado folha. Um nó folha cuja lista de primitivas é vazia é chamado folha-vazia e os nós intermediários são chamados de nós internos.

O algoritmo para a construção da árvore é descrito a seguir:

Procedimento: CriaÁrvore

Entrada: nóRaiz, nívelAtual, primitivas
nó = ClassificaNó(primitivas, nóRaiz.min, nóRaiz.max)
se nó == FOLHA **então**
    nóRaiz = FOLHA
senão se nó == FOLHA_VAZIA **então**
    nóRaiz = FOLHA_VAZIA
senão se no == INTERNO **então**
    nóRaíz = INTERNO
    SubdivideNó(raiz)
    Para cada filho F de raiz **faça**
        CriaÁrvore(F, nívelAtual + 1, nóRaiz.primitivas)

Para o *ray tracing*, o que se deseja é percorrer a árvore a partir da raiz, passando pelos nós internos até se encontrar as folhas. Ao se atingir uma folha, os cálculos de interseção entre o raio e as primitivas da folha são feitos, porém nenhum cálculo teve de ser feito com o restante da cena. Além disso, ao se encontrar uma folha-vazia, sabe-se que não há primitivas nos níveis subseqüentes da árvore e, portanto, o raio não precisa continuar seu caminho naquela direção.

3.3.2 REPRESENTAÇÃO DOS DADOS DA OCTREE

Conforme mencionado na seção 2.1 - Octrees, existem diversas maneiras de se representar os nós de uma *octree*. Entre elas, a representação por ponteiros, indexação em uma estrutura linear e a utilização de tabelas *hash*. Neste trabalho foram implementadas tanto a representação linear quanto a representação com tabela *hash*, a fim de avaliar qual apresenta o melhor desempenho. Independente do tipo de representação, a construção da árvore é feita

da mesma forma. A única exceção é a função de dispersão, usada para determinar a posição de memória ocupada por cada nó da *octree*.

Cada nó da árvore é uma estrutura composta pelas seguintes informações:

- Uma caixa envolvente que delimita sua região no espaço;
- Um indicador que determina em qual nível da árvore o nó está localizado;
- Um indicador do tipo do nó (interno, folha ou folha-vazia);
- Uma lista de primitivas contida no nó;
- Um índice do nó dentro da estrutura que armazena a árvore.

3.3.2.1 REPRESENTAÇÃO EM TABELA HASH

Uma tabela *hash* é uma estrutura de dados que utiliza uma função de dispersão para mapear chaves em posições dentro da tabela. Dada uma chave qualquer, a função de dispersão é aplicada para determinar em qual posição da tabela o dado referente à chave em questão se encontra.

De acordo com o volume de dados a ser armazenado na tabela e à função de dispersão empregada, mais de uma chave pode ser mapeada em uma mesma posição. Neste caso, uma função de tratamento de colisão deve ser empregada. Existem duas formas de se resolver problemas de colisão: através de encadeamento aberto ou fechado.

O tratamento de colisões com encadeamento aberto emprega uma lista onde são armazenados os dados com chaves mapeadas para a mesma posição na tabela. Assim sendo, quando ocorre uma colisão, o novo dado é inserido no final da lista na posição da tabela correspondente à sua chave. Enquanto na CPU (Central Processing Unit), em geral se resolve este tipo de colisão com uma lista encadeada simples, na GPU é preciso alocar uma lista seqüencial de tamanho fixo, elevando o consumo de memória.

Assim, a tabela *hash* passa a ser representada como uma espécie de matriz, onde a primeira linha é acessada ao se aplicar a função de dispersão na chave desejada e as linhas seguintes (da coluna encontrada pela função de dispersão) são percorridas até que se encontre o dado cuja chave é igual à chave buscada. No caso da inserção, a lista é percorrida até se encontrar a primeira posição vazia, local este onde o dado será inserido. A Figura 8 exemplifica esta representação, onde um nó cuja chave tenha sido mapeada para a posição C0 é inserido na linha V0 da coluna C0. O próximo nó a ser mapeado nesta coluna é armazenado na linha V1 e assim em diante.

C_0	C_1	C_2	C_3	...	C_n
V0	V0	V0	V0	...	V0
V1	V1	V1	V1	...	V1
V2	V2	V2	V2	...	V2
V3	V3	V3	V3	...	V3
V4	V4	V4	V4	...	V4

Figura 8: Representação da tabela hash com encadeamento aberto através de uma lista linear.

Há ainda a opção de se utilizar o encadeamento fechado. Neste caso, quando ocorre uma colisão uma segunda função de dispersão é aplicada para determinar a posição da chave na tabela. A busca, então, consiste em aplicar a função de dispersão e verificar se a chave do dado encontrado é igual à chave buscada. Em caso positivo, retorna-se o dado. Caso contrário, a nova função de dispersão é aplicada e repete-se o processo até encontrar o valor buscado. A inserção é feita de forma análoga, enquanto a função de dispersão retornar uma posição ocupada na tabela, uma nova função é aplicada para continuar a busca por uma posição vaga.

[MADEIRA, 2010] utiliza como primeira de função de dispersão uma operação de módulo, dada por $h(k) = k \bmod m$, onde m é o número de índices da tabela *hash* e k a chave do dado inserido ou buscado. Como segunda função de dispersão para o tratamento de colisões, é utilizada a função $h'(k) = h(k) + 1$. Assim, quando ocorre uma colisão o dado é armazenado na primeira posição livre na tabela. Isto, no entanto, pode elevar o tempo de busca para $O(n)$, uma vez que no pior caso todas as posições da tabela teriam de ser testadas antes de se encontrar a chave buscada.

Neste trabalho optou-se por manter a função de dispersão como sendo a operação de módulo. No entanto, o encadeamento aberto com uma lista seqüencial de tamanho fixo foi empregado para o tratamento de colisões. Assim, cada célula da tabela *hash* contém um vetor com os nós da *octree* ali inseridos.

[CASTRO, 2008] e [MADEIRA, 2010] utilizam o Código de Morton [MORTON, 1966] para codificar as chaves dos nós da *octree*. A raiz da árvore tem chave igual a 1 e a chave de cada nó *n* é gerada através da concatenação da chave do seu pai com os três bits

correspondentes ao seu octante, conforme demonstrado na Figura 9. A chave do pai de um nó pode ser obtida fazendo o caminho inverso, isto é, truncando os três bits menos significativos da chave do filho. A Figura 10 exemplifica em uma *quadtree* como ficam representadas as chaves geradas pelo Código de Morton.

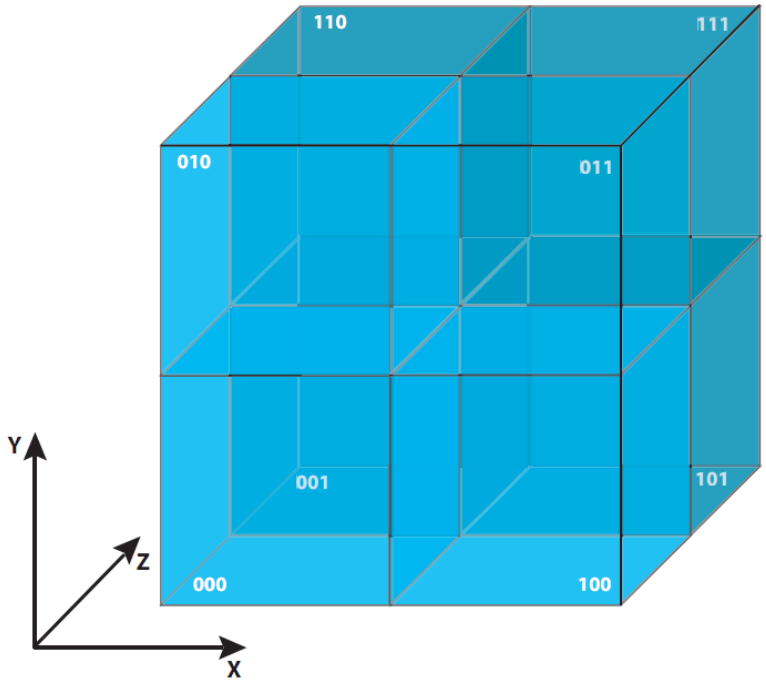

Figura 9: Sufixos concatenados à chave do pai para obter a chave dos filhos [CASTRO, 2008].

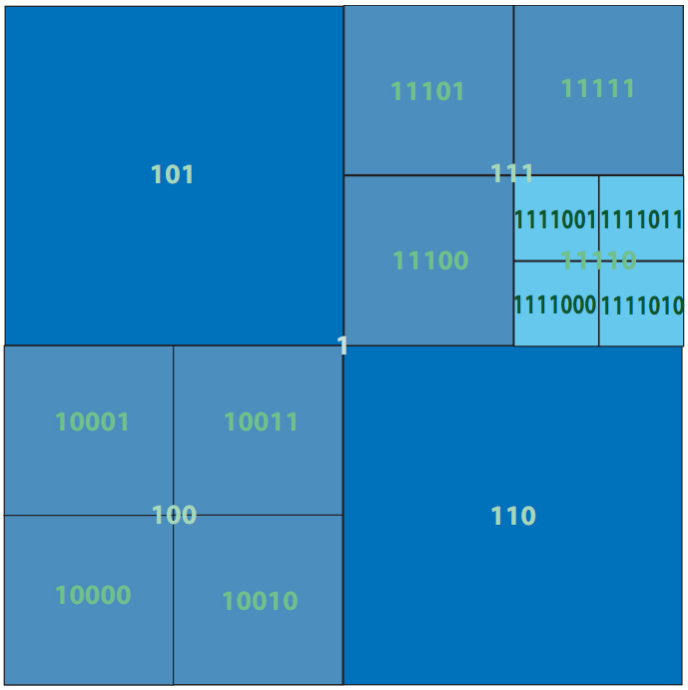

Figura 10: *Quadtree* com o Código de Morton dos filhos gerados a partir do código do pai [CASTRO, 2008].

