Universidade Federal de Mato Grosso do Sul
Faculdade de Computação

Traçado de Raios de Cenas Dinâmicas em CUDA

Marco Aurélio Martins

Dissertação apresentada à Faculdade de Computação da Universidade Federal de Mato Grosso do Sul como parte dos requisitos para obtenção do título de **Mestre em Ciência da Computação**.

ORIENTADOR: Paulo Aristarco Pagliosa

Campo Grande - MS
2010

Feliz aquele que transfere o que sabe e aprende o que ensina. (Cora Coralina)

A minha esposa Alexandra e meus pais Nelson (in memorian) e Aparecida.

Agradecimentos

Agradeço primeiramente a Deus por ter permitido que eu realizasse mais este sonho. Este trabalho é a concretização de um sonho que carrego comigo desde minha graduação, oportunidade em que tive de adiá-lo para que pudesse trabalhar.

Agradeço a meu orientador Paulo Aristarco Pagliosa, professor dedicado, de competência inquestionável e caráter inestimável. Serei eternamente grato pelas suas palavras de incentivo e compreensão em momentos difíceis, e pela sua sabedoria em me conduzir pelo caminho apropriado para que eu atingisse este objetivo.

Agradeço também a minha esposa Alexandra, que muitas vezes, abrindo mão de seus próprios objetivos pessoais, fez dos meus sonhos seus próprios sonhos, e sempre acreditou na realização de cada um deles. Pela sua compreensão nos momentos de minha ausência, quando, de forma muito amiga sempre me dirigiu palavras de incentivo.

E finalmente, agradeço a meus pais que foram responsáveis pela constituição de meu caráter e princípios morais, tão importantes na formação de qualquer pessoa. Por tudo que fizeram por mim até hoje, sou muito grato.

Resumo

Martins, M.A. *Traçado de Raios de Cenas Dinâmicas em CUDA.* Campo Grande, 2010. Dissertação de Mestrado - Faculdade de Computação da Universidade Federal de Mato Grosso do Sul.

O objetivo geral deste trabalho é o estudo das técnicas de traçado de raios para renderização de cenas dinâmicas, estudo de estruturas de dados para aceleração e desenvolvimento de soluções de traçado de raios para CUDA. Traçado de raios é uma técnica capaz de renderizar imagens através da simulação dos efeitos da luz sobre uma cena, portanto, como consequência, é capaz de reproduzir efeitos de sombreamento, reflexão e transparência de forma natural. Cenas dinâmicas são cenas onde os atores podem sofrer transformações ao longo do tempo, seja através de deformação ou mudança de posição. CUDA é uma arquitetura de hardware e software para desenvolvimento de aplicações para GPU. A estrutura de dados implementada neste trabalho é uma BVH, que é uma estrutura hierárquica de volumes envolventes, que visa produzir aceleração para o processo de interseção dos raios com os atores da cena. Como resultado final deste trabalho apresentamos três algoritmos de traçado de raios para GPU e um algoritmo de geração da BVH em GPU. O sistema é composto por um interpretador de uma gramática que descreve a cena, o traçador de raios, responsável pela geração da imagem, e uma interface para apresentação da imagem produzida.

Palavras-chave: *traçado de raios, CUDA, GPU, renderização de imagens, BVH*

Abstract

Martins, M.A. *Traçado de Raios de Cenas Dinâmicas em CUDA.* Campo Grande, 2010. MSc dissertation - Faculdade de Computação da Universidade Federal de Mato Grosso do Sul.

The objective of this work is to study the techniques of ray tracing for rendering dynamic scenes, study of data structures for acceleration and development of ray tracing solutions for CUDA. Ray Tracing is a technique capable of rendering images by simulating the effects of light on a scene, so, consequently, is able to reproduce the effects of shading, reflection and transparency so natural. Dynamic scenes are scenes where the actors may be transformed along the time, either by deformation or change of position. CUDA is an architecture of hardware and software for developing applications for the GPU. The data structure implemented in this work is a BVH, which is a bounding volume hierarchy, which aims to produce the acceleration process intersection of the rays with the actors of the scene. As a final result we present three algorithms for tracing rays and an algorithm for GPU's generation of BVH. The system consists of an interpreter of a grammar of scene, the ray tracer, responsible for generating the image, and an interface for image display produced.

Keywords: *ray tracing, CUDA, GPU, image rendering, BVH*

Conteúdo

1 Introdução **17**

1.1 Motivação e Justificativa . . . 17

1.2 Objetivos e Contribuições . . . 19

1.3 Visão Geral . . . 20

1.4 Organização do Texto . . . 21

2 Traçado de Raios em CPU **23**

2.1 Introdução . . . 23

2.2 Algoritmo de Traçado de Raios . . . 24

2.3 Hierarquia de Volumes Limitantes . . . 27

2.3.1 Trabalhos Relacionados . . . 28

2.3.2 Qualidade de uma BVH . . . 29

2.3.3 Construção da BVH . . . 30

2.3.4 Percurso da BVH . . . 35

2.4 Aspectos de Implementação . . . 38

2.4.1 Leitura da Cena . . . 39

2.4.2 Aplicação . . . 48

2.4.3 Visualização . . . 50

2.5 Comentários Finais . . . 50

3 Traçado de Raios em GPU **53**

3.1 Introdução . . . 53

3.2 Trabalhos Relacionados . . . 53

3.3 CUDA . . . 55

3.3.1 A GPU como Coprocessador da CPU . . . 56

3.3.2 Grids, Blocos e Threads . . . 56

3.3.3 Implementação do Hardware . . . 57

3.3.4 Modelo de Execução . . . 57

3.3.5 Modelo da Memória . . . 58

3.3.6 Interface de Programação . . . 60

3.4 Traçado de Raios em CUDA . . . 61
3.4.1 Algoritmo 1 . . . 64
3.4.2 Algoritmo 2 . . . 67
3.4.3 Algoritmo 3 . . . 68
3.5 Geração da BVH em CUDA . . . 70
3.6 Comentários Finais . . . 70

4 Resultados **73**
4.1 Introdução . . . 73
4.2 Descrição das Cenas . . . 73
4.3 Traçado de Raios Primários . . . 73
4.4 Traçado de Raios . . . 79
4.5 Construção da BVH . . . 84
4.6 Imagens . . . 86
4.7 Comentários Finais . . . 87

5 Conclusão **91**
5.1 Discussão dos Resultados Obtidos . . . 91
5.2 Trabalhos Futuros . . . 93

A Gramática do Leitor de Cenas **95**
A.1 Especificação de Cenas . . . 95
A.2 Atores e Modelos . . . 95
A.3 Luzes . . . 98
A.4 Câmera . . . 99
A.5 Ambiente . . . 100
A.6 Ajustes Globais . . . 100
A.7 Materiais . . . 100
A.8 Transformações . . . 101
A.9 Declarações . . . 101
A.10 Expressões . . . 101
A.11 Inclusão de Arquivos . . . 102

Lista de Figuras

2.1 Frustum. 24
2.2 Traçado de raios. 25
2.3 Raio de pixel. 26
2.4 Intersecção de R_p com O_p. 26
2.5 Raio de luz (ou de sombra). 27
2.6 Representação de uma BVH até o nível 6. 28
2.7 Conjunto de triângulos de uma cena. 32
2.8 Raiz da BVH. 32
2.9 BVH com planos de corte candidatos. 33
2.10 BVH dividida com plano de corte $p3$ escolhido e vetor T rearranjado. 34
2.11 Vetor T inicial. 35
2.12 Índice l avança e encontra elemento a ser permutado. 35
2.13 Índice r retrocede e encontra elemento a ser permutado. 35
2.14 Permuta de $T[l]$ e $T[r]$ é realizada. 36
2.15 Índice l avança e r retrocede. 36
2.16 Índice l avança. 37
2.17 Índice l avança e o algoritmo termina com T rearranjado. 37
2.18 Percurso de um raio em uma BVH. 38
2.19 Representação da BVH da figura 2.18. 38
2.20 Diagrama da classe `Application`. 39
2.21 Diagrama da classe `SceneReader`. 39
2.22 Diagrama da classe `Scene`. 42
2.23 Diagrama da classe `Actor`. 44
2.24 AABB. 46
2.25 Diagrama da classe BVH. 46
2.26 Estrutura de dados para BVH. 47
2.27 Estrutura de dados para malha de triângulos. 48
2.28 Diagrama da classe `RayTracer`. 49
2.29 Diagrama da classe `GLUTMainForm`. 51

3.1 Camadas CUDA. 55
3.2 Blocos de threads. 56
3.3 Dispositivo CUDA. 58
3.4 Modelo de memória. 59
3.5 Estrutura de dados de luzes em GPU. 62
3.6 Estrutura de dados de materiais em GPU. 63
3.7 Estrutura de dados de raios em GPU. 65
3.8 Estrutura de dados de interseção. 65
3.9 Exemplo de compactação de raios a serem traçados. 66

4.1 Traçado de raios primários × traçado de raios. 74
4.2 Traçado de raios primários: aceleração × número de luzes. 77
4.3 Traçado de raios primários: aceleração × número de triângulos (cenas 6 e 11 a 14). . 77
4.4 Traçado de raios primários: aceleração × resolução das imagens (cena 3). 78
4.5 Traçado de raios primários: eficiência. 78
4.6 Traçado de raios: aceleração × número de luzes. 80
4.7 Traçado de raios: aceleração × número de triângulos (cenas 6 e 11 a 14). 81
4.8 Traçado de raios: aceleração × resolução das imagens (cena 3). 82
4.9 Traçado de raios: aceleração × níveis de recursão (cena 5). 82
4.10 Traçado de raios: eficiência. 83
4.11 Geração BVH em GPU - Aceleração X número de nós. 84
4.12 Número de nós gerados por nível (cena 12). 85
4.13 Tempo de geração da BVH X número de nós por nível (Cena 12). 85
4.14 Geração BVH em GPU - Aceleração X número de triângulos. 86
4.15 Cena 1. 86
4.16 Cenas 2 a 5. 87
4.17 Cenas 6 a 11. 88
4.18 Cenas 12 a 15. 89
4.19 Cenas 16 e 17. 89

Lista de Tabelas

2.1 Classe `SceneReader`. 40
2.2 Classe `RayTracerSettings`. 40
2.3 Classe `Parser`. 41
2.4 Classe `Camera`. 42
2.5 Classe `Scene`. 43
2.6 Classe `Actor`. 43
2.7 Classe `Light`. 44
2.8 Classe `Model`. 44
2.9 Classe `Material`. 45
2.10 Classe `Surface`. 45
2.11 Classe `BVH`. 47
2.12 Classe `Renderer`. 49
2.13 Classe `RayTracer`. 50

3.1 Classe `BVHData`. 62
3.2 Classe `CameraData`. 63
3.3 Classe `RayTracerData`. 64
3.4 Classe `CUDARayStack`. 68

4.1 Relação de cenas. 74
4.2 Traçado de raios primários: performance CPU $\times$ GPU (resolução 640 $\times$ 480). 75
4.3 Traçado de raios primários: aceleração e eficiência. 76
4.4 Traçado de raios primários: aceleração $\times$ resolução das imagens (cena 3). 76
4.5 Traçado de raios: performance CPU $\times$ GPU (10 níveis)-(resolução 640 $\times$ 480). 79
4.6 Traçado de raios: aceleração e eficiência (10 níveis). 80
4.7 Traçado de raios: aceleração $\times$ resolução (cena 3). 81
4.8 Traçado de raios: aceleração $\times$ níveis de recursão (cena 5). 82
4.9 Relação de cenas para construção em GPU. 84

Capítulo 1

Introdução

1.1 Motivação e Justificativa

Computação gráfica é o conjunto de técnicas e métodos matemáticos e computacionais envolvidos na criação, armazenamento e manipulação de modelos e de imagens de objetos reais ou imaginários. Um modelo é uma representação das características principais de um objeto, construída com o propósito de permitir a visualização e a compreensão da estrutura e do comportamento do objeto. Uma imagem é uma coleção discreta de pontos coloridos chamados pixels, organizados em um arranjo retangular chamado mapa de pixels. Dentre as várias disciplinas relativas à computação gráfica, estamos interessados, neste trabalho, na síntese de imagens em computador, ou renderização, isto é, na criação de imagens de objetos a partir dos modelos computacionais correspondentes. Particularmente, estamos interessados na síntese de imagens em paralelo usando unidades de processamento gráfico, ou GPUs (*graphics processing units*).