3.3.2.2 REPRESENTAÇÃO EM LISTA LINEAR

Uma *octree* pode ser representada em uma lista linear através da relação entre o índice de um nó com os índices de seus filhos. Neste modelo, a raiz da árvore é alocada na posição zero do vetor e a posição *p* de cada filho é dada por *p = 8 * índiceDoPai + índiceDoFilho*, onde para cada filho é atribuído um índice variando entre 1 e 8. Desta forma, os filhos de um dado nó sempre são alocados em posições adjacentes de memória.

A Figura 11 exemplifica esta estrutura. Os filhos do nó-raiz são armazenados nas posições de 1 a 8. O primeiro filho, o nó 1, tem seus filhos listados entre as posições 9 (8 * 1 + 1) e 16 (8 * 1 + 8). A posição dos filhos dos nós restantes é encontrada de forma análoga.

0	1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16	17

Figura 11: Esquema de armazenamento de uma *octree* em uma lista linear.

3.3.3 ARMAZENAMENTO DOS DADOS DA OCTREE NA GPU

Foram definidas algumas estruturas para armazenar os dados da *octree* na GPU, possibilitando sua consulta em paralelo durante a execução do algoritmo de *ray tracing*.

A estrutura *HashItem* armazena os dados relativos aos nós da *octree* na tabela *hash*. Esta estrutura contém uma chave, um índice para uma lista de primitivas, o tipo do nó, o nível do nó e as coordenadas e tamanho de sua caixa envolvente. O tipo do nó é dado por um tipo enumerado que pode assumir quatro valores: folha (LEAF), folha vazia (EMPTY_LEAF), interno (INTERNAL) ou inválido (INVALID, significando que aquela posição na tabela não é usada por nenhum nó).

```
struct HashItem
{
	unsigned int key;
	int primitiveList;
	NodeType nodeType;
	int level;
	float boundingBoxX;
	float boundingBoxY;
	float boundingBoxZ;
	float boundingBoxSize;
};
```

```
enum NodeType
{
	LEAF,
	EMPTY_LEAF,
	INTERNAL,
	INVALID
};
```

As primitivas são armazenadas em uma lista externa, fora da *octree*. Os nós da *octree* são utilizados apenas para indexar as primitivas contidas em si. Esta abordagem foi utilizada por dois motivos: armazenar as primitivas em uma lista externa impede que elas tenham que ser replicadas para cada nó, gerando assim uma economia da memória; e as primitivas foram modeladas de forma a aproveitar a arquitetura do CUDA (Computer Unified Device Architecture) [NVIDIA, 2011] para otimizar o acesso aos dados através do uso da memória de textura.

Quando tratamos a cena como uma malha poligonal, os dados são armazenados em duas texturas distintas, conforme mostrado na Figura 4. Uma textura é utilizada para armazenar a lista de vértices e outra para a lista de faces. A consulta é feita em dois passos, primeiro é preciso acessar a posição desejada na lista de faces. As coordenadas retornadas nesta busca são os índices dos vértices que compõem a face. De posse destes índices, basta buscar os vértices na outra textura e assim encontrar suas coordenadas na cena.

A representação da nuvem de pontos utiliza somente uma textura para armazenar a lista de vértices. O acesso neste caso é direto às coordenadas do vértice desejado, bastando saber apenas seu índice na lista.

A estrutura da *octree* não indexa diretamente as primitivas contidas em cada nó. A implementação desta forma traria um alto desperdício de espaço, uma vez que teria que ser alocado um espaço grande o suficiente para armazenar o tamanho do vetor da *octree* multiplicado pelo número de faces máxima de cada nó. Visto que a *octree* é composta por um grande número de nós internos e folhas vazias, buscou-se uma maneira de alocar espaço para a lista de primitivas somente para nós folhas.

Cada nó possui um índice (*primitiveList*) indicando a posição de memória onde está armazenada sua lista de primitivas. Nós internos e folhas vazias armazenam o valor *-1* neste campo, indicando que não há primitivas relacionadas àquele nó. Por outro lado, as folhas sempre terão um valor diferente de *-1*.

O número máximo de primitivas permitido numa folha é usado para alocar uma quantidade equivalente de vetores na memória global. Colocados um abaixo do outro, estes vetores funcionam como uma matriz, onde a coluna equivale ao índice *primitiveList* de seu nó e as linhas são os índices das primitivas em si.

Um determinado nó folha pode ter uma ou mais primitivas. Caso o nó não ocupe todas as primitivas alocadas para si, um valor *-1* é usado para indicar que o espaço daquela primitiva não está em uso. Pelo fato da arquitetura de GPUs utilizada neste trabalho não permitir alocação dinâmica, há ainda um desperdício de memória, porém essa representação reduz o desperdício somente às folhas não-cheias.

Na Figura 12 é apresentada uma visão de como esta estrutura funciona. A primeira linha numerada (a) representa os nós da *octree*. Neste caso supomos uma árvore linear com dois níveis somente. A linha seguinte (b) armazena o índice da lista de primitivas relativas ao nó. Nota-se que os nós {0, 1, 3, 5, 8} são internos ou folhas vazias, visto que sua lista de primitivas aponta para um valor inválido. Os nós {2, 4, 6, 7} são folhas e para cada um foi atribuído de forma seqüencial uma coluna da lista de primitivas. Assim, eles apontam respectivamente para {0, 1, 2, 3}.

Para se consultar as primitivas do nó 4, por exemplo, primeiramente seria lido seu índice na linha *b*, de onde se obteria o resultado 1. A seguir, a posição 1 de cada lista de primitivas *c* seria consultada e cada valor diferente de *-1* encontrado indicaria o índice na lista de primitivas do nó 4.

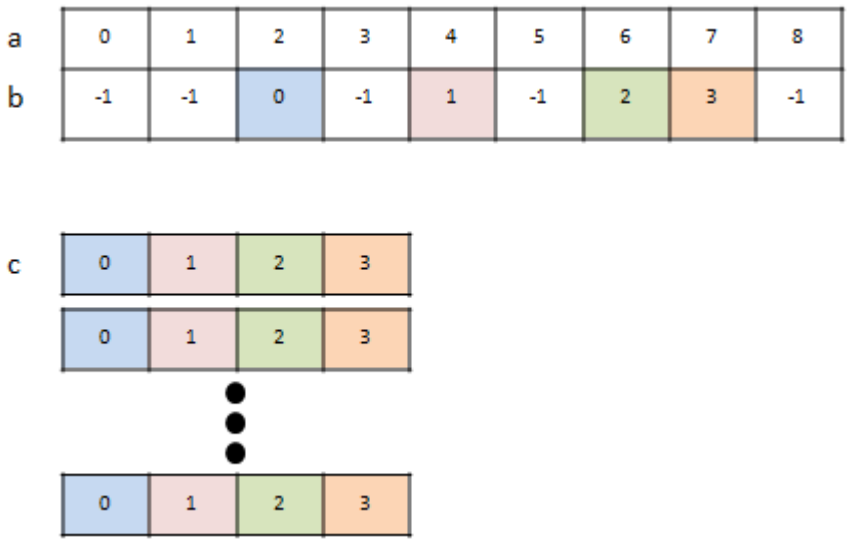

Figura 12: Representação da lista de primitivas na memória da GPU.

Apesar do espaço adicional necessário para o armazenamento do índice para esta lista, a representação da árvore desta forma resulta em uma representativa economia de memória. No pior caso, uma *octree* cheia tem o número de nós folha igual a 8^h, onde *h* é a altura da árvore com a raiz tendo altura zero. Esta mesma árvore cheia tem um número total de nós igual a $8^h + 8^{h-1}$, onde 8^{h-1} representa seus nós internos. Tomando como exemplo uma árvore cheia de nível máximo igual a sete, temos:

Nós	Total
Internos	262.144
Folhas	2.097.152
Árvore cheia	2.359.296

Tabela 1: Quantidade de nós em uma árvore cheia.

Digamos agora que cada nó guarda um vetor de seis posições para indexar suas primitivas. Neste caso, o consumo total de memória é igual ao número de nós da árvore cheia (2.359.296) vezes a quantidade de índices por nó (6). Assim, chegamos ao valor total de 14.155.776.

Caso a abordagem descrita anteriormente seja usada, temos o consumo de memória dado pelo tamanho da árvore cheia (2.359.296) mais o número de folhas (2.097.152) vezes a quantidade de índices por nó (6). O resultado é 2.359.296 + 12.582.912 = 14.942.208, ou seja, no pior caso (árvore cheia), o consumo total de memória é maior. No entanto, em casos onde a árvore não seja cheia, obtém-se uma considerável economia de memória.

Utilizando um caso real, um modelo com 64 faces foi armazenado em uma octree com altura igual a 3. Foram 64 folhas, 56 folhas vazias e 17 nós internos. O consumo de memória desta árvore é dado por tamanho da árvore cheia (512) mais número de folhas vezes 6 (384) igual a 896. A abordagem inicial totalizaria 3.072 índices (tamanho da árvore cheia vezes 6).

CAPÍTULO 4 – APLICAÇÃO

Este capítulo descreve a aplicação da *octree* junto ao algoritmo de *ray tracing* em paralelo na GPU. Aqui são apresentados detalhes sobre a implementação e os algoritmos utilizados.

4.1.1 FLUXO DE EXECUÇÃO

O sistema desenvolvido obedece ao fluxo de execução definido na Figura 13. Inicialmente, a *octree* é construída e carregada na GPU. Esta estrutura é armazenada na memória da GPU de forma permanente durante a execução do programa, sendo removida somente ao término da execução. Os passos seguintes são repetidos enquanto o sistema estiver sendo executado. As configurações de visualização correntes (isto é, coordenadas da câmera e os parâmetros das fontes de luz) são enviadas para a GPU e o *kernel* de execução do *ray tracing* em paralelo é disparado. Ao final da execução do *kernel*, a imagem gerada pelo *ray tracing* deixa de ser utilizada pelo CUDA e o OpenGL a exibe na tela. Enfim, as coordenadas da câmera e das fontes de luz são atualizadas e o ciclo se repete. Cada uma das etapas é descrita nas seções a seguir.

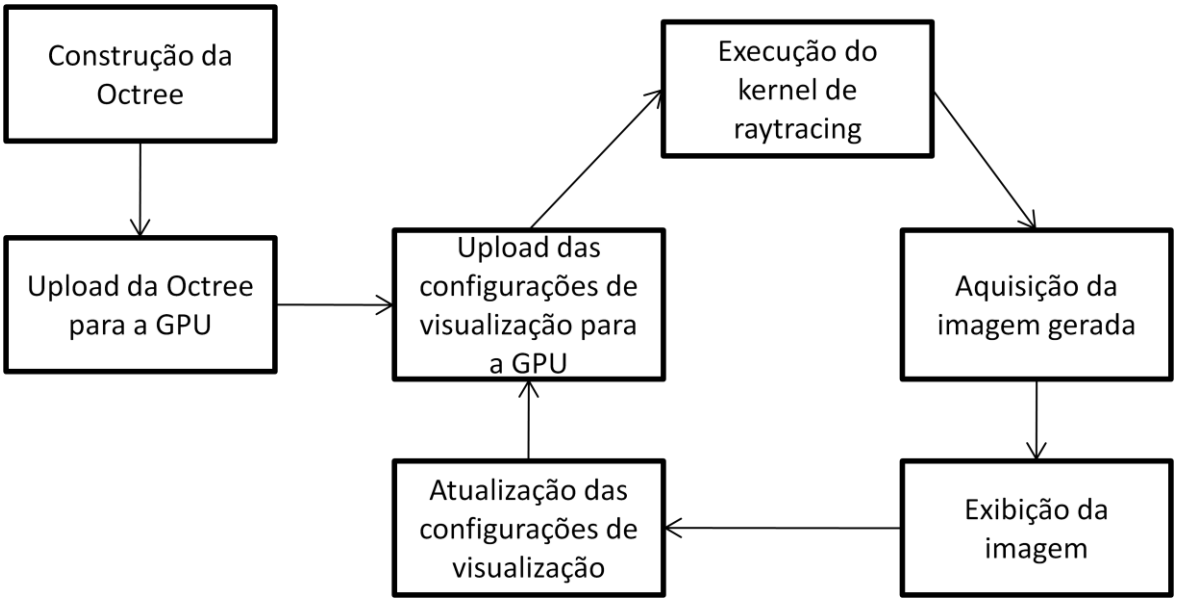

Figura 13: Fluxo de execução do sistema.

4.1.2 CONSTRUÇÃO DA OCTREE

A construção da *octree* é realizada na CPU e este processo é detalhado na seção 3.3.1 - Construção da Octree. Resumidamente, uma caixa envolvente é criada ao redor da cena e é

subdividida recursivamente até se encontrar as folhas ou que se alcance o nível máximo permitido. As folhas contêm os dados de geometria da cena.

4.1.3 ENVIO DA OCTREE PARA A GPU

Para ser acessada pelo algoritmo de *ray tracing*, a *octree* deve primeiramente ser carregada na memória da GPU. De acordo com as representações utilizadas neste trabalho, os dados de geometria da cena são armazenados separadamente da *octree* e esta apenas contém índices para referenciar as primitivas contidas em seus nós.

As primitivas da cena são armazenadas na memória da GPU como uma textura 1D. Para a nuvem de pontos, cada pixel desta textura equivale a um ponto na nuvem. No caso da malha poligonal, duas texturas são empregadas. Cada ponto na primeira textura representa uma face, cujas coordenadas indicam os índices dos vértices que a compõem. A busca por uma primitiva para o cálculo de interseção é feita da seguinte forma:

- Se cena representada por nuvem de pontos:
  - Busca as coordenadas do ponto na textura a partir de um índice;
- Se cena representada por malha poligonal:
  - Busca índices dos pontos na textura de faces;
  - Busca as coordenadas dos pontos que compõem a face na textura de pontos, a partir dos índices obtidos no passo anterior.

A *octree* é armazenada em uma tabela *hash*, conforme descrito na seção 3.3.3 - Armazenamento dos Dados da Octree na GPU. Estes dados são armazenados na memória global da GPU e ficam disponíveis para acesso por todas as *threads*. Uma região de memória na GPU é alocada para armazenar os dados da tabela *hash* e a seguir estes dados são copiados. O processo de transferência de dados entre a GPU e a memória principal é lento e limitado pela largura de banda da arquitetura utilizada (algo altamente dependente do hardware). Como No entanto, como esta cópia é executada uma única vez, não há grande perda de desempenho.

4.1.4 ENVIO DAS CONFIGURAÇÕES DE VISUALIZAÇÃO PARA A GPU

A cada quadro, as configurações de visualização devem ser enviadas para a GPU para representar possíveis modificações feitas na cena. Essas configurações são enviadas como entrada do algoritmo de *ray tracing* e incluem as coordenadas da câmera e das fontes de luz.

Depois que todo o processo do *ray tracing* é executado, os dados de visualização são atualizados na CPU para que no quadro seguinte os parâmetros de entrada do algoritmo sejam condizentes com as configurações correntes de visualização da cena.

4.1.5 EXECUÇÃO DO RAY TRACING

Uma *thread* de GPU é disparada para cada raio do algoritmo de *ray tracing*. Cada *thread* é executada em paralelo de acordo com a quantidade de processadores de fluxo disponíveis na placa gráfica e seu resultado é a cor do *pixel* correspondente. Para determinar esta cor, o raio é testado com as primitivas da cena a fim de verificar aquela que se encontra mais próxima à câmera em sua linha de visão, isto é, aquela que será vista pelo observador. Uma vez que esta primitiva tenha sido encontrada, aplica-se uma equação de iluminação para determinar sua cor naquele ponto.

O algoritmo de *ray tracing* é descrito a seguir:

Procedimento: TraçaRaio

---

para cada pixel **faça**

    CalculaDireçãoDoRaio()

    registroDeColisão = PercorreCena()

    corDoPixel = CalculaIluminação(registroDeColisão)

O passo inicial (*CalculaDireçãoDoRaio*) determina a direção do raio que partindo do ponto de visualização da cena deve passar pelo pixel amostrado. Para cada *thread*, é atribuído um *pixel* na região de visualização representando a origem do raio que será disparado em direção à cena.

A cena é então percorrida por cada raio com o objetivo de encontrar o ponto que será tonalizado. Aqui é empregado um registro de colisão que pode armazenar um conjunto de dados necessário para esta avaliação, como a menor distância de interseção entre o raio e uma primitiva e os índices da primitiva nas listas em memória. Este percorrimento pode ser alterado de acordo com a necessidade, empregando-se desde o percorrimento linear de toda a cena até a utilização de estruturas de aceleração, como a *octree* utilizada neste trabalho.

Com o resultado do percorrimento da cena, obtém-se um registro de colisão indicando a primitiva que foi atingida por um determinado raio e sua distância em relação ao observador. Uma vez que se determine o ponto de interseção do raio com a cena, deve-se

aplicar uma equação de iluminação para calcular a cor do pixel naquele ponto. A seção 2.2 descreve algumas equações que podem ser empregadas neste cálculo.

4.1.5.1 ACELERAÇÃO DO CÁLCULO DE INTERSEÇÃO

A fim de reduzir a quantidade de operações necessária para determinar quais primitivas da cena foram atravessadas pelos raios do *ray tracing*, uma estrutura de aceleração é empregada. Esta estrutura, descrita na seção 2.1 - Octrees, permite particionar a cena de acordo com a distribuição espacial das primitivas e realizar testes de interseção dos raios apenas com um pequeno subconjunto das primitivas que compõem a cena.

Esta estrutura de aceleração é empregada na etapa de cálculo de interseção do *ray tracing*. Na solução trivial, toda a cena é percorrida e cada raio é testado com cada primitiva. Com a utilização da *octree*, este teste é substituído pelo seguinte algoritmo:

Procedimento: PercorreOctree

Entrada: raio, nóAtual
se HáInterseção(raio, nóAtual) **então**
    se nóAtual.tipo == INTERNO **então**
        para cada filho F de nóAtual **faça**
            PercorreOctree(raio, F)
    senão se nóAtual.tipo == FOLHA **então**
        para cada primitiva P de nóAtual **faça**
            se HáInterseção(raio, P) **então**
                d = Distância(raio, P)
                menorDistância = min(menorDistância, d)
            retorna menorDistância
    senão se nóAtual.tipo == FOLHA_VAZIA **então**
        retorna

Neste algoritmo, começando pela raiz da árvore, verifica-se se há interseção entre o raio e o nó corrente. Em caso positivo, verifica-se o tipo do nó. Para nós internos, o algoritmo é aplicado recursivamente nos filhos. Nós vazios retornam imediatamente, encerrando o percorrimento naquela sub-árvore vazia. Enfim, quando uma folha é encontrada, procura-se entre as primitivas contidas no nó aquela cujo ponto de interseção com o raio tem a menor distância em relação ao ponto de visualização (o ponto de visualização é igual à origem do

raio). A primitiva encontrada é usada na etapa seguinte do *ray tracing*, o cálculo de iluminação.