Uma imagem é gerada a partir da descrição de uma cena, uma coleção de atores e luzes. Um ator representa um objeto visível da cena, sendo caracterizado por um modelo geométrico que define precisamente sua posição, formas e dimensões. O modelo geométrico de um ator possui, entre alguns de seus atributos, o material que constitui sua superfície, sua posição na cena, etc. Uma luz representa uma fonte luminosa virtual, cujos raios incidem sobre as superfícies dos modelos geométricos dos atores da cena. A porção visível da cena é determinada por uma câmera virtual que representa o ponto de vista do observador.

A síntese de imagens realísticas de uma cena em computador pode ser dividida em determinação de superfícies visíveis, tonalização e texturização. A determinação de superfícies visíveis consiste em descobrir quais pontos das superfícies dos modelos geométricos dos atores de uma cena são visíveis pela câmera virtual. Para materiais opacos, tais pontos são aqueles mais próximos da posição do observador e que, portanto, escondem os pontos das superfícies que estão atrás de si. Para estes casos, a determinação de superfícies visíveis equivale ao problema de remoção de superfícies escondidas. Tonalização é o processo de determinação da cor de cada ponto, ou pixel, de uma imagem da cena, a qual é fundamentada na interação da luz da cena com o material das superfícies dos modelos geométricos que definem os atores da cena. Texturização é o método de determinação da variação, de ponto a ponto, das propriedades das superfícies dos modelos geométricos dos atores da cena. O objetivo da texturização é fornecer a uma superfície detalhes que não são definidos pela geometria da superfície.

A tonalização de uma superfície é baseada em um modelo de iluminação, um conjunto de equações matemáticas usualmente derivadas das leis da física que simula a interação da luz com os materiais das superfícies dos objetos. Embora derivados da física, os modelos de iluminação mais utilizados em computação gráfica levam em consideração um grande número de hipóteses simpli-

ficadoras em suas formulações, introduzidas para simplificar os cálculos e aumentar sua eficiência computacional. Um dos modelos de iluminação mais simples, e um dos primeiros a ser utilizados em computação gráfica, é o modelo de iluminação difuso, também chamado de modelo lambertiano (neste modelo, uma superfície difusa possui uma aparência fosca, sem brilho). O modelo difuso foi seguido por uma variedade de modelos de iluminação mais realísticos, os quais simulam também reflexão especular [5, 10, 21, 34, 49]. Em 1985, Kajiva [23] introduziu modelos de iluminação anisotrópicos, nos quais as propriedades de reflexão especular variam de acordo com a direção dos raios de luz. Posteriormente, outros autores formularam outros modelos de iluminação anisotrópicos [6, 29, 37].

Os modelos de iluminação citados anteriormente são chamados modelos de iluminação local, porque consideram a luz incidente em uma superfície como sendo diretamente emitida pelas fontes de luz da cena. No começo da década de 1980, a maioria dos esforços dos pesquisadores de modelos de iluminação recaiu sobre os chamados modelos de iluminação global, os quais consideram não somente a luz emitida pelas fontes da luz da cena, mas também a iluminação indireta resultante dos efeitos de reflexão especular e refração sob objetos transparentes ou translúcidos. As técnicas de síntese de imagens mais empregadas para simular efeitos de iluminação global são traçado de raios [15, 46] e radiosidade [4]. Enquanto a técnica de radiosidade admite somente a simulação da iluminação em objetos cujas superfícies sejam Lambertianas, a técnica de traçado de raios permite considerar, de forma simples e elegante, a reflexão especular e as superfícies transparentes ou translúcidas, sendo atualmente uma das técnicas mais utilizadas em programas de síntese de imagens realísticas.

Para simulação de efeitos adicionais, como profundidade de campo, translucência, movimento borrado e, mais importante, a atenuação do efeito de aliasing, uma característica intrínseca e indesejável, sempre presente em imagens geradas por técnicas de amostragem tais como o traçado de raios, pode ser aplicada a técnica de traçado de raios distribuído. Esta técnica é computacionalmente mais intensiva que o traçado de raios determinístico pelo fato de que um número maior de raios é considerado na amostragem.

Por ser computacionalmente intensivo, o traçado de raios tem sido mais empregado em aplicações offline nas quais o tempo de renderização não é mais relevante que a qualidade das imagens geradas, como é o caso de animações computadorizadas e efeitos especiais em filmes de cinema. Para aplicações de simulação e visualização interativas e/ou em tempo real, contudo, o processo, ou pipeline gráfico, de geração de imagens mais usado é a rasterização baseada no algoritmo de Z-buffer [42], simples o bastante para ser implementado diretamente em GPU.

Em 1987 foram introduzidos pela IBM os controladores VGA, utilizados apenas como memória de vídeo, deixando todo o esforço computacional para CPU. Em 1990 foi lançado pela NVIDIA o termo GPU, pois o termo VGA já não descrevia corretamente o hardware gráfico. Assim, até 1998, foram desenvolvidas GPUs com capacidade de interpolação, ou seja, capacidade para calcular os pixels a partir dos vértices de um triângulo e aplicar texturas. Em 1999 e 2000 surgiram os hardwares gráficos provendo transformações de vértices e cálculos de iluminação por vértice. Em 2001 aparece um novo modelo com capacidade de programação do estágio do pipeline gráfico responsável pela manipulação de vértices. O programa é chamado shader de vértice e é executado em paralelo por unidades da GPU chamadas processadores de vértices. Um shader de vértice pode acessar e transformar atributos associados a vértices de triângulos, tais como, posição, vetor normal, cor, texturas, etc. No ano seguinte, surgem os dispositivos com capacidade de programação do estágio do pipeline responsável pela manipulação de fragmentos. O programa é chamado shader de fragmento e executa em paralelo nos chamados processadores de fragmento da GPU. Um shader de fragmento é associado a um pixel da imagem sendo renderizada e é responsável, basicamente, por determinar a cor final e o valor de profundidade do pixel.

A possibilidade de programação de GPUs, aliada ao seu baixo custo relativo e crescente capacidade de processamento paralelo, fez com que estes dispositivos fossem usados não somente em

computação gráfica, mas em outros campos da computação e também em ciências e engenharia. Com isso, surge uma nova área da computação chamada de processamento de propósito geral em GPU, ou GPGPU (*General-purpose Computation on Graphics Processing Units*), ou ainda GPU Computing [17].

Finalmente, em 2007, a NVIDIA lança as primeiras GPUs CUDA (*Compute Unified Device Architecture*). CUDA é uma arquitetura de computação paralela que possibilita mais diretamente o emprego de GPUs em processamento de propósito geral. Em CUDA, não há distinção entre processadores de vértices e de fragmentos; como o nome sugere, a arquitetura da GPU é unificada e constituída por multiprocessadores cada qual contendo um número de processadores escalares SIMD (*single instruction, multiple data*). Desde então diversas aplicações baseadas em CUDA têm sido desenvolvidas, em diversas áreas [11].

O objetivo principal deste trabalho é o desenvolvimento de um traçador de raios para cenas dinâmicas em CUDA. Cenas dinâmicas são aquelas nas quais atores, luzes e câmera podem sofrer transformações ao longo do tempo e/ou atores e luzes podem ser criados e/ou destruídos ao longo do tempo. O traçador de raios apresentado neste trabalho trata de efeitos de reflexão e refração, mas, não trata de texturas, técnicas de tratamento de aliasing ou outros efeitos de luz ou câmera.

O que principalmente motiva e justifica o trabalho é que, com o uso de GPU, há possibilidade do traçado de raios ser empregado para renderização de imagens fotorealísticas em aplicações interativas e/ou em tempo real, tais como jogos digitais. Em tais aplicações, a renderização tem sido mais comumente baseada em rasterização, sendo diretamente implementada em hardware gráfico, através de um pipeline gráfico (conjunto de passos sequências implementados em hardware gráfico com o objetivo de realizar transformações a fim de realizar a renderização de imagens). Embora eficiente, o pipeline requer estágios extras para consideração de efeitos de iluminação global, profundidade de campo, movimento borrado e outros, os quais muitas vezes são simulados por algoritmos complexos e imprecisos. Em traçado de raios, ao contrário, estes efeitos podem ser tratados de maneira mais natural, através da simulação dos efeitos da luz sobre os atores da cena. Assim, espera-se que, com o aumento da capacidade de processamento das GPUs, o traçado de raios possa ser uma alternativa viável para aplicações interativas ou em tempo real que requeiram imagens fotorealísticas, ou, pelo menos, que a GPU possa ser empregada para aceleração da renderização de cenas dinâmicas em aplicações de visualização e simulação offline.

1.2 Objetivos e Contribuições

O objetivo geral deste trabalho é o desenvolvimento de um traçador de raios para cenas dinâmicas em CUDA. Os objetivos específicos são:

- Estudo e implementação de uma estrutura de dados espacial para aceleração dos cálculos de interseção de raios com atores de cenas dinâmicas. Tanto a geração quanto o percurso da estrutura são implementados em CPU e GPU.

- Projeto e implementação de algoritmos paralelos de traçado de raios em GPU. Tais algoritmos são distintos da implementação recursiva em CPU, uma vez que as GPUs disponíveis não admitem programação com funções recursivas.

- Testes de desempenho do traçador de raios.

O número de atores e luzes em uma cena dinâmica é limitado somente pela quantidade de memória disponível. O traçador de raios trata efeitos de reflexão e transparência até um nível máximo de recursão arbitrário, mas o traçado de raios não é distribuído; como consequência, não há tratamento

de aliasing e outros efeitos tratados por esta técnica. O traçado de raios distribuído consiste em realizar o traçado de mais de um raio por pixel, atenuando assim os efeitos de uma amostragem menos significativa, que é o caso quando usamos apenas um raio por pixel. Também não há mapeamento de textura. A geometria dos atores é exata ou aproximadamente representada por malhas de triângulos. Uma malha de triângulo é um conjunto de triângulos conectados entre si.

Pretende-se que o traçador de raios proposto neste trabalho possa ser empregado como ferramenta de ensino de graduação e pós-graduação em disciplinas de computação gráfica e desenvolvimento de jogos digitais, bem como servir de base para outras pesquisas do Grupo de Visualização, Simulação e Jogos Digitais da UFMS, em áreas tais como computação gráfica, jogos digitais e GPU Computing.

1.3 Visão Geral

O traçador de raios desenvolvido neste trabalho recebe as informações da cena a ser renderizada através de um arquivo texto contendo a descrição de todos os atores e luzes da cena, bem como da câmera virtual e de parâmetros de configuração do traçador de raios. A sintaxe para especificação de tais dados é definida por uma gramática que criamos especificamente para este fim, descrita no apêndice A.