4.1.6 AQUISIÇÃO E EXIBIÇÃO DA IMAGEM GERADA

Antes de iniciar o ray tracing, um buffer do OpengGL é criado na memória de vídeo. Este buffer (PBO – Pixel Buffer Object) é mapeado para poder ser utilizado pelo CUDA. Cada raio disparado no *ray tracing* escreve em uma posição de memória especifica deste buffer a cor do *pixel* que foi calculada.

Ao término do cálculo do *ray tracing*, uma imagem representando a visualização da cena foi gerada, com base na cor que cada raio calculou para seu pixel correspondente. O *pixel buffer* previamente mapeado para o CUDA é liberado e pode ser novamente utilizado pelo OpenGL. Para exibir na tela o resultado do *ray tracing*, este *buffer*, que nada mais é do que a representação de uma textura bidimensional, é mapeado em um quadrado de tamanho igual à janela de visualização.

Assim, todo o cálculo do *ray tracing* pode ser efetuado diretamente na GPU, sem a necessidade de que dados sejam transferidos entre a memória de vídeo e a memória principal, evitando gargalos no tráfego desses dados entre a GPU e a CPU.

CAPÍTULO 5 – RESULTADOS

Este capítulo apresenta os testes realizados utilizando as implementações de *ray tracing* em GPU desenvolvidas neste trabalho. Todos os testes foram realizados com uma placa de vídeo GeForce 9800 GT, com 1 GB de memória e 112 processadores de fluxos rodando a 1350 MHz.

Foram realizadas duas classes de testes diferentes: com uma nuvem de pontos gerados aleatoriamente e com modelos tridimensionais carregados de um arquivo. Em ambos os casos, grupos com resoluções variadas foram testados a fim de medir o desempenho do algoritmo para diferentes casos.

Três implementações distintas do algoritmo de *ray tracing* foram utilizadas nos testes. O primeiro teste é feito com o *ray tracing* exaustivo, onde todos os raios são testados com todos os objetos da cena. A seguir, é testado o *ray tracing* com a *octree* representada na tabela *hash* utilizando o Código de Morton para a busca dos filhos de um nó e por fim o *ray tracing* na *octree* linear.

Todas as imagens geradas pelo algoritmo de *ray tracing* foram na resolução 512 por 512 *pixels* e apenas os raios primários foram levados em conta nos testes.

Cada algoritmo foi executado entre cinco e dez vezes para cada conjunto de dados. Foram feitas cerca de cem medições de tempo para se encontrar o menor, o maior e a média de tempo de execução em cada teste.

Os tempos medidos são apresentados em todas as tabelas em segundos.

5.1.1 TESTES COM NUVEM DE PONTOS

Para os testes com uma nuvem de pontos foram geradas nuvens com uma distribuição aleatória dos pontos. Por questões de consumo de memória, alguns testes com nuvens de pontos muito grandes não puderam ser feitas nas representações da cena em árvore. Um exemplo de nuvem de pontos testada é apresentado na Figura 14.

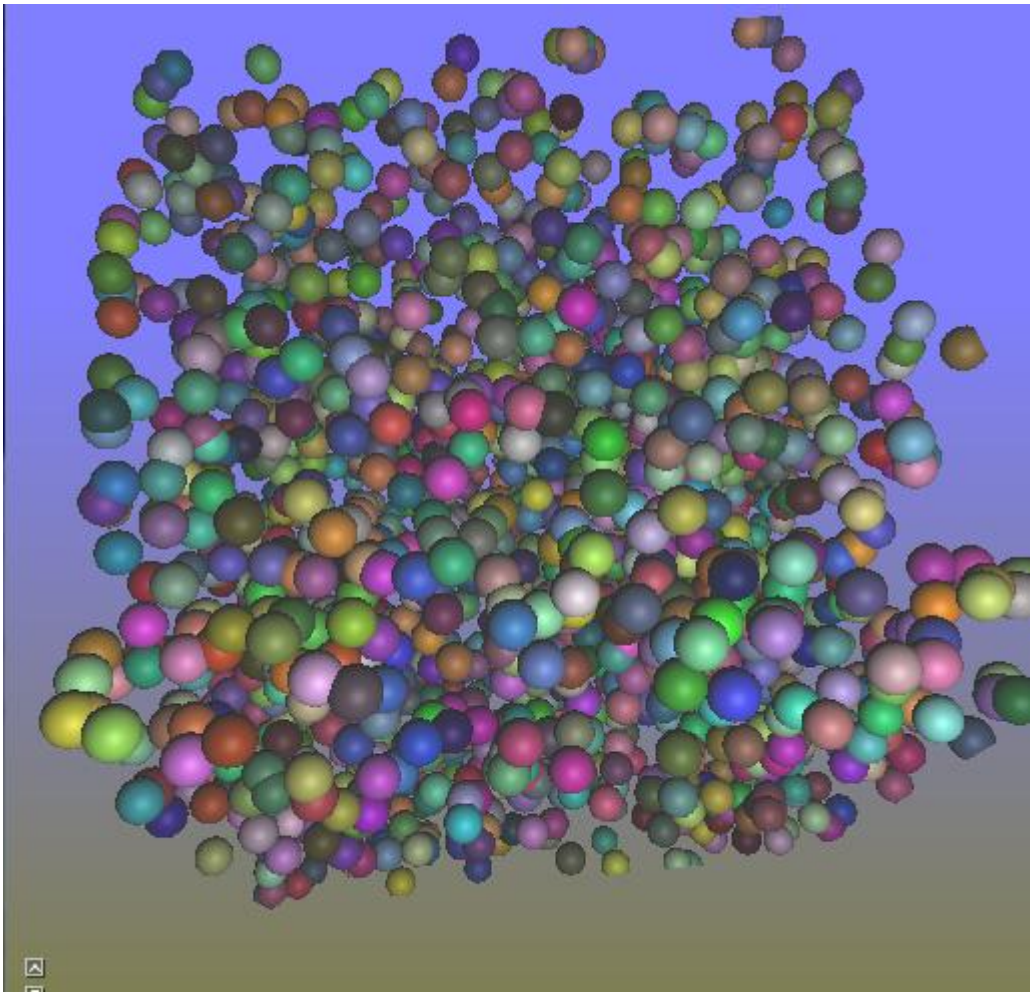

Figura 14: Nuvem de 2000 pontos utilizada nos testes.

Os testes com o *ray tracing* exaustivo foram realizados variando a quantidade de pontos entre 1000 e 64000. A Tabela 2 lista os resultados (nas linhas) mínimo, médio e máximo obtidos, em segundos para cada quantidade de pontos (colunas). Nota-se na Figura 15 que há um grande crescimento no tempo gasto pelo algoritmo conforme a quantidade de dados aumenta. Este crescimento, como pode ser observado em um intervalo qualquer entre duas medições, é linear.

Pontos	Mínimo	Média	Maximo
1000	0,017	0,0182	0,0217
2000	0,0307	0,0327	0,0367
4000	0,0579	0,0609	0,064
8000	0,114	0,1202	0,1266
16000	0,2305	0,2423	0,2545
32000	0,4572	0,4802	0,505
64000	0,9123	0,956	1,002

Tabela 2: Resultados dos testes com a nuvem de pontos e o *ray tracing* exaustivo. Percebe-se que o tempo de processamento dobra juntamente com a quantidade de pontos (tempo em segundos).

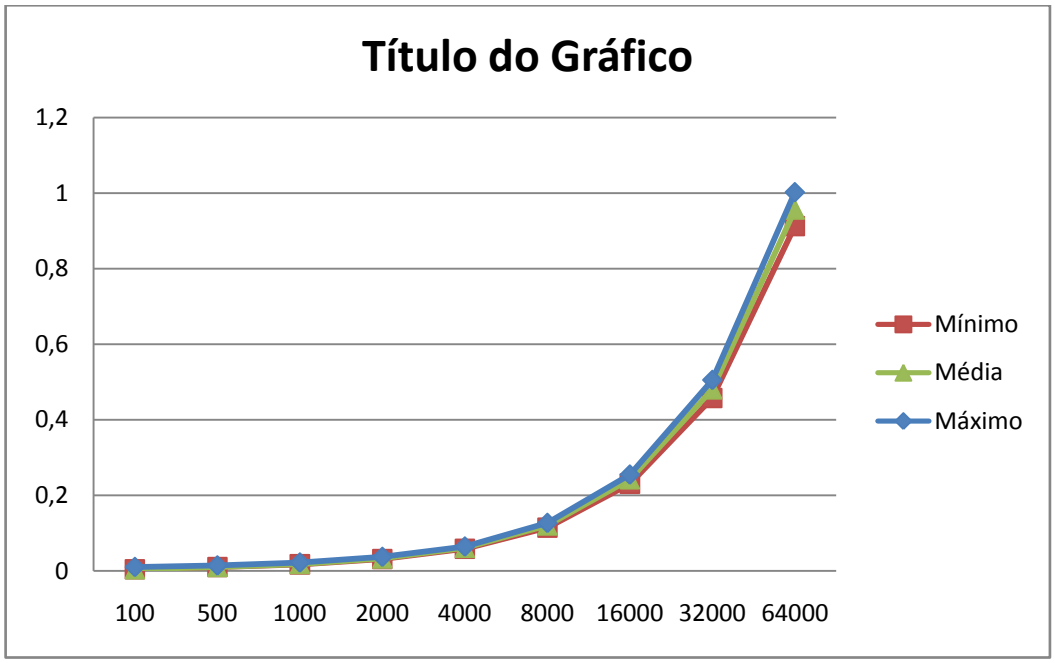

Figura 15: Gráfico detalhando o comportamento do *ray tracing* exaustivo com a nuvem de pontos (tempo em segundos).

A seguir, foram feitos testes com o *ray tracing* utilizando a *octree*. Neste primeiro caso, a *octree* está representada em uma tabela *hash e* foram testadas nuvens de pontos variando entre 1000 e 32000 vértices.