A partir da descrição da cena são geradas as malhas de triângulos que representam os atores e a criação das estruturas de dados adequadas para o armazenamento dos atores, luzes, materiais, etc. Posteriormente, as malhas de triângulos de todos atores são agrupadas para a formação de uma única malha contendo todos os triângulos da cena, dividida em duas partes: a primeira contém os triângulos de todos os atores estáticos, isto é, aqueles que não são modificados ao longo do tempo e que constituem o cenário; a segunda contém os triângulos de todos os atores dinâmicos, isto é, aqueles que podem sofrer alterações. As cenas dinâmicas consideradas neste trabalho são aquelas em que os atores podem se movimentar ou deformar, mas não há criação de novos atores e nem destruição de atores ao longo do tempo.

Conforme discutido no capítulo 2, a operação computacionalmente mais intensiva do traçado de raios é o cálculo de interseção de raios com a geometria da cena. A fim de acelerar este cálculos torna-se necessária a utilização de uma estrutura de dados espacial. Após o estudo de diversas estruturas de dados disponíveis na literatura, optamos pelo uso de uma BVH (*bounding volume hierarchy*), dada suas propriedades no que diz respeito à exigência de memória, aceleração proporcionada e possibilidade de melhor adaptação a cenas dinâmicas, e fazemos uso de uma heurística para divisão espacial da cena. O próximo passo, então, consiste na criação de uma BVH a partir da malha de triângulos da cena. Assim como a malha, a BVH também é dividida em duas subestruturas, uma para a porção estática e outra para porção dinâmica da cena. A segmentação da malha e da estrutura de aceleração em uma parte estática e outra dinâmica visa reduzir o trabalho para reconstrução da estrutura de aceleração, pois apenas a parte dinâmica deve ser considerada.

Após a construção das estruturas de dados citadas, o traçado de raios pode ser realizado em CPU ou GPU. A implementação do traçado de raios em CPU é utilizada para validação dos algoritmos em GPU e análise comparativa de performance. Para a GPU foram implementados três algoritmos de traçado de raios, bem como, o algoritmo de geração da BVH.

O programa efetua a geração e o traçado dos raios primários e secundários (reflexão e transparência), até um nível de recursão arbitrário (conforme configuração do arquivo de cena), gerando uma imagem armazenada na memória global da GPU, a qual pode ser exibida no monitor de vídeo ou salva em um arquivo de imagem no formato BMP.

Ao final deste trabalho apresentamos uma análise comparativa de performance entre os algorit-

mos desenvolvidos e as imagens produzidas.

1.4 Organização do Texto

No capítulo 2 fazemos uma introdução à técnica de traçado de raios, onde discutimos sobre as principais estruturas de aceleração disponíveis na literatura e justificamos a escolha de uma estrutura de aceleração para este trabalho. Apresentamos o algoritmo de construção e percurso da estrutura de aceleração escolhida e também o algoritmo de traçado de raios recursivo para CPU. Por fim, apresentamos detalhes da implementação do traçador de raios para CPU, com apresentação das principais classes de objetos e seus principais atributos e métodos.

No capítulo 3 realizamos uma breve introdução sobre o traçado de raios em GPU e uma revisão da literatura, onde apresentamos os principais trabalhos relacionados a este tema. Inicialmente resumimos os principais conceitos da arquitetura CUDA, necessários ao entendimento do traçador de raios desenvolvido, como modelo de execução, modelo de memória, implementação de hardware e interface de programação. Em seguida, apresentamos os três algoritmos de traçado de raios para GPU implementados e comentamos algumas das principais características e diferenças entre eles. Também apresentamos o algoritmo de construção da estrutura de aceleração em GPU.

No capítulo 4 apresentamos os tempos obtidos e uma análise da eficiência de cada um dos algoritmos implementados. Através da variação de parâmetros das cenas, como número de luzes, número de triângulos, níveis de recursão e resolução, buscamos comentar as diferenças de performance para cada algoritmo construído.

No capítulo 5 revisamos os objetivos gerais e específicos estabelecidos e relacionamos uma série de oportunidades para trabalhos futuros.

Capítulo 2

Traçado de Raios em CPU

2.1 Introdução

Neste capítulo fazemos uma introdução sobre a técnica de traçado de raios, a fim de facilitar a compreensão do algoritmo implementado e descrito nas seções subsequentes. Na seção 2.2 apresentamos uma versão recursiva do algoritmo de traçado de raios implementada para CPU. Na seção 2.3 discutimos sobre as principais estruturas de dados usadas para aceleração destes algoritmos e apresentamos os algoritmos de construção e percurso da estrutura de aceleração escolhida. A seção 2.4 apresenta as estruturas de dados utilizadas para implementação do traçador de raios e as principais classes, atributos e métodos que compõem o mesmo.

Para geração de nossas cenas, definimos uma gramática de descrição de cena, que após processada realiza a geração da representação da cena através de uma *malha de triângulos*, o que nos permite armazenar estes dados em áreas contínuas da memória através do uso de vetores, evitando a utilização de ponteiros, numa forma bastante apropriada ao tratamento em GPU por permitir acesso coalescido à memória.

Representamos esta coleção de triângulos através de um conjunto de vetores, um vetor T para armazenar a lista de triângulos, um vetor N para armazenar as normais de cada um dos respectivos triângulos armazenados no vetor T, o vetor V para armazenar os vértices, o vetor M para armazenar os dados dos materiais e o vetor L para armazenar as luzes da cena.

Para calcularmos as normais dos triângulos em um determinado ponto, realizamos uma interpolação linear das normais dos vértices, caso essas tenham sido fornecidas no arquivo de entrada. Quando elas não estão presentes é usada a própria normal do triângulo. Assim, as únicas informações contidas nos triângulos são os índices de seus três vértices e das suas normais.

A partir dos triângulos da cena, realizamos a geração da estrutura hierárquica de volumes limitantes, BHV, que representamos também através de um vetor, conforme explicado na Seção 2.3.3.

O programa toma como argumentos o nome de um arquivo texto contendo a descrição da cena, conforme gramática citada, que suporta a especificação de atores, luzes, câmera, ambiente, ajustes globais, materiais, transformações e expressões envolvendo vetores, cores e números reais.

Um ator é um objeto de cena caracterizado por um modelo geométrico primitivo, que é definido por uma malha de triângulos, uma caixa ou uma esfera. O programa também suporta o uso de objetos no formato *Wavefront OBJ*, que descrevem malhas de triângulos e a aplicação de modificadores de objeto, que são especificação do material da superfície do objeto ou uma sequência de transformações aplicadas ao objeto.

Os tipos de fonte de luz suportadas são pontual e direcional. Uma fonte de luz pontual emite

luz de determinada cor, a partir de determinado ponto, em todas as direções, enquanto uma fonte de luz direcional emite luz de determinada cor em uma direção constante. Uma câmera é definida pela sua posição, o tipo de projeção a ser utilizada (paralela ou perspectiva), um vetor que indica o ponto focal e um vetor que indica o ângulo de projeção. O ambiente da cena é caracterizado pela cor de fundo, cor de luz ambiente e índice de refração do meio no qual os objetos da cena estão inseridos. Ajustes globais são variáveis de controle do programa de traçado de raios, como tamanho da imagem produzida, peso mínimo do raio de luz e nível máximo de recursão, ambos condições de parada para o algoritmo de traçado de raios.

A gramática também suporta produções para especificação do material, onde são especificados: cor, brilho, coeficiente de reflexão especular e transparência, luz difusa, luz ambiente, etc. Transformações como rotação, escala, translação e combinações destas também podem ser aplicadas sobre os objetos definidos, bem como a inclusão de outros arquivos descritores (de cores, materiais, etc.).

2.2 Algoritmo de Traçado de Raios

O algoritmo de traçado de raios toma como entrada a descrição de uma cena, composta por atores, luzes e uma câmera, e produz como saída uma imagem da cena. A porção da cena projetada na imagem é definida pela interseção de seus atores com um volume de vista derivado de um modelo simples de câmera chamado "orifício de alfinete". Este consiste de uma caixa fechada dentro da qual há um filme fotográfico em uma de suas faces. No centro da face oposta está o "orifício de alfinete" pelo qual entram os raios luminosos que impressionam o filme e, assim, formam a imagem. Em computação gráfica, a fim de se evitar imagens invertidas, o "filme" é posto fora da caixa à frente do orifício. Os raios de luz que passam pelas bordas do filme e convergem para o orifício de alfinete definem o volume de vista, uma pirâmide infinita chamada frustum cujo ápice representa a posição do observador. O ponto no centro do "filme", agora chamado janela de projeção, é denominado ponto focal da câmera. O vetor formado pela posição do observador e o ponto focal define a direção de projeção. Usualmente, a profundidade máxima do frustum é delimitada por um plano (de fundo) paralelo à janela de projeção, como ilustrado na a 2.1.

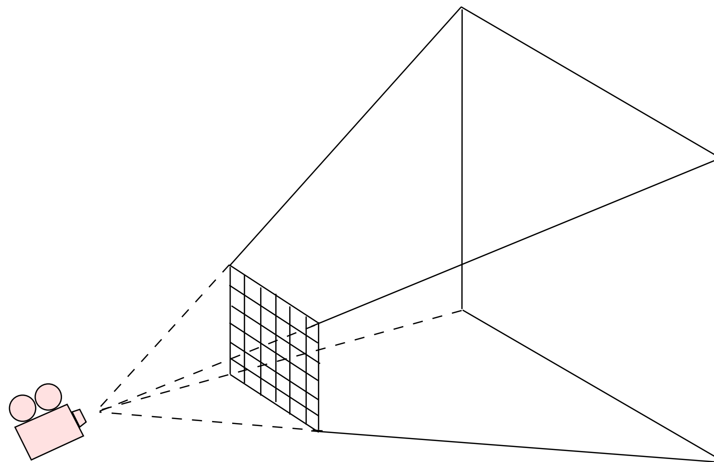


Figura 2.1: Frustum.

No chamado traçado de raios progressivo, a imagem é formada seguindo-se a trajetória de todos os raios emitidos por todas as fontes luminosas e considerando-se apenas aqueles que, após interagirem com as superfícies dos atores da cena, são refletidos e/ou refratados e alcançam o olho do observador. Como exemplo, veja a cena da Figura 2.2.

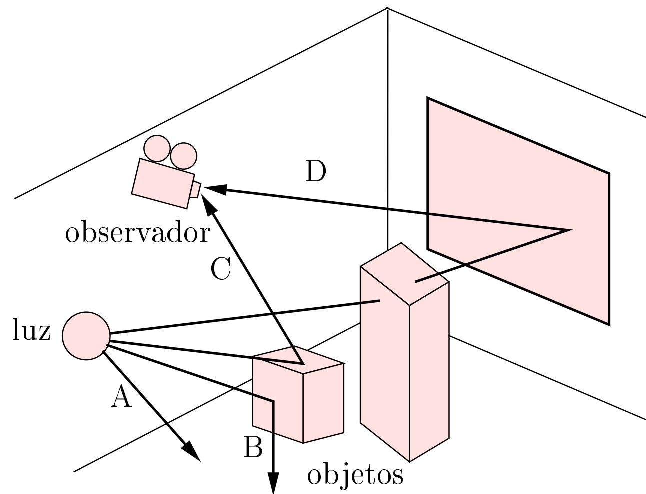


Figura 2.2: Traçado de raios.

O raio A não intercepta nenhum objeto da cena e não contribui para a formação da imagem. Da mesma forma, o raio B não contribui para a imagem, pois reflete na caixa menor e segue uma trajetória que não lhe conduz em direção à posição do observador. O raio C, ao contrário, intercepta a caixa menor e por ela é refletido em direção ao olho do observador, passando por um ponto da imagem (ou seja, entrando na "caixa" pelo "orifício de alfinete" e impressionado o "filme"). A cor do pixel correspondente ao raio C é determinada em função da energia da fonte emissora, da redução dessa energia à medida que o raio trafega no espaço e das propriedades materiais da superfície da caixa menor no ponto de interseção. Observamos um ponto sobre a superfície do objeto como consequência da interação de um raio de luz diretamente proveniente de uma fonte luminosa (ou seja, efeito da iluminação direta, a qual independe de quaisquer outros objetos presentes na cena).