Na Tabela 3 estão listados os tempos obtidos na execução do algoritmo variando a quantidade de pontos e limitando o número máximo de primitivas nas folhas em 1. Esta limitação faz com que a árvore fique mais profunda e em conseqüência disso o percorrimento passa por uma quantidade maior de nós internos. O crescimento neste cenário não é tão acentuado e o tempo não dobra mais à medida que a quantidade de pontos o faz, no entanto o desempenho do algoritmo ainda é inferior ao *ray tracing* exaustivo devido à grande

quantidade de cálculos de interseção necessários para percorrer a árvore e também às manipulações de índices necessárias para encontrar os nós filhos na tabela *hash* através do Código de Morton (Figura 16).

Pontos	Mínimo	Média	Maximo
1000	0,1453	0,1705	0,1873
2000	0,1823	0,2074	0,2402
4000	0,2277	0,2656	0,3043
8000	0,3035	0,3616	0,4005
16000	0,3861	0,4605	0,5327
32000	0,5033	0,5917	0,6868

Tabela 3: Resultados dos testes com a nuvem de pontos e o *ray tracing* usando a tabela *hash*.

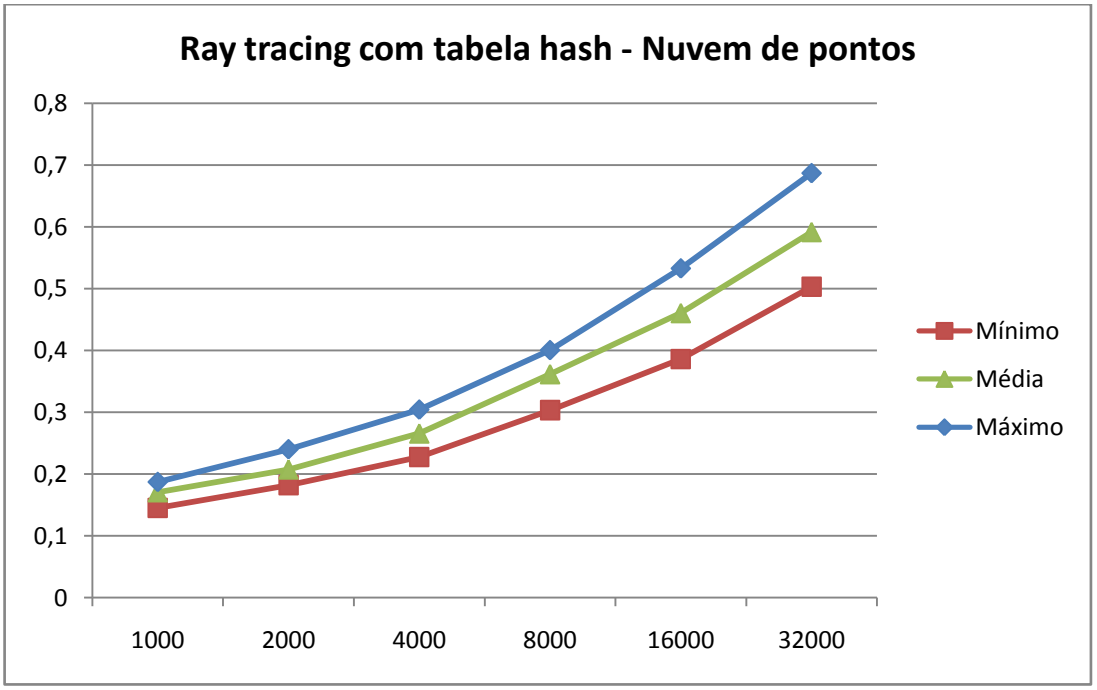

Figura 16: Detalhamento dos resultados do *ray tracing* com tabela *hash* (tempo em segundos).

Por fim, a nuvem de pontos foi testada com o *ray tracing* utilizando a octree linear. Neste caso, foram feitos testes com a quantidade de pontos variando entre 1000 e 16000. Novamente a limitação de memória impediu testes com nuvens de pontos maiores. O algoritmo é mais eficiente que os anteriores tão logo o volume de dados começa a crescer.

Ao empregar a *octree* na lista linear percebe-se uma redução no ritmo de aumento do custo computacional. Enquanto em casos anteriores o custo chegava a dobrar juntamente com

o volume de dados, aqui o custo apresenta-se mais estável. É preciso aumentar cerca de 16 vezes o volume de dados para observar o tempo dobrar.

Pontos	Mínimo	Média	Maximo
1000	0,0209	0,0241	0,0289
2000	0,0244	0,0282	0,0328
4000	0,0383	0,0427	0,0472
8000	0,0436	0,0476	0,0515
16000	0,0476	0,0562	0,064

Tabela 4: Resultados do *ray tracing* com a *octree* linear.

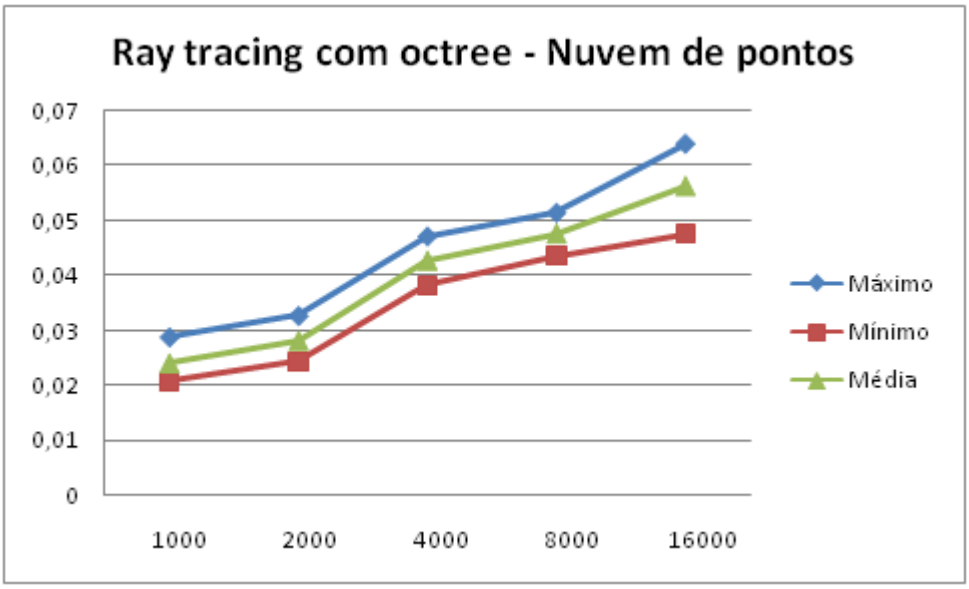

Figura 17: Progressão do custo da *octree* linear conforme cresce o volume de dados (tempo em segundos).

Conforme visto, em casos onde o volume de dados é grande o suficiente, o custo extra de percorrer uma *octree* é compensado e o algoritmo obtém um desempenho bastante superior ao do percorrimento linear da cena. Também se pôde observar que a *octree* em tabela *hash* tem um custo agregado razoavelmente grande, o que faz com que a implementação da *octree* linear, ainda que custosa em termos de espaço, a melhor opção em questão de tempo de processamento.

Pontos	Exaustivo	Tabela *hash*	Diferença
1000	0,0182	0,1705	9,368132
2000	0,0327	0,2074	6,342508
4000	0,0609	0,2656	4,361248
8000	0,1202	0,3616	3,008319
16000	0,2423	0,4605	1,900537
32000	0,4802	0,5917	1,232195
64000	0,9123	0,7158	0,784610

Tabela 5: Comparativo de tempo de processamento entre o algoritmo exaustivo e o algoritmo utilizando a tabela *hash*.

A Tabela 5 mostra que inicialmente, com baixo volume de dados, há uma enorme vantagem em realizar o *ray tracing* diretamente com todos os vértices da cena. No caso inicial, com 1000 pontos, ele chega a ser 9 vezes mais rápido que a implementação da tabela *hash*. No entanto, com o crescimento do número de vértices essa diferença é reduzida e se inverte no último caso. A Figura 18 mostra esse comportamento.

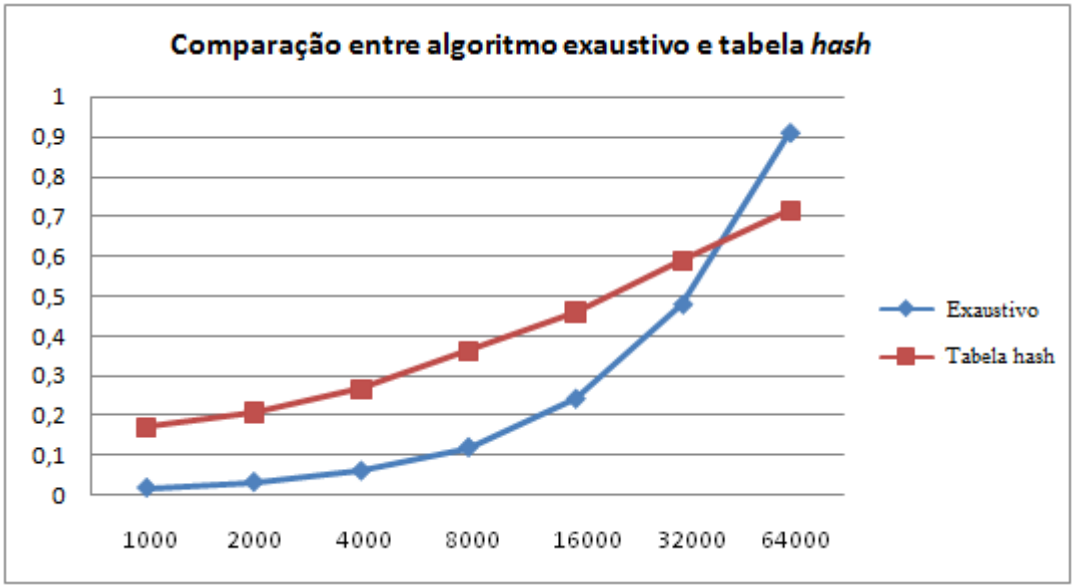

Figura 18: Progressão de custo dos algoritmos comparados (tempo em segundos).

Um comparativo similar pode ser feito entre a implementação do *ray tracing* exaustivo e da *octreee* linear. Neste caso, devido às limitações de espaço da *octree* linear, os testes foram feitos somente até 16000 vértices.

Pontos	Exaustivo	*Octree* linear	Diferença
1000	0,0182	0,0241	1,324176
2000	0,0327	0,0282	0,862385
4000	0,0609	0,0427	0,701149
8000	0,1202	0,0476	0,396007
16000	0,2423	0,0562	0,231944

Tabela 6: Comparativo de tempo de processamento entre o algoritmo exaustivo e o algoritmo utilizando a *octree* linear.

Aqui, a Tabela 6: mostra a progressão da relação de desempenho entre as duas representações. O *ray tracing* exaustivo só obtém um desempenho melhor no caso com o menor número de vértices, posteriormente o algoritmo que utiliza a *octree* é superior em todos os casos, aumentando a diferença conforme a quantidade de dados cresce. A Figura 19 evidencia este comportamento, onde o crescimento da *octree* linear se estabiliza, ao contrário do *ray tracing* exaustivo.

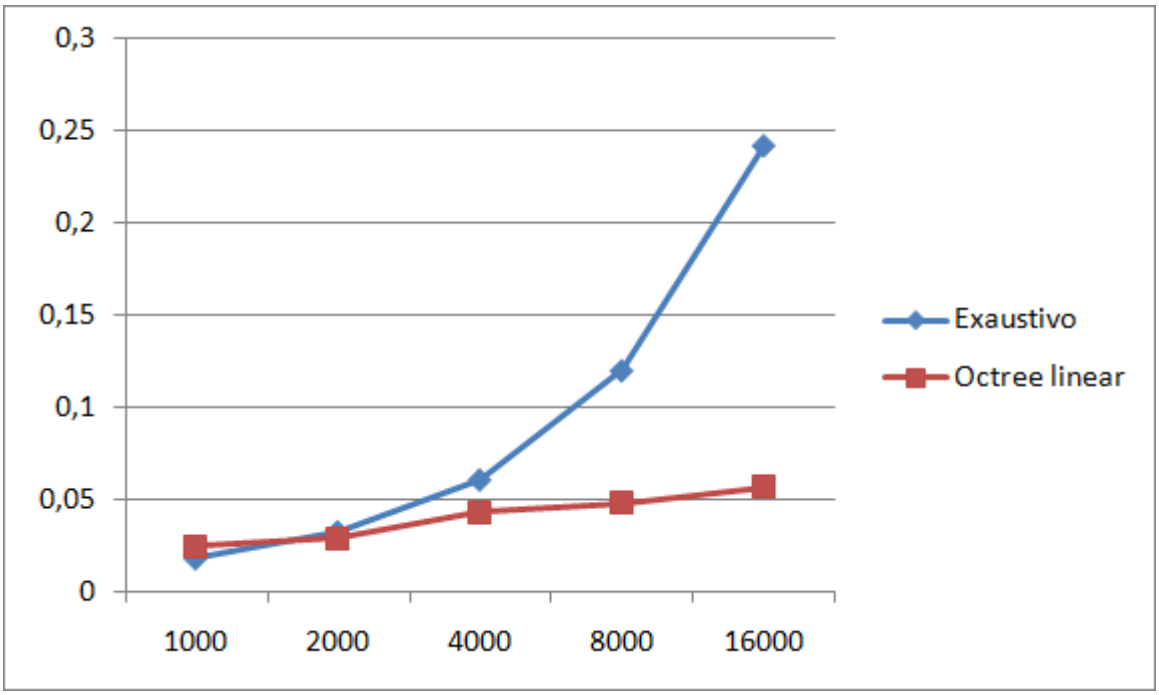

Figura 19: Progressão de custo do algoritmo exaustivo e da *octree* linear.

5.1.2 TESTES COM MALHA POLIGONAL

Também foram realizados testes do *ray tracing* utilizando uma cena construída a partir do uma malha poligonal. Empiricamente foi definido que o número máximo de primitivas por folha seria 12, valor suficiente para construir as árvores de todos os modelos testados.

Foram testados quatro modelos 3D distintos, mostrados na Figura 20 e detalhados na Tabela 7. É importante notar que, em função da maneira que a *octree* é construída, uma mesma face pode pertencer a mais de um nó na árvore. Assim, a contagem de faces para o *ray tracing* com *octree* é, em geral, maior que o *ray tracing* exaustivo. Nos testes, a contagem de faces na *octree* oscilou entre 3 e 5 vezes mais que o número original do modelo. Isto pode afetar os resultados até certo ponto, dando a impressão que o ganho da *octree* sobre o método exaustivo é menor que o ganho real.

Nome	**Vértices**	**Faces**	**Faces na *octree***
cubes	104	156	609
blackmage	824	1644	4757
house	1460	3028	10979
ant	486	912	4967

Tabela 7: Número de vértices e faces de cada modelo.

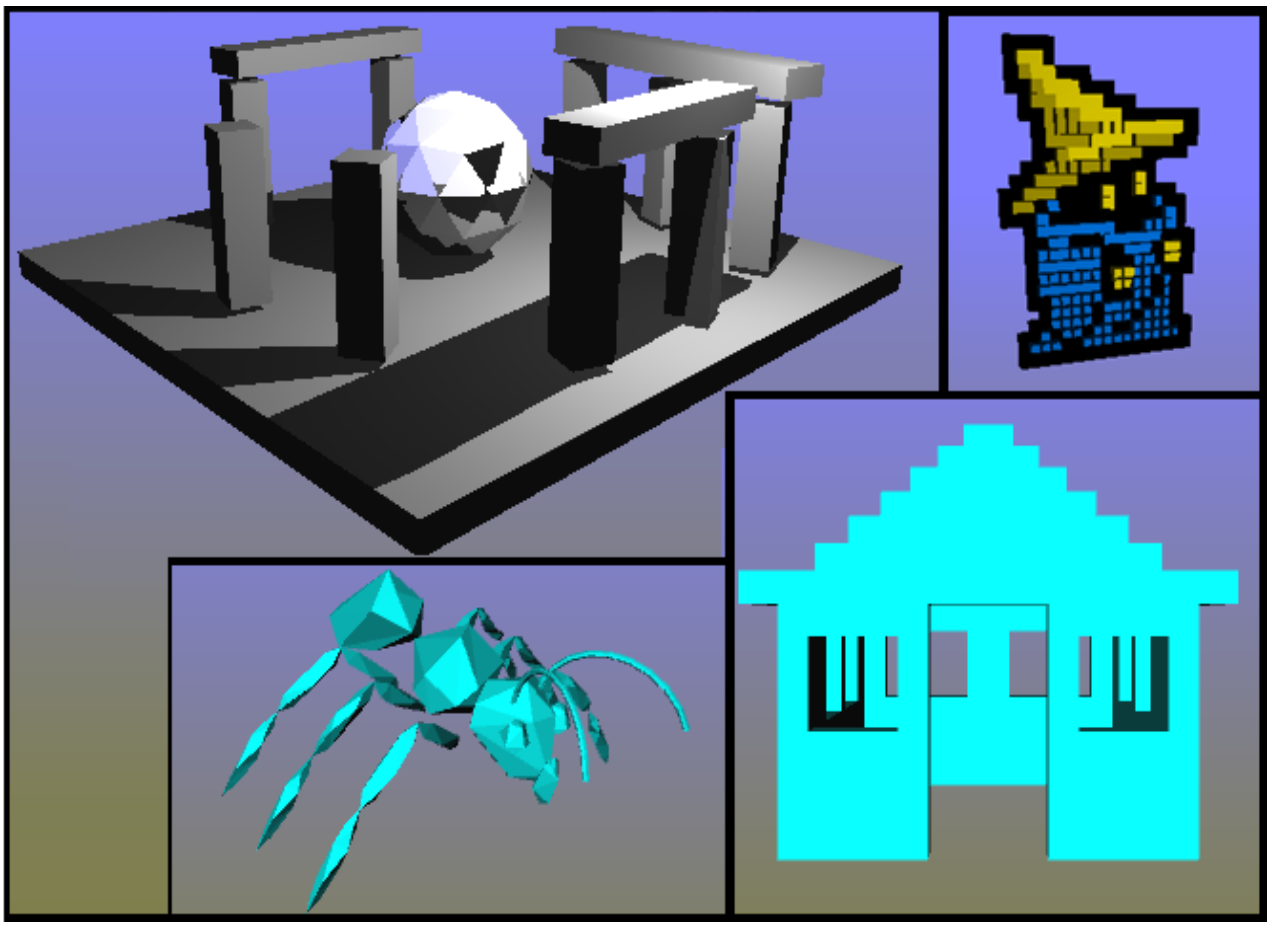

Figura 20: Modelos tridimensionais usados nos testes.

A seguir, a Tabela 8 lista os resultados obtidos com o *ray tracing* utilizando a *octree* como estrutura de aceleração para a cena construída a partir de diversas malhas poligonais. Nota-se certa constância no tempo de execução, a despeito do tamanho da cena. Exemplificando, o modelo *cubes* tem cerca de seiscentas faces e leva um tempo médio de execução próximo de 0,09 segundos. Com dezesseis vezes mais faces, o modelo *house* possui quase onze mil faces e tem o tempo médio de renderização de aproximadamente 0,05 segundos.