O raio D também contribui para a formação da imagem, mas atinge o olho do observador após refratar na caixa maior (supostamente transparente ou translúcida) e refletir no espelho da parede. A contribuição do raio não depende apenas da luz e das propriedades da caixa maior, mas também das propriedades do espelho no qual a caixa maior aparece refletida (ou seja, iluminação indireta). Diz-se que, por tratar efeitos tanto da iluminação direta quanto indireta, o traçado de raios implementa um modelo de iluminação global.

Obviamente que o tempo de execução do traçado de raios progressivo é inviável computacionalmente, dada a quantidade de raios de luz que partem das fontes luminosas, mesmo para cenas com número reduzido de atores. Ao invés de seguir os raios da fonte de luz até a posição do observador, na prática faz-se o oposto: traçam-se raios partindo da posição da câmera em direção à janela de projeção, verifica-se se há interseção desses raios com os atores da cena e então determina-se a cor nestes pontos de interseção. Este é o traçado de raios regressivo, ou simplesmente traçado de raios [15].

Para a geração de uma imagem de tamanho $w \times h$ pixels, a janela de de projeção é dividida em $w \times h$ áreas, onde cada uma será um pixel da imagem. Na versão mais simples do algoritmo, apresentada a seguir, consideramos que a cor de um pixel é determinada por somente um raio, chamado raio de pixel, disparado da posição do observador em direção ao centro do pixel, conforme Figura 2.3.

Passo 1 Para cada pixel (i, j) da imagem, $0 < i < w(i = 1, 2, ..., w-1)$, $0 < j < h(j = 1, 2, ..., h-1)$, trace um raio de pixel R_p do olho do observador em direção ao centro do pixel. Um raio de pixel é também chamado de raio primário. A cor do raio R_p será a cor do pixel (i, j).

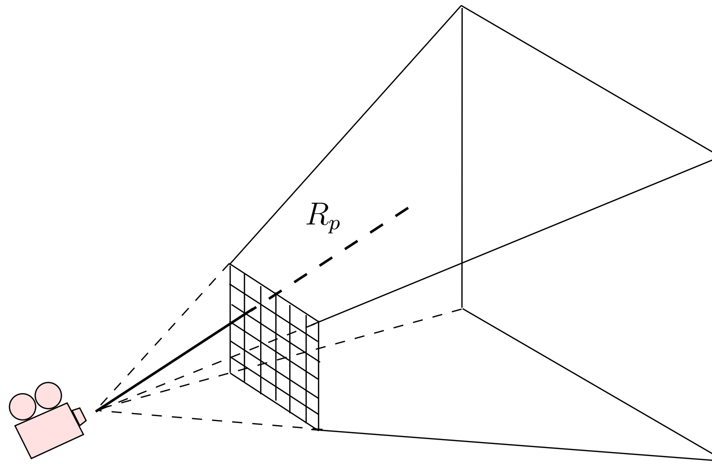


Figura 2.3: Raio de pixel.

Passo 2 Se R_p não interceptar nenhum ator da cena, sua cor será a cor de fundo. Caso contrário, seja I_p o ponto de intersecção de R_p com a superfície do objeto O_p mais próximo da origem de R_p, conforme Figura 2.4.

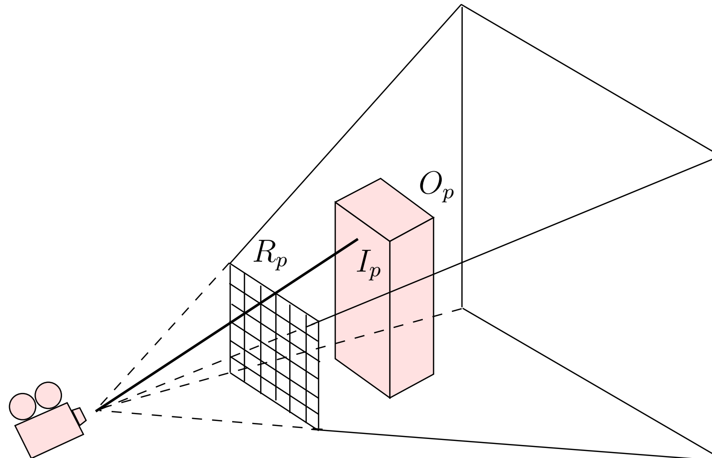


Figura 2.4: Intersecção de R_p com O_p.

Passo 3 Do ponto I_p trace raios em direção a cada uma das fontes de luz da cena. Estes raios são chamados de raios de luz (ou raios de sombra). Seja então uma fonte de luz l e o raio de luz R_l, conforme Figura 2.5. Se R_l não interceptar nenhum ator da cena, então l ilumina diretamente I_p. A cor de R_l é determinada em função do modelo de iluminação local de Phong [34] e adicionada à cor de R_p. Um modelo de iluminação local considera apenas as propriedades da fonte luminosa e do material, negligenciando os fatores relativos ao transporte da luz no espaço e a reflexão e/ou refração entre objetos da cena. Caso haja interseção do raio com algum objeto da cena antes que o raio atinja a fonte de luz, então R_l é um raio de sombra e sua cor é preta. O traçado de raios que considera apenas a iluminação direta para determinação das cores dos pixels da imagem é chamado *ray casting* [2], ou traçado de raios primários. Um traçado de raios mais completo deve considerar também os passos 4 e 5 a seguir.

Passo 4 Se o material de O_p em I_p for reflexivo, traça-se um raio R_r partindo de I_p na direção de reflexão, determinada em função da direção do raio de pixel e da normal à superfície de O_p

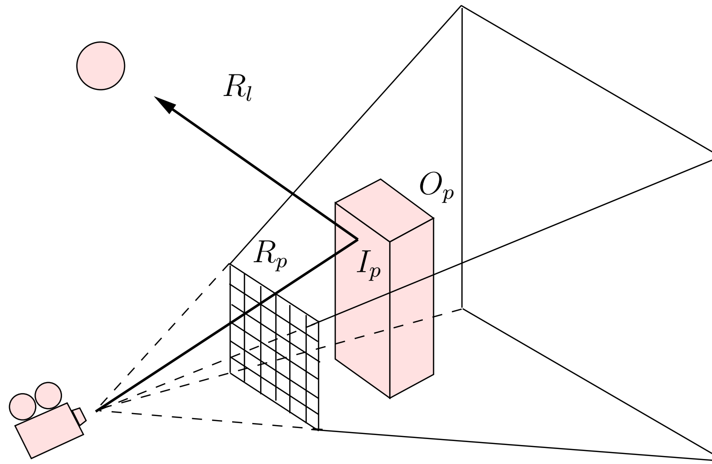


Figura 2.5: Raio de luz (ou de sombra).

em I_p. R_r é chamado raio de reflexão e sua cor é adicionada à cor de R_p. A determinação da cor de R_r é feita executando-se recursivamente o algoritmo, considerando-se que R_r é um raio de pixel e que I_p é o olho do observador.

Passo 5 Se o material de O_p em I_p for transparente, traça-se um raio R_t partindo de I_p na direção de refração, determinada em função da direção do raio de pixel, da normal à superfície de O_p em I_p e dos índices de refração de O_p e do meio no qual trafega o raio de pixel. R_t é chamado raio de transparência ou de refração e sua cor é adicionada à cor de R_p. Como feito no passo 4, a determinação da cor de R_t é feita executando-se recursivamente o algoritmo, considerando-se que R_t é um raio de pixel e que I_p é o olho do observador.

Raios de reflexão e refração são chamados de raios secundários. Note que nos passos 4 e 5 do algoritmo, cada ponto de interseção de um objeto com um raio (primário ou secundário) pode gerar até dois novos raios secundários, um de reflexão e outro de refração, cujas cores são determinadas recursivamente, o que resulta em uma árvore (binária) de recursão durante a execução do algoritmo. Há dois critérios de parada para a recursão:

- Nível de recursão. Todo raio tem um nível de recursão igual a 1 para um raio primário, que é incrementado para cada raio secundário traçado recursivamente. Portanto, o nível de recursão de um raio secundário s é $l_s = l_p + 1$, onde l_p é o nível de recursão do raio "paï" de s (o qual pode ser um raio de pixel ou outro raio secundário). Um raio secundário só é traçado se seu nível de recursão for menor que determinado nível máximo de recursão. Se este for igual a 1, apenas os raios primários são traçados.

- Peso. Todo raio também possui um *peso*, igual a 1 para um raio primário e que decai para os raios secundários em função do peso do raio "pai" e das propriedades do material no ponto de interseção. Um raio secundário só é traçado se seu peso for maior que um determinado peso mínimo.

2.3 Hierarquia de Volumes Limitantes

O desempenho do traçado de raios está principalmente relacionado à eficiência com a qual se determina a interseção de raios (de pixel, de luz, de reflexão e de refração) com os objetos de uma cena. Para uma cena com NA atores e NL luzes, uma imagem de $w \times h$ pixels requer $w \times h \times$ NA testes

de interseção apenas para os raios primários, considerando um teste para cada ator e um raio por pixel. Se p raios primários interceptam algum objeto, são necessários mais $p \times \mathrm{NL} \times \mathrm{NA}$ testes de interseção para os raios de luz, fora os raios secundários. Além disso, a geometria de um ator pode ser representada por uma coleção de primitivos gráficos (neste trabalho nosso primitivo gráfico é um triângulo). À coleção destes primitivos damos o nome de malhas de triângulos.

Assim, se o modelo geométrico de um ator possuir NT triângulos, a interseção de um raio com o ator requer NT testes, o que para cenas com milhares de triângulos, inviabiliza claramente o emprego da força bruta, mesmo em aplicações offline. Para contornar o tempo de execução exigido pelos cálculos de interseção com todos os triângulos, discutiremos nas seções seguintes sobre o uso de estruturas de dados, particularmente as hierárquicas, que permitam a redução do número de cálculos citados e consequentemente aceleram o traçado de raios.

2.3.1 Trabalhos Relacionados

Várias estruturas de dados espaciais foram propostas para acelerar a interseção raio/objeto. As primeiras iniciativas aplicaram a idéia de dividir a cena usando uma grade regular formada por células, ou caixas, com faces alinhadas aos eixos de um sistema de coordenadas cartesianas [1, 3, 39]. Neste método, cada célula mantém uma lista com os primitivos que a interceptam. Se um raio não intercepta a célula, todos os primitivos contidos (parcial ou totalmente) na célula são descartados no teste de interseção com o raio. A estrutura não leva em conta a distribuição da densidade de primitivos no espaço, o que pode resultar em grades com poucas células contendo um número maior de primitivos e muitas células com poucos primitivos ou vazias. Por isso, estruturas não regulares e hierárquica foram propostas, dentre as quais grades adaptativas, grades hierárquicas e octrees [19].

As principais estruturas hierárquicas empregadas em traçado de raios são BVH (*bounding volume hierarchy*) [16], descrita nesta seção (ver figura 2.6), e kd-tree [14].