É possível notar ainda que cenas com metade do tamanho de outras possuem desempenho parecido e, em alguns casos, ligeiramente pior (como é o caso do modelo *ant* que perde em todos os testes para o modelo *house*). A proximidade dos tempos medidos se deve ao ganho do uso da *octree*, que substitui grande parte dos cálculos de interseção pelo percorrimento da árvore. O desempenho ruim de alguns modelos pode ser atribuído à forma que as faces estão distribuídas pelo cenário, fazendo com que as folhas fiquem em níveis muito baixos da árvore, aumentando o tempo de percorrimento de cada raio.

Nome	**Mínimo**	**Médio**	**Máximo**
cubes	0,0826	0,0905	0,1994
blackmage	0,043	0,0454	0,0553
house	0,0468	0,0499	0,0543
ant	0,0712	0,0753	0,0794

Tabela 8: Resultados obtidos com o *ray tracing* com *octree* para cenas com malhas poligonais.

Na Tabela 9 são apresentados os resultados obtidos com o *ray tracing* exaustivo. Nota-se um crescimento acentuado no tempo de execução de acordo com o tamanho do modelo.

Nome	**Mínimo**	**Médio**	**Máximo**
cubes	0,0209	0,0224	0,0903
blackmage	0,2248	0,2278	0,226
house	0,4131	0,4157	0,4145
ant	0,1252	0,127	0,1261

Tabela 9: Resultados obtidos com o *ray tracing* exaustivo para cenas com malhas poligonais.

Enfim, a Figura 21 compara o tempo médio de execução do *ray tracing* para cada um dos modelos utilizando o método exaustivo ou a *octree*. Apenas para o modelo *cubes* o método exaustivo foi mais rápido. A baixa quantidade de polígonos nesta cena fez com que o

custo de percorrer a árvore fosse maior do que percorrer toda a cena. Nos demais casos, o alto número de polígonos foi suficiente para obter um ganho no emprego da *octree*.

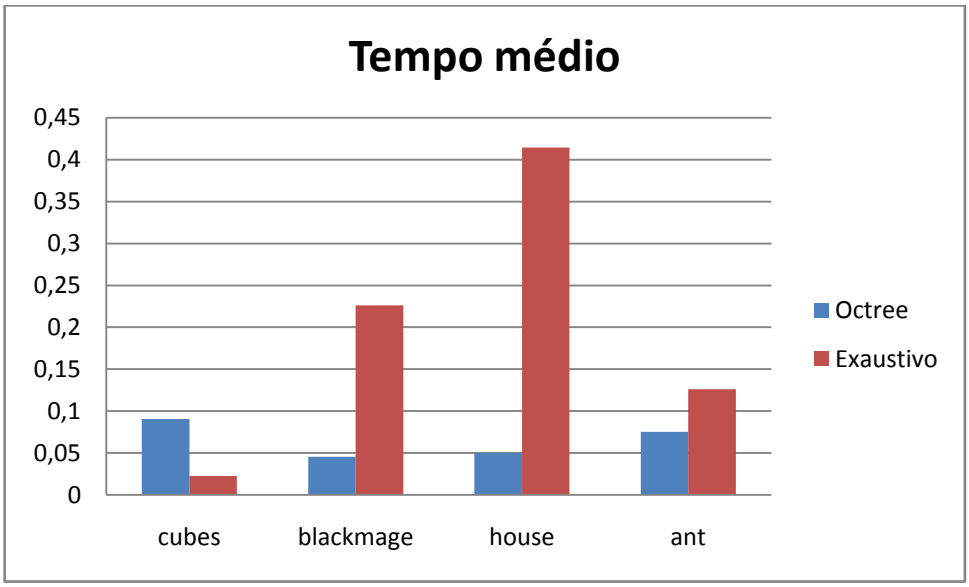

Figura 21: Comparação do tempo de execução entre o *ray tracing* exaustivo e com a utilização de *octree*.

CAPÍTULO 6 – CONCLUSÃO E TRABALHOS FUTUROS

Este trabalho apresentou uma implementação do algoritmo de *ray tracing* em paralelo na GPU utilizando uma estrutura de dados *octree* para reduzir o número do total de cálculos de interseção e aumentar seu desempenho.

Foram estudadas diferentes formas de representar a *octree* na memória de vídeo de acordo com a arquitetura de GPU utilizada. Foi proposta uma representação de listas para a indexação de primitivas contidas nas folhas da *octree*, diminuindo o uso de memória em uma arquitetura de placas gráficas que não permite alocação dinâmica.

Testes comparativos foram realizados entre diferentes implementações para o percorrimento da cena no *ray tracing*: o uso de tabelas *hash*, *octrees* representadas em listas lineares e o *ray tracing* exaustivo, onde nenhuma estrutura de aceleração é empregada.

Os testes mostraram que a implementação da *octree* utilizando uma lista linear é mais eficiente, apesar mais ser mais custosa em relação ao gasto de memória. Esta eficiência em relação à representação em tabela *hash* é devida principalmente à manipulação de índices que deve ser realizada para percorrer esta tabela. Como o acesso aos nós na lista linear é feito somente através de uma adição e uma multiplicação, tem-se uma grande economia de instruções, algo que se refletiu fortemente na prática.

Como sugestão de trabalhos futuros, sugere-se a utilização dos recursos disponibilizados placas gráficas com arquitetura Fermi. Estas placas permitem o uso de ponteiros, o que abre um caminho para experimentação de uma nova maneira de representação da *octree*. Além disso, a possibilidade de se realizar recursão nesta arquitetura permite realizar a busca na árvore de forma mais simplificada e econômica.

Também é sugerida a implementação deste trabalho em um *cluster* de GPUs. Devido à natureza paralelizável do algoritmo de *ray tracing* e ao fato dos dados estarem sempre na memória de vídeo, a adaptação para um *cluster* não deve ser problemática, ao passo que espera-se um grande ganho de desempenho com acréscimo de poder computacional.

Por fim, propõe-se a elaboração de uma estrutura de vizinhança na octree, onde os nós podem ser percorridos de acordo com sua relação de proximidade. Esta adição pode ser especialmente interessante para consultar nós vizinhos no cálculo dos raios secundários, por exemplo.

REFERÊNCIAS

AKENINE-MOLLER, T. Fast 3D Triangle-Box Overlap Testing. in:· Proceeding SIGGRAPH '05. New York, NY, 2005.

APPEL, A. Some techniques for shading machine renderings of solids. SJCC, p27–45, 1968.

BARBOZA, D., CLUA, E. A GPU-Based Data Structure for a Parallel Ray Tracing Illumination Algorithm. In: X Simpósio Brasileiro de Jogos e Entretenimento Digital, 2011, Salvador. Proceedings of the X Simpósio Brasileiro de Jogos e Entretenimento Digital - Computing - Full Papers, 2011. (Submetido)

CASTRO, R. et al. Statistical optimization of octree searches. Computer Graphics Forum, v. 27, p. 1557-1566, 2008.

DEUL, C., BURGER, M., HILDENBRAND, D., KOCH, A. Raytracing point clouds using geometric algebra. Proceedings of the GraVisMa workshop, 2010.

HALL, R. A., GREENBERG, D. P. A Testbed for Realistic Image Synthesis. IEEE Computer Graphics and Applications, v.3, n.8, p10-20, 1983.

HORN, D., SURGEMAN, J., HOUSTON, M., HANRAHAN, P. Interactive K-D Tree GPU Raytracing. I3D '07 Proceedings of the 2007 symposium on Interactive 3D graphics and games. ACM New York, NY, 2007.

KIRK, D; CUDA Memory Model. 2008.

KUCHKUDA, R. An Introduction to Ray Tracing. Theoretical Foundations of Computer Graphics and CAD, Springer-Verlag, Berlin, p1039-1060, 1988.

LINSEN, L., MULLER, K., ROSENTHAL, P. Splat-based Ray Tracing of Point Clouds. Journal of WSCG, 15(1-3), 51–58, 2007.

MADEIRA, D. Uma Estrutura Baseada em Hash Table para Buscas Otimizadas em Octree em GPU. Dissertação de mestrado, Universidade Federal Fluminense, 2010.

MORTON, G. M. A computer oriented geodetic data base and a new technique in file sequencing. [S.l.], 1966.

NVIDIA. Nvidia Cuda. Best Practices Guide. 2011.

PHONG, B. T. Illumination for computer generated pictures. Commun. ACM 18, 6 (June 1975), p311-317, 1975. DOI=http://doi.acm.org/10.1145/360825.360839.

PURCELL, T. J., BUCK, I., MARK, W. R., Hanrahan, P. Ray Tracing on Programmable Graphics Hardware. In ACM Transactions on Graphics 21, 3 (July 2002), 703-712.

REVELLES, J. URENA, C., LASTRA, M. An Efficient Parametric Algorithm for Octree Traversal. Journal of WSCG, p.212-219, 2000.

SANTOS, E. Avaliacao do algoritmo de ray tracing em multicomputadores. Dissertação de Mestrado, Universidade de São Paulo, 1994.

SEGENCHUK, S. Implementation of an Accelerated Ray Tracer. 1997.

WALD, I., SLUSALLEK, P. State of the Art in Interactive Ray Tracing. EUROGRAPHICS ’01. 2001.

WHITTED, T. An improved illumination model for shaded display. Communications of the ACM 23(6), 343-349, 1980.

ANEXO A – LISTAGEM DE CÓDIGO DO PERCORRIMENTO DA ÁRVORE NO RAY TRACING

Este anexo apresenta o trecho de código que realiza o percorrimento da árvore que armazena a cena visualizada pelo *ray tracing*. A função *TraverseScene* implementa o percorrimento da árvore a fim de encontrar as interseções entre os raios e as primitivas da cena. O retorno desta função é uma estrutura contendo o índice da primitiva mais próxima ao ponto de visão interceptada e a distância da interseção. Neste exemplo, a *octree* linear é percorrida. O percorrimento da *octree* em forma de tabela *hash* é análogo, substituindo o acesso aos filhos de um dado nó pela concatenação do código espacial ao código de Morton do nó pai.

```
// Define uma fila de tamanho máximo pré-definido
int nodeQueue[QUEUE_SIZE], currentQueueItem = -1, lastQueueItem = -1;

// Acessa a raíz da árvore
HashItemGPU currentItem = octree[0];

// Calcula a bounding box para a raíz
float4 bb = make_float4(currentItem.x, currentItem.y, currentItem.z,
      currentItem.w);
float3 sceneMin = make_float3(bb.x - bb.w, bb.y - bb.w, bb.z - bb.w);
float3 sceneMax = make_float3(bb.x + bb.w, bb.y + bb.w, bb.z + bb.w);

int item;
float tMin, tMax;
bool currentItemValid = true;

// Loop executado enquanto o item atualmente avaliado for válido
while(currentItemValid == true) {
      // Itera pelos filhos do item atual
      for(unsigned int i = 0; i < 8; i++) {
            // Obtém o índice do i-ésimo filho na octree
            item = currentItem.mortonCode * 8 + i + 1;

            // Teste para garantir que o vetor não foi estourado
            if(item >= HASH_SIZE) break;
```

```
            // Acessa o i-ésimo filho e verifica seu tipo
            HashItemGPU node = octree[item];
            if(node.nodeType == INTERNAL) {
                  // Se for um nó interno, verifica se ele deve ser
                  // enfileirado (ou seja, há interseção com o raio)
                  if(RayBoxIntersection(
                        make_float3(node.x - node.w, node.y - node.w,
                              node.z - node.w),
                        make_float3(node.x + node.w, node.y + node.w,
                              node.z + node.w),
                        ray.origin, ray.invertedDirection, tMin, tMax)) {
                        // Em caso positive, o item é adicionado na fila
                        // de nós a serem visitados
                        lastQueueItem++;
                        nodeQueueProbe[lastQueueItem] = item;
                  }
            }
            else if(node.nodeType == LEAF && node.primitiveList != -1) {
                  // Se for um nó folha e sua lista de primitivas for
                  // válida, verifica interseção do raio com as primitivas
                  CheckPrimitive(ray, node, hitRecord);
            }
      } // fim-for

      // Ao término do teste dos filhos, um novo nó é acessado na fila
      if(currentQueueItem != lastQueueItem) {
            // Se ainda existem nós a serem removidos da fila,
            // o próximo nó a ser visitado é removido
            currentQueueItem++;
            currentItem = octree[nodeQueue [currentQueueItem]];
            currentItemValid = true;
      }
      else {
            // Caso contrário, o item atual é inválido e o laço externo é
            // encerrado
            currentItemValid = false;
      }
} // fim-while
```

ANEXO B – LISTAGEM DE CÓDIGO DO ACESSO AOS DADOS DE PRIMITIVAS NA MEMÓRIA

Este anexo apresenta o trecho de código que implementa o acesso aos dados das primitivas na memória para o cálculo de interseção com o raio. Este acesso é implementado na função *CheckPrimitive* de duas maneiras: uma para a interseção com triângulos e outra para a interseção com esferas. Em ambos os casos, a entrada do algoritmo é um raio e um nó da *octree* e o retorno é um registro de colisão.

A seguir é apresentado o código referente ao acesso aos dados da cena representada por triângulos. A representação da cena com esferas é simplificada, sendo necessário se obter apenas o ponto central da esfera e seu raio.

```
if(primitives[item.primitiveList] != -1) {
      // Obtém os índices dos três vértices da face atual (i)
      int currentFace = primitives[node.primitiveList];
      float4 vertexIndices = tex1Dfetch(faceTexture, currentFace);

      // Obtém as coordenadas dos vértices da face
      float4 vertex1 = tex1Dfetch(vertexTexture, vertexIndices.x);
      float4 vertex2 = tex1Dfetch(vertexTexture, vertexIndices.y);
      float4 vertex3 = tex1Dfetch(vertexTexture, vertexIndices.z);

      // Calcula as arestas da face atual, saindo do vértice 1
      float4 edge1 = make_float4(vertex2.x - vertex1.x,
            vertex2.y - vertex1.y, vertex2.z - vertex1.z, 0);
      float4 edge2 = make_float4(vertex3.x - vertex1.x,
            vertex3.y - vertex1.y, vertex3.z - vertex1.z, 0);

      // Calcula a interseção do raio com o triângulo
      float t = RayTriangleIntersection(ray,
            make_float3(vertex1.x, vertex1.y, vertex1.z),
            make_float3(edge1.x, edge1.y, edge1.z),
            make_float3(edge2.x, edge2.y, edge2.z));

            if(t < hitRecord.t && t > 0.001) {
                  // Se houve interseção, atualiza o registro de colisão
                  hitRecord.t = t;
                  hitRecord.hitIndex = currentFace;
            } }
```

ANEXO C – TRATAMENTO DAS THREADS NO RAY TRACING EM PARALELO

Este anexo descreve como é realizado o controle das *threads* do CUDA para a execução do *ray tracing* em paralelo. Utilizando o CUDA, diversas *threads* são disparadas simultaneamente a fim de dividir o trabalho do *ray tracing* e executá-lo paralelamente.

Cada *thread* é responsável por tratar um raio disparado do ponto de visão que passa por um *pixel* na tela em direção à cena. Para determinar o *pixel* correspondente a cada *thread* são realizadas algumas operações sobre os índices das *threads*. O *kernel* do *ray tracing* foi configurado neste trabalho para trabalhar com blocos de tamanho (8, 8, 1) e *grids* de tamanho (*largura da imagem / largura do bloco*, *altura da imagem / altura da imagem*). Nos testes realizados, foram utilizadas imagens de 512x512 *pixels*, portanto o *grid* utilizado tem o tamanho igual a (512 / 8, 512 / 8) -> (64, 64).

Quando um *kernel* em CUDA é disparado, diversas *threads* são executadas e para cada *thread* são atribuídos índices relativos ao bloco à qual a *thread* pertence (*blockIdx*) e ao seu índice interno no bloco (*threadIdx*). A imagem é divida em regiões de 8x8 *pixels* que são delegadas a cada bloco. O índice do bloco determina o início da região tratada por um determinado bloco, enquanto o índice de cada *thread* indica o *pixel* exato que cada *thread* processa dentro do bloco.

As coordenadas do *pixel* tratado por cada *thread* é dada pelos seguintes cálculos utilizando os índices das *threads* e blocos:

```
unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;
```

Tomando como exemplo o eixo X, as *threads* com *blockIdx.x = 0* tratam as oito primeiras colunas, com *threadIdx.x* variando de 0 a 7 (Figura 22a). Observando a variação do índice do bloco (de 0 a 64) e fixando o índice da *thread* em 0, temos o processamento da primeira coluna de cada região (Figura 22b). A combinação da variação destes dois índices permite processar toda a imagem, atribuindo um *pixel* para cada *thread* executada em paralelo.

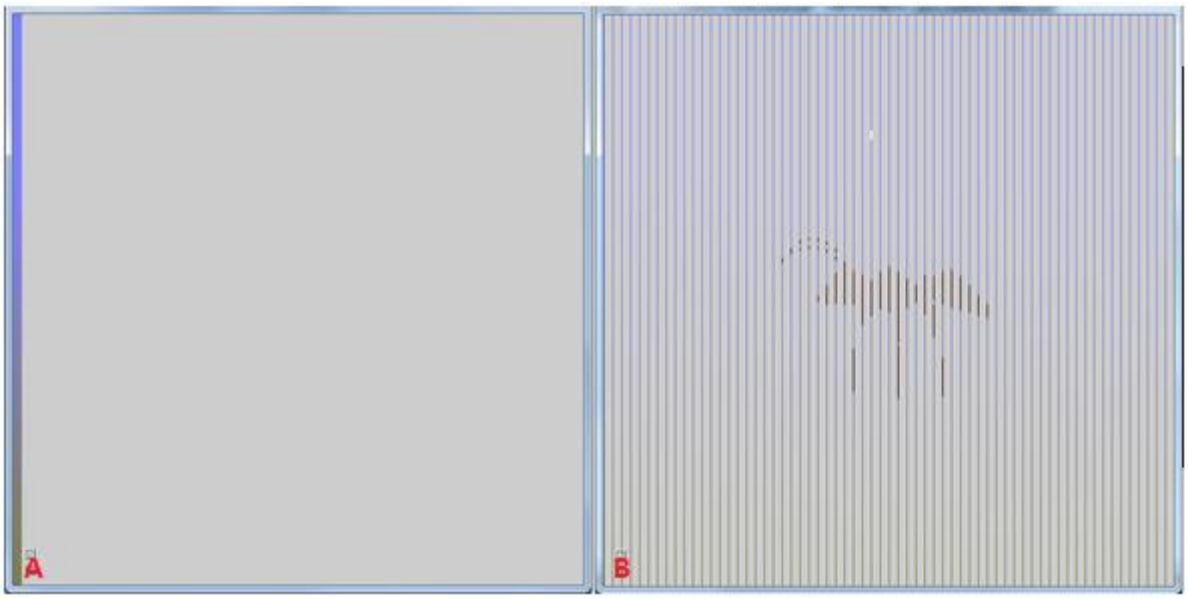

Figura 22 – a) Região processada pelas *threads* pertencentes aos blocos com índice 0 no eixo X; b) Região processada pelas *threads* com índice 0 no eixo X em cada bloco.