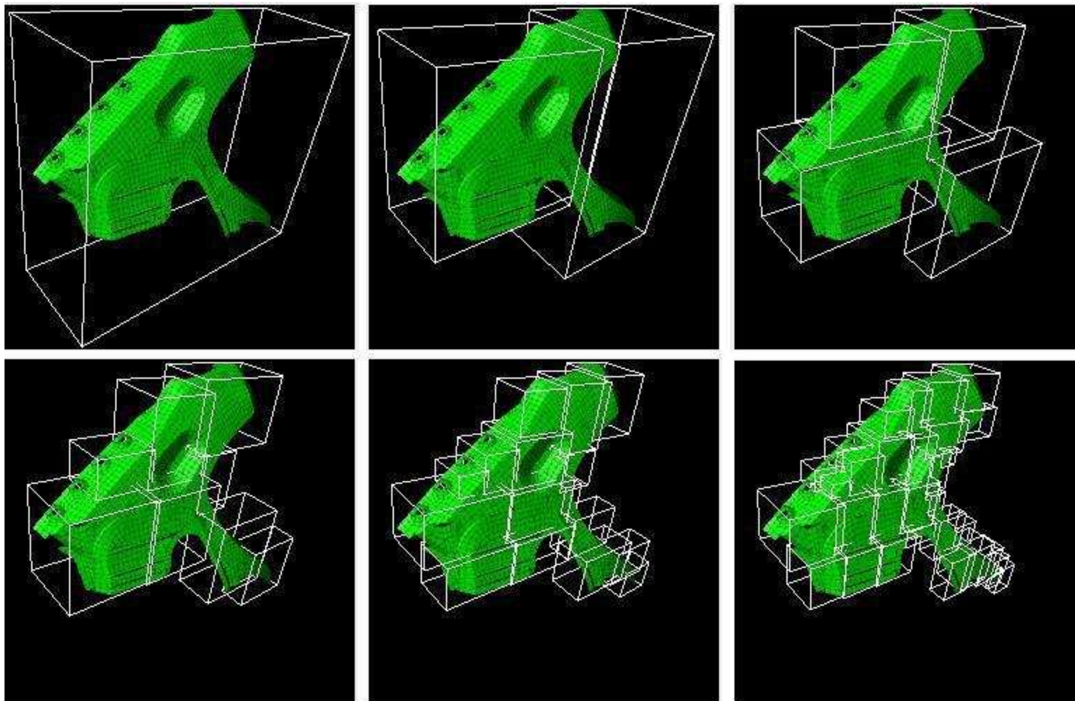


Figura 2.6: Representação de uma BVH até o nível 6.

Uma kd-tree é uma árvore binária cujos nós representam um volume do espaço, usualmente uma caixa com as faces alinhadas aos eixos Cartesianos, ou AABB (*bounding box axis-aligned*). Um nó

folha da árvore mantém uma lista dos primitivos contidos em seu volume. Os dois nós filhos de um nó interno são resultantes da subdivisão do volume do nó pai por um plano perpendicular a um dos eixos Cartesianos e que passa por um ponto qualquer do nó pai [19]. O critério de escolha do eixo normal e do ponto pelo qual passa o plano de corte de um nó interno tem por objetivo produzir uma árvore melhor balanceada, principalmente em cenas nas quais os primitivos são esparsamente posicionados, a fim de proporcionar um tempo de percurso menor.

A partir destas estruturas sugiram algumas variações, dentre as quais *light height*-BVH [9], BIH [48] e bkd-tree [47]. Em geral, estas variações oferecem diferenças na forma de armazenamento, buscando um uso menor de memória, introdução de algumas capacidades adaptativas a métodos de reconstrução, etc.

Há vários trabalhos dedicados à análises comparativas das estruturas citadas, enfatizando as principais qualidades e deficiências de cada uma [3, 20, 40, 41, 43, 44]. Estes trabalhos destacam principalmente que não há uma melhor estrutura de aceleração para o caso geral. As estruturas propostas são mais ou menos eficientes, dependendo da cena em questão. Há especial destaque às estruturas kd-tree e BVH, destacando-se a kd-tree como a melhor estrutura de aceleração em CPU para cenas estáticas, enquanto que a BVH apresenta desempenho muito próximo neste mesmo cenário, e desempenho e características mais apropriadas para tratamento de cenas dinâmicas ou utilização em GPU.

Optamos pelo uso de uma BVH como estrutura de aceleração, tendo em vista que quando comparada a uma kd-tree, o consumo de memória é próximo a 1/3, o que em termos de memória a torna mais apropriada para utilização em GPU, onde o volume de memória disponível é limitado.

Como mencionado acima, o uso de uma hierarquia de volumes limitantes, ou BVH, é uma das alternativas para se contornar o tempo linear do algoritmo de traçado de raios [16, 18, 25, 26, 27]. Uma BVH é uma árvore na qual cada cada nó representa, um volume do espaço, usualmente na forma de um AABB. Embora um nó possa conter múltiplos filhos, consideramos neste trabalho somente BVHs binárias, isto é, cada nó interno da árvore contém exatamente dois nós filhos. Estes representam dois volumes que podem se sobrepor e conter outros volumes limitantes. Um nó folha de uma BVH contém uma lista dos primitivos da cena que são internos ao volume por ele definido. Durante o teste de interseção de um raio com um volume representado por um nó da árvore, uma falha no teste com o volume descarta a necessidade de testes de interseção com todo seu ramo descendente ou com a lista de primitivos contidos neste. A interseção de um raio com uma BVH bem construída (aquela que se aproxima de uma árvore balanceada) tem complexidade $O(\log \mathrm{NT})$, onde NT é o número de primitivos geométricos da cena.

2.3.2 Qualidade de uma BVH

No contexto do algoritmo de traçado de raios, a qualidade de uma BVH é o custo esperado para o seu percurso até um nó folha, dado um raio arbitrário. Uma função para calcular este custo foi proposta em [16]. Esta função, chamada SAH (*surface area heuristic*), baseia-se na probabilidade de que um raio arbitrário r intercepte um volume B, dado que ele interceptou seu volume pai A.

Existem diferentes formas de se construir uma BVH. Um dos métodos mais explorados atualmente que busca maximizar a eficiência no percurso é construir a hierarquia de volumes descendentemente — isto é, da raiz em direção às folhas —, fazendo-se uso de uma heurística de área de superfície, ou SAH (*surface area heuristic*), para particionar recursivamente o conjunto de primitivos contidos no espaço ocupado pela cena a ser renderizada [12, 16, 28]. A SAH é baseada na probabilidade de um raio R interceptar um volume limitante V, estimada com base na área da superfície do volume limitante e no número de primitivos contidos neste volume.

Dado um conjunto NT de primitivos gráficos contidos em um nó N da árvore que representa um

volume limitante V, e assumindo-se que N será dividido por um plano de corte qualquer, em dois novos volumes e dois novos nós L e R serão gerados, com um número de primitivos NT_L e NT_R e com volumes associados V_L e V_R, respectivamente, o custo esperado para percurso de cada possível partição de N pode ser estimado por [45]:

$$\text{custo}(V \rightarrow \{L, R\}) = K_T + \frac{K_I}{\mathrm{SA}(V)} \left[\mathrm{SA}(V_L)\, \mathrm{NT}_L + \mathrm{SA}(V_R)\, \mathrm{NT}_R\right], \tag{2.1}$$

onde $\mathrm{SA}(V)$ é a área da superfície de V e K_T e K_I são constantes definidas pela aplicação que representam, respectivamente, os custos computacionais estimados de um passo de percurso e de interseção de um raio com um primitivo gráfico.

Para um nó N com NT primitivos, o número de partições possíveis de N é $2^{\mathrm{NT}} - 2$, sendo inviável testar todas essas possibilidades a fim de encontrar o particionamento mais adequado de N. Ao invés disto, adotamos uma técnica de amostragem na qual considera-se um número fixo de planos (usualmente perpendiculares aos eixos Cartesianos) e avalia-se o custo de divisão de N por cada um desses planos. O plano que possuir o menor custo será escolhido para partição dos NT primitivos de N.

Como a estimativa de custo para um determinado plano de corte será utilizada apenas para comparação com o custo estimado para os demais planos, podemos eliminar da Equação (2.1) os fatores que são comuns a todos os planos e que, por isso, não influenciarão na comparação. Assim, o custo de cada possível partição é estimado por

$$\text{custo}(V \rightarrow \{L, R\}) = \mathrm{SA}(V_L)\mathrm{NT}_L + \mathrm{SA}(V_R)\mathrm{NT}_R. \tag{2.2}$$

Usando-se esta estimativa de custo, avaliamos entre as várias possíveis partições do conjunto de primitivos e selecionamos aquela que apresentar o menor custo estimado. Esta estratégia é repetida recursivamente para ambos os nós produzidos pela partição de N até que cada nó gerado contenha um número mínimo M de primitivos, ou até que a estimativa de custo de dividir um nó aponte para um valor maior que o custo de transformar um nó em folha [45]. O valor de M pode ser obtido experimentalmente ($M = 5$ neste trabalho).

2.3.3 Construção da BVH

Esta seção apresenta o algoritmo para construção de uma BVH, que toma como entrada uma cena constituída por NT triângulos, representados através do vetor T, e devolve como saída um vetor B contendo os nós da BVH gerada e o vetor T contendo os triângulos reordenados após a geração da BVH.

O valor de k, utilizado no passo 3 do algoritmo 2.1 pode ser obtido experimentalmente. Aumentando-se o valor de k minimizamos o valor do custo estimado pela SAH, porém aumentamos o tempo de execução do algoritmo, lembrando que a decisão de dividir um nó N em k planos perpendiculares busca evitar a complexidade de calcularmos todas as $2^{\mathrm{NT}} - 2$ possibilidades.

Em uma versão preliminar deste trabalho, implementamos a divisão do AABB apenas através do plano perpendicular ao eixo paralelo a maior aresta do AABB, pois teoricamente isto aumenta as chances de estarmos sobre o eixo onde os primitivos estão mais esparsos. Após a implementação e os primeiros testes de performance, optamos por dividir o AABB sempre ao longo dos três eixos cartesianos, pois apesar do maior tempo para construção da BVH, esta solução é capaz de produzir árvores de maior qualidade o que implica em ganho de performance durante o traçado de raios.

Em nosso algoritmo usamos $k = 16$ (em CPU) e $k = 32$ (em GPU), pois em nossos experimentos, e conforme alguns trabalhos encontrados na literatura [18, 45, 35], valores superiores a estes

Algoritmo 2.1 Construção da BVH.

Passo 1 Calcule o menor AABB que contenha todos os NT triângulos da cena e crie o nó raiz N da BVH. Adicione N ao vetor B. Faça com que o primeiro triângulo de N seja o elemento 0 de T e o último seja o último elemento de T

Passo 2 Se o número de triângulos de N for menor que um determinado número M, então não haverá divisão e faça N um nó folha. Termine o algoritmo e devolva o vetor B.

Passo 3 Caso contrário, divida o AABB representado por N em k sub-regiões, usando-se planos equidistantes, perpendiculares a cada um dos eixos cartesianos. A cada um destes planos damos o nome de plano de corte. Cada plano de corte dividirá o AABB representado por N em dois novos AABBs candidatos. A cada uma destas regiões divididas por cada plano de corte damos o nome de cestos.

Passo 4 Calcule o número de triângulos de N pertencentes a cada um dos cestos determinados pelos planos de corte. A determinação é feita verificando-se qual cesto contém o centróide de cada triângulo de N.

Passo 5 Examine os $k - 1$ planos separadores dos cestos, em cada um dos eixos cartesianos, calculando-se o valor do custo estimado da função SAH para cada um dos planos de corte. Escolha o plano de corte que minimiza o valor da função SAH para dividir o AABB e descarte os outros planos de corte.

Passo 6 Se o custo estimado com a SAH do plano de corte escolhido for maior do que o custo de se transformar o nó N em folha, então a divisão não é realizada, transforme o nó N em folha, devolva o vetor B e finalize o algoritmo.

Passo 7 Caso contrário, transforme os dois cestos relativos ao plano de corte escolhido em nó esquerdo e direito do nó N, adicione ambos os nós ao vetor B e realize o rearranjo do vetor T para que os triângulos do nó esquerdo e direito esteja propriamente agrupados (ver algoritmo de rearranjo citado adiante).

Passo 8 Execute o algoritmo recursivamente, a partir do passo 2, para os dois nós computados no passo 6, primeiramente para o nó esquerdo e depois para o nó direito.

---

geralmente demonstram um considerável aumento no tempo de construção da estrutura hierárquica, sem produzir resultados significativos no traçado dos raios.

Em nossa implementação, armazenamos uma BVH em um vetor B, onde cada elemento representa um nó. Cada nó $B[i]$ mantém um índice para seu filho direito. Para um nó que tenha o filho direito armazenado na posição j, o filho esquerdo de $B[i]$ é armazenado na posição $j - 1$, ou seja, na posição imediatamente anterior ao filho direito.

O conjunto de triângulos da cena, dado como entrada no algoritmo de construção é armazenado no vetor T. Para evitar o uso de ponteiros, um nó folha da BVH faz uso de dois índices para indicar seu conjunto de triângulos, um para o primeiro e outro para último triângulo. Após a divisão de um nó em dois outros nós, pode ocorrer que, no vetor T, algum triângulo que pertence ao nó esquerdo esteja à direita de um triângulo que pertence ao nó direito. Este fato impede a representação dos nós criados com o uso apenas do índice inicial e final no vetor de triângulos. Desta forma, faz-se necessário um rearranjo dos triângulos armazenados em T, contidos no nó que está sofrendo divisão, a fim de se agrupar os triângulos dos nós filhos esquerdo e direito. Para evitar alocação adicional de memória, esta ordenação é executada in loco.

Para ilustrar a geração de uma BVH vamos fazer uso de uma cena bidimensional, porém o método se estende a cenas 3D de maneira idêntica e sem restrições. Na figura 2.7 visualizamos um conjunto de triângulos de uma cena 2D que é dado como entrada para o algoritmo de construção da BVH.

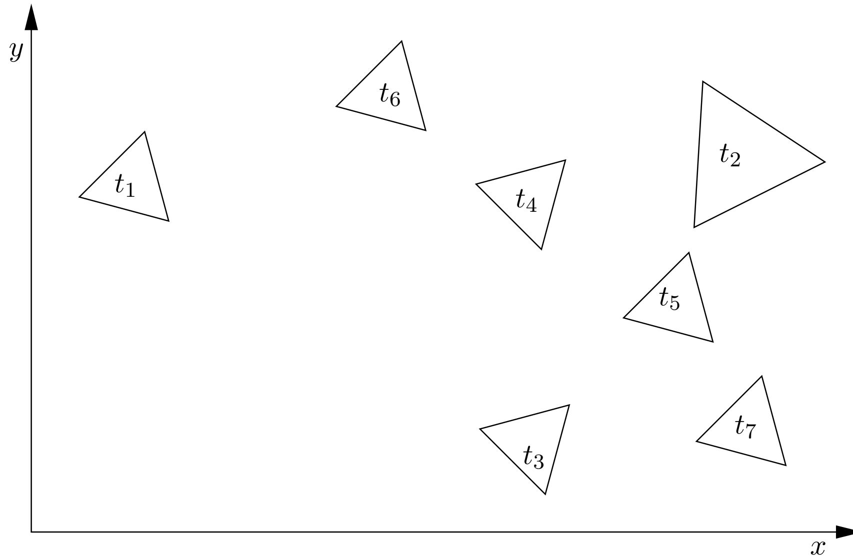


Figura 2.7: Conjunto de triângulos de uma cena.

No primeiro passo, o algoritmo calcula o menor AABB, capaz de enclausurar todos os triângulos da cena, cria o nó N e o armazena na posição $B[0]$, que contém o índice para o primeiro e para o último triângulo do volume envolvente representado por $B[0]$, neste caso todo o vetor T, conforme mostra a figura 2.8.

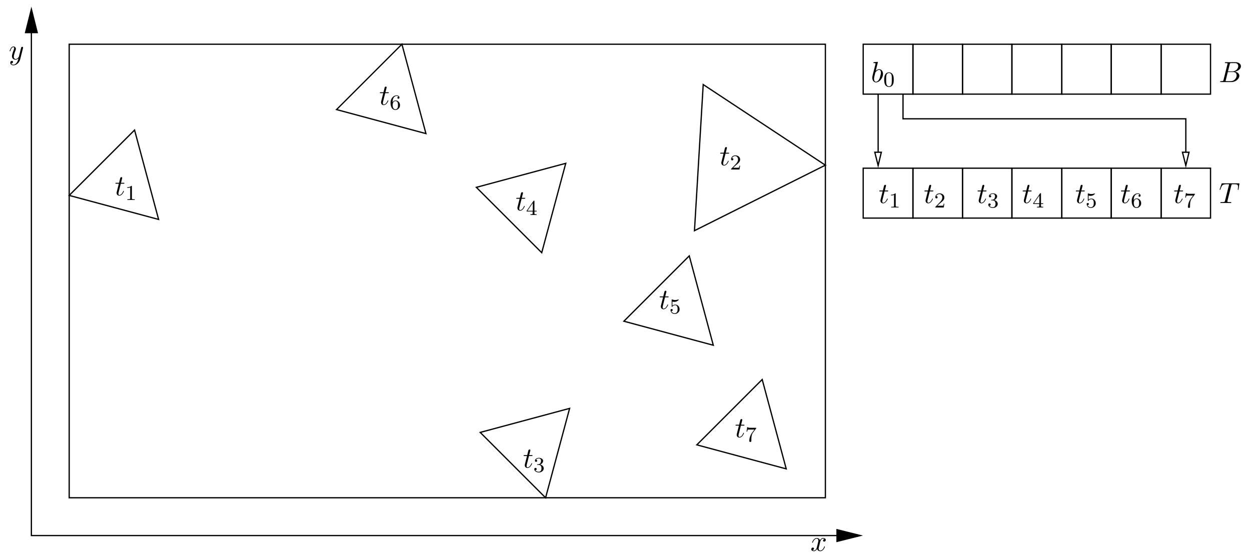


Figura 2.8: Raiz da BVH.

No próximo passo é realizado o corte do AABB ao longo dos três eixos ordenados. O corte é realizado com uso de planos ortogonais a cada um dos eixos escolhidos. Em nosso exemplo, usamos $k = 3$ (número de cortes realizados para escolha do corte de menor custo), $M = 2$ (número mínimo de triângulos em um nó para que a divisão seja considerada viável) e realizamos o corte ao longo

de apenas um dos eixos.

No passo 3 do algoritmo de construção da BVH, dividimos o AABB indicado por N, neste caso $B[0]$, usando os 3 cortes possíveis. À cada um destes cortes damos o nome de plano de corte, representados na figura 2.9, através dos planos de corte p_1, p_2, p_3. Cada plano de corte dividirá o nó N em dois novos AABBs, que serão candidatos a filhos do nó N. À cada uma destas regiões divididas por cada plano de corte damos o nome de cestos.

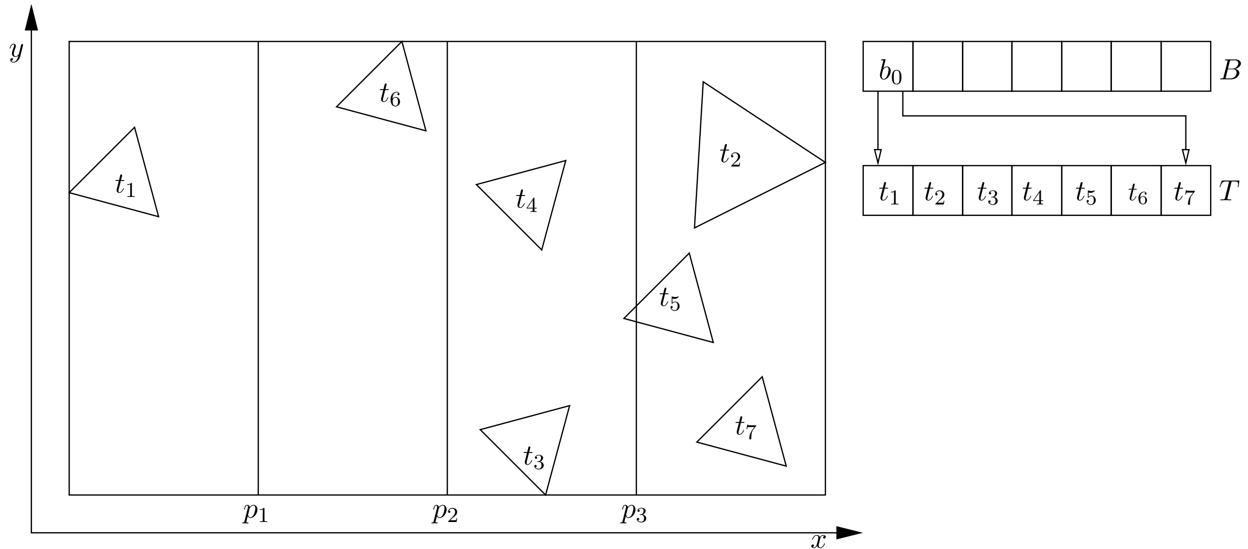


Figura 2.9: BVH com planos de corte candidatos.

Como alguns triângulos podem ser interceptados pelo plano de corte, a determinação do cesto que irá conter o triângulo é realizada com base na posição do centróide do triângulo.

Após a determinação dos triângulos de cada cesto, cada um dos AABBs obtidos são inflados, a fim de que os triângulos estejam inteiramente contindos no AABB. Uma das propriedades de uma BVH é que os AABBs podem se sobrepor, desta forma não há impecilho que um AABB se sobreponha outro, a fim de enclausurar todos os triângulos com centróide nele contido.

A estimativa SAH, demonstrada na equação 2.1, é calculada para cada um dos planos de corte e o plano de corte de custo mínimo é selecionado para divisão do AABB. Neste exemplo, vamos supor que o plano de corte p_3 é o plano de corte de custo mínimo que divide o AABB. O plano de corte p_3 divide o AABB em dois novos AABBs, um com 4 e outro com 3 triângulos. Supondo que o custo obtido pela SAH no plano de corte p_3 é inferior ao custo de transformar o nó N em folha e dado o fato que o AABB representado por N possui 7 triângulos, $7 > M$, então a divisão é considerada viável e é realizada segundo este plano, formando os nós $B[1]$ e $B[2]$, que representarão respectivamente, os nós esquerdo e direito do nó $B[0]$. Para que possamos representar os índices do primeiro e do último triângulo de cada nó criado, o vetor T deve ser rearranjado em dois sub-vetores. A figura 2.10 demonstra a BVH dividida pelo plano de corte p_3 e o vetor T já rearranjado. O algoritmo é executado novamente, a partir do passo 2, para os nós esquerdo e direito, até que o custo de gerar um novo nó seja maior que transformar o nó em folha, ou o número de triângulos de um cesto seja inferior a M.

Rearranjo do Vetor de Triângulos

Explicaremos a seguir, com mais detalhes, como o rearranjo do vetor de triângulos citado é realizado in loco.

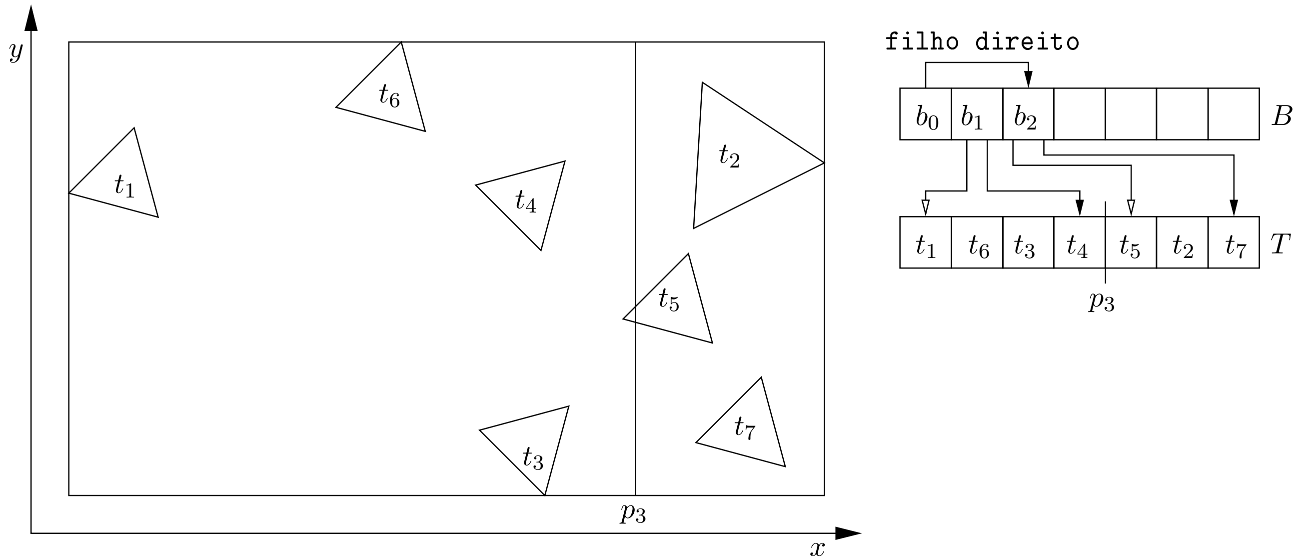


Figura 2.10: BVH dividida com plano de corte $p3$ escolhido e vetor T rearranjado.

Após a divisão de um nó N em filho esquerdo N_l e filho direito N_r é necessário o rearranjo do vetor de triângulos T, para que seja possível que cada um destes nós consiga identificar seu primeiro e último triângulo no vetor T de maneira que todos os triângulos entre o primeiro e o último pertençam ao nó criado. No vetor B, o nó N possui um índice para seu filho direito, indicado por N_r e o nó N_l é armazenado na posição imediatamente anterior ao nó N_r, evitando a exigência de um novo índice para indicar a posição do filho esquerdo. Para o rearranjo do vetor T que mencionamos, o algoritmo de rearranjo a seguir recebe como entrada, em N, o nó da BVH que está sendo dividido.

Algoritmo 2.2 Rearranjo do vetor de triângulos.

Passo 1 $l \leftarrow$ índice do primeiro triângulo do nó N

Passo 2 $r \leftarrow$ índice do último triângulo do nó N

Passo 3 Se $l < r$, então incremente l até que $T[l]$ identifique um triângulo com centróide contido no AABB representado pelo nó direito, ou l seja igual a r

Passo 4 Se $r > l$, então decremente r até que $T[r]$ identifique um triângulo com centróide contido no AABB representada pelo nó esquerdo, ou r seja igual a l

Passo 5 Se $l >= r$, então termine o algoritmo, pois o vetor já encontra-se rearranjado.

Passo 6 Se $l < r$ realize a troca dos triângulos de posição $T[l]$ e $T[r]$.

Passo 7 $l \leftarrow l + 1$; $r \leftarrow r - 1$.

Passo 8 Volte ao passo 3.

Para exemplificarmos a execução do algoritmo de rearranjo, vamos tomar o exemplo da figura 2.10. Após a divisão do AABB pelo plano de corte p_3, $T_1, T_3, T_4, T_6 \in B[1]$ e $T_2, T_5, T_7 \in B[2]$. O algoritmo de rearranjo recebe como entrada o nó $B[0]$, que está sofrendo a divisão. Desta forma, no ínicio do algoritmo de rearranjo $l \leftarrow 0$ e $r \leftarrow 6$, conforme demonstrado na figura 2.11.

No passo 3 do algoritmo, o mesmo faz uso do iterador l para avaliar o lado esquerdo do vetor.

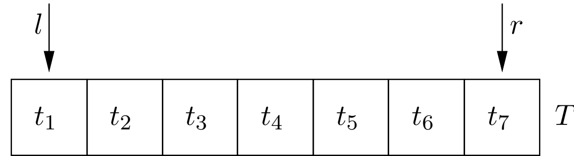


Figura 2.11: Vetor T inicial.

Como o triângulo $T[0]$ já está posicionado no lado correto do vetor (seu centróide indica que este deve pertencer ao filho esquerdo de $B[0]$), l avança, $l \leftarrow 1$. O triângulo $T[1]$ é avaliado, e como este deve pertencer ao nó direito, o avanço do vetor l (que indica o elemento a ser permutado) é interrompido, conforme figura 2.12 e o algoritmo segue ao passo 4.

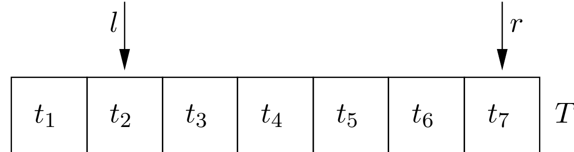


Figura 2.12: Índice l avança e encontra elemento a ser permutado.

No passo 4 o algoritmo inicia a avaliação do lado direito do vetor, procurando um elemento fora de local, a fim de permutá-lo com o elemento do lado esquerdo já encontrado. Como $T[6]$ encontra-se no lado correto do vetor, r é decrementando, $r = 5$. Percebe-se que $T[5]$ está posicionado no local incorreto, assim o retrocesso de r é interrompido e o algoritmo segue ao passo 5, conforme ilustrado na figura 2.13.

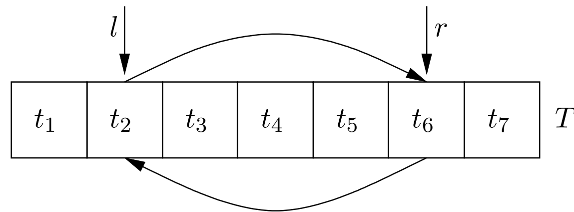


Figura 2.13: Índice r retrocede e encontra elemento a ser permutado.

No passo 5, como $l < r$, o algoritmo segue ao passo 6, para execução da permuta. No passo 6 a permuta de $T[1]$ e $T[5]$ é realizada, produzindo o vetor da figura 2.14.

Como os elementos de $T[0]$ à $T[l]$ estão posicionados corretamente, assim como os elementos de $T[r]$ à $T[6]$, estes elementos não precisam ser revisitados, o algoritmo avança l e retrocede r no passo 7 e segue ao passo 3, conforme mostra figura 2.15.

No passo 3, como $T[2]$ está no lado correto do vetor, l avança para $T[3]$ 2.16.

Como $T[3]$ também está do lado correto do vetor, l avança para $T[4]$, atingindo neste momento a condição de parada do algoritmo, $l = r$, o que indica que o vetor T já está rearranjado, conforme demonstra a figura 2.17.

2.3.4 Percurso da BVH

O algoritmo de percurso da BVH toma como entrada um vetor B contendo os nós da árvore, um vetor de triângulos T e um raio R, e devolve como saída t, que representa o índice do triângulo interceptado que encontra-se mais próximo a origem do raio e d que é a distância da origem do raio

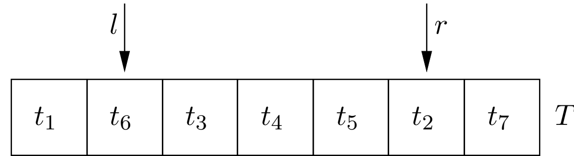


Figura 2.14: Permuta de $T[l]$ e $T[r]$ é realizada.

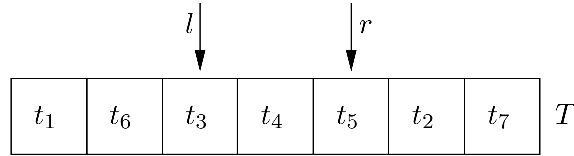


Figura 2.15: Índice l avança e r retrocede.

até o ponto de interseção com o triângulo encontrado. Quando nenhum triângulo é interceptado, o algoritmo devolve $t = -1$ e $d = \infty$.

O percurso de uma estrutura binária como a BVH é naturalmente realizado através de um algoritmo recursivo, que não será apresentado aqui dada a simplicidade do mesmo. O algoritmo a seguir descreve o percurso da BVH através do uso de uma pilha dinâmica de nós, S, e a mesma idéia é aplicada ao algoritmo em GPU, que será apresentado no próximo capítulo, dada a impossibilidade de recursão no dispositivo.

Algoritmo 2.3 Percurso da BVH.

Passo 1 $d \leftarrow \infty$, $t \leftarrow -1$.

Passo 2 Se o raio R interceptar o AABB da raíz da BVH, então empilhe o nó raíz.

Passo 3 Se houver algum nó em S (pilha), então seja N o nó retirado do topo da pilha; senão, devolva t e d e finalize o algoritmo.

Passo 4 Se N for um nó folha, então vá para o passo 5. Senão, teste a interseção do raio R com os AABBs esquerdo e direito de R, filhos de N. Calcule a distância de interseção da origem do raio R com ambos os nós. Empilhe em S os nós interceptados. Vá para o passo 3.

Passo 5 Teste a interseção do raio R com todos os triângulos contidos em N. Sempre que uma interseção de R com um triângulo $t_i \in T$ for encontrada, Calcule a distância d_i do ponto de interseção com t_i até a origem de R; se a distância encontrada for menor que a distância d, então $d \leftarrow d_i$ e $t \leftarrow t_i$. Volte ao passo 3.

Na figura 2.18 ilustramos o percurso de um raio r_1 em uma cena que contém 11 triângulos. Os triângulos encontram-se enclausurados pelos seus respectivos AABBs, representados por b_0 até b_6, conforme indicado na figura.

A representação da cena da figura 2.18 na forma dos vetores que representam a BVH, e o respectivo conjunto de triângulos, é apresentada na figura 2.19. A BVH é representada pelo vetor B e o conjunto de triângulos, pelo vetor T.As ligações entre os elementos do vetor B representam a indicação do filho direito de cada nó da BVH, a indicação do filho esquerdo de cada nó é implícita, pois o mesmo sempre é armazenado na posição imediatamente anterior ao filho direito. As ligações que partem de elementos do vetor B para elementos do vetor T referem-se a indicação do primeiro

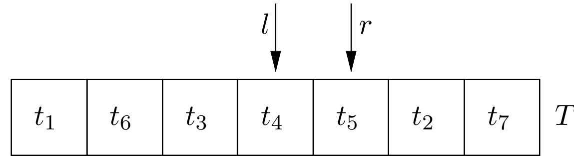


Figura 2.16: Índice l avança.

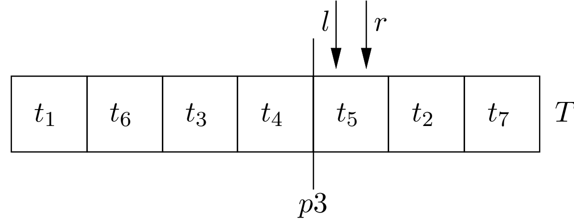


Figura 2.17: Índice l avança e o algoritmo termina com T rearranjado.

e último triângulo de um nó folha da BVH, sendo o primeiro indicado por uma seta vazada e o último por uma seta preenchida.

Usamos o cenário descrito anteriormente nas figuras apresentadas, para descrevermos passo a passo, o atravessamento da estrutura hierárquica pelo raio r_1. A execução abaixo faz referência aos passos do algoritmo 2.3.

Passo 1 Um raio r_1, com origem na câmera é gerado e disparado em direção a cena.

Passo 2 Após teste de interseção do raio r_1 com b_0 indicar positivo, o nó b_0 é empilhado, indicando que o mesmo deve ser visitado.

Passo 3 No passo 3, do algoritmo, o nó b_0 é retirado da pilha e armazenado em N.

Passo 4 No passo 4 do algoritmo, como N não é folha, um teste de interseção com seus filhos, b_1 e b_4 é realizado. Como ambos os testes retornam positivo, ambos os nós são empilhados e o algoritmo segue ao passo 3.

Passo 5 **N**o passo 3 do algoritmo, o nó b_1 é retirado da pilha e atribuído a N.

Passo 6 **N**o passo 4 do algoritmo, como N não é folha, um teste de interseção com seus filhos, b_2 e b_3 é realizado e apenas o nó b_2 é empilhado, uma vez que não há interseção com o nó b_3, e o algoritmo segue a seu passo 3.

Passo 7 **N**o passo 3 do algoritmo, o nó b_2 é retirado da pilha e atribuído a N.

Passo 8 **N**o passo 4 do algoritmo, como N é folha, o algoritmo segue ao passo 5.

Passo 9 **N**o passo 5, o teste detecta interseção com o triângulo t_1, armazenando o índice de t_1 em t e a distância de interseção em d e volta ao passo 3.

Passo 10 **N**o passo 3 do algoritmo, o nó b_4 é retirado da pilha e armazenado em N.

Passo 11 **N**o passo 4, como N não é folha, um teste de interseção é realizado e o nó b_6 é empilhado. O algoritmo segue ao passo 3.

Passo 12 **N**o passo 3, b_6 é retirado da pilha e armazenado em N.

Passo 13 **N**o passo 4, como N é folha, o algoritmo segue ao passo 5.

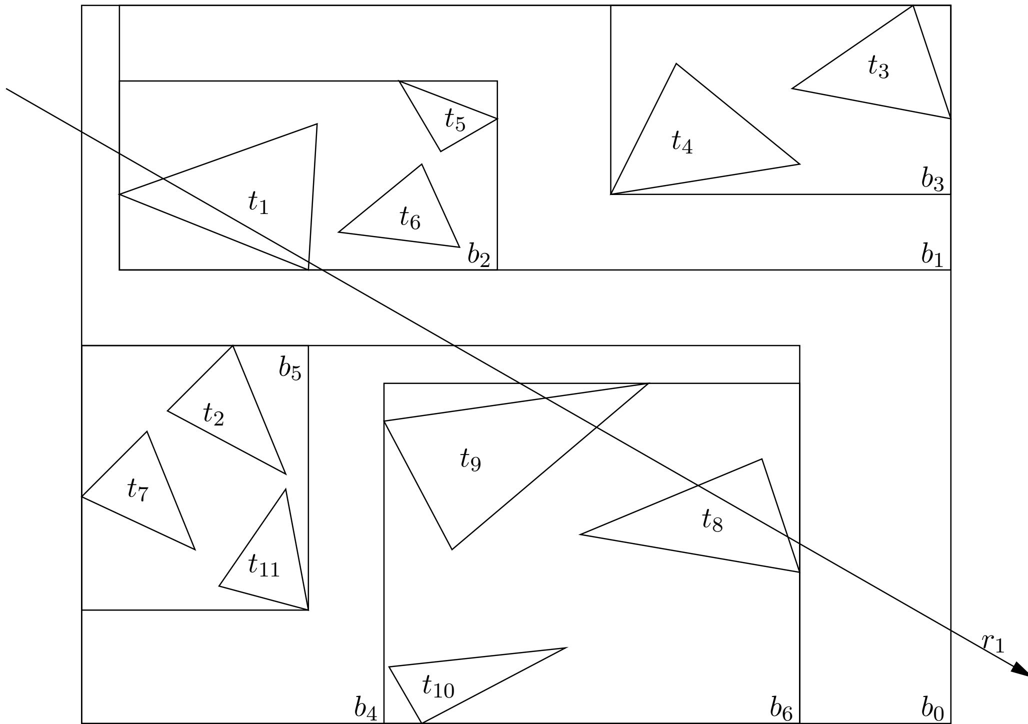


Figura 2.18: Percurso de um raio em uma BVH.

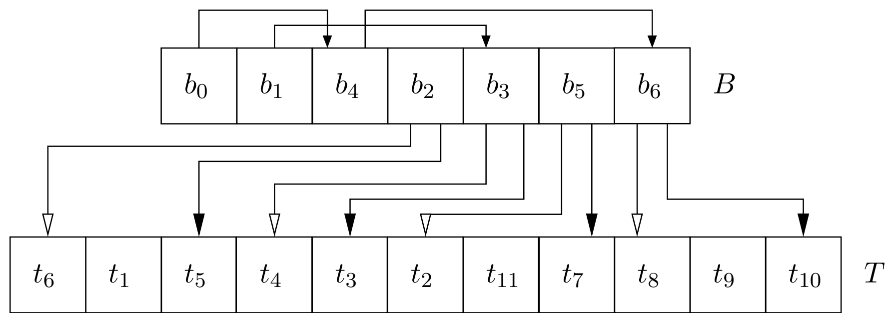


Figura 2.19: Representação da BVH da figura 2.18.

Passo 14 **N**o passo 5, o teste detecta interseção com os triângulos t_9 e t_8. Como nenhum dos dois triângulos interceptados, possui distância de interseção menor que a já obtida, t_1 é mantido como o triângulo mais próximo até este momento. O algoritmo volta ao passo 3.

Passo 15 **N**o passo 3, como a pilha está vazia, o algoritmo retorna t_1 como o triângulo mais próximo e d como sua distância de interseção.

2.4 Aspectos de Implementação

Nesta seção descrevemos as principais classes de objetos que implementam o traçador de raios em CPU. Usamos diagramas de classes UML para mostrar os relacionamentos entre as classes e descrevemos seus principais atributos e métodos em tabelas. As principais estruturas de dados utilizadas pelo traçador de raios são apresentadas em um conjunto de figuras, a fim de facilitar o entendimento.

O traçador de raios é executado através de linha de comando, usando-se a sintaxe `yart <scenefile.scn>`. O parâmetro `<scenefile.scn>` indica ao traçador de raios o arquivo descritor de cena a ser utilizado. A gramática de descrição de cena utilizada é descrita com detalhes no apêndice A.

Ao executar o traçador de raios, um objeto da classe `Application`, responsável pelo controle da aplicação é criado (ver figura 2.20).

Figura 2.20: Diagrama da classe `Application`.

Este objeto realiza a criação de um objeto da classe `SceneReader`, responsável pela leitura da cena e um objeto da classe `GLUTMainForm`, responsável pela interface com o usuário. As classes responsáveis pela leitura da cena são apresentadas na seção 2.4.1, a classe relativa ao traçador de raios é apresentada na seção 2.4.2, e as classes que realizam a apresentação na janela de visualização do usuário são descritas na seção 2.4.3.

2.4.1 Leitura da Cena

Classe `SceneReader`

Um objeto da classe `SceneReader` é responsável pela leitura da cena e criação em memória dos objetos descritos no arquivo descritor de cena. A classe `SceneReader` é derivada da classe `Reader`, que implementa um leitor de arquivo simples, conforme diagrama de classe ilustrado na figura 2.21.

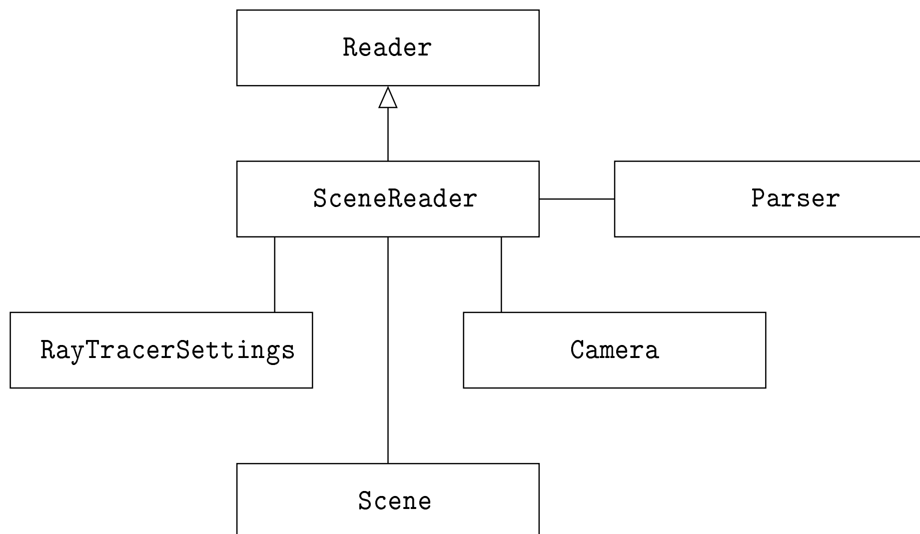


Figura 2.21: Diagrama da classe `SceneReader`.

Um objeto da classe `SceneReader` é instanciado passando-se ao construtor da classe o nome do arquivo descritor de cena que é informado na linha de comando. Este objeto é responsável pela leitura e interpretação da cena descrita, bem como a criação em memória dos objetos que representam os atores, luzes, câmera e parâmetros de configuração do traçador de raios. Na tabela 2.1 relacionamos os principais atributos e métodos da classe.

Classe `SceneReader`	
Atributos	
`Camera camera`	Câmera da cena
`RayTracerSettings RTS`	Configurações do traçador de raios
`Scene* scene`	Ponteiro para cena
Métodos	
`SceneReader(char *filename)`	Construtor da classe que recebe como parâmetro o nome do arquivo descritor da cena

Tabela 2.1: Classe `SceneReader`.

Classe `RayTracerSettings`

Um objeto da classe `RayTracerSettings`, introduzida na figura 2.21, armazena em memória os parâmetros de configuração do traçador de raios. Representamos na tabela 2.2 os principais atributos da classe.

Classe `RayTracerSettings`	
Atributos	
`uint H`	Altura da janela de projeção
`uint W`	Largura da janela de projeção
`uint maxRecursionLevel`	Nível máximo de recursão usado pelo traçador
`REAL minWeight`	Peso mínimo de um raio para que o traçado de raios seja executado

Tabela 2.2: Classe `RayTracerSettings`.

Os atributos `maxRecursionLevel` e `REAL minWeight` são critérios de parada do traçador de raios. O primeiro informa ao traçador de raios, quantas vezes um raio pode ricochetear na cena, e o segundo indica o peso mínimo que um raio deve possuir para que o traçado de raios prossiga. A medida que um raio interage com a cena, através da interseção do raio com os objetos, seu peso é reduzido conforme as características do material em cada ponto de interseção, fazendo com que o peso do raio seja reduzido até atingir o peso mínimo.

Classe `Parser`

Um objeto da classe `Parser`, representado na tabela 2.3, é responsável pela análise dos dados lidos do arquivo descritor de cena e indicação de cada termo lido, para que sejam criados, em memória, os respectivos objetos.

Classe `Parser`	
Métodos	
`void start()`	Inicia a análise do arquivo descritor de cena
`Actor* matchActor()`	Cria o ator conforme descrição encontrada
`Camera* matchCamera()`	Cria a câmera conforme descrição encontrada
`TriangleMeshShape* matchSphere()`	Cria uma malha de triângulos que representa a esfera descrita
`TriangleMeshShape* matchBox()`	Cria uma malha de triângulos que representa a caixa descrita
`TriangleMeshShape* matchMesh()`	Cria a malha de triângulos especificada

`Light* punctualLight()`	Cria uma luz pontual
`Light* directionalLight()`	Cria uma luz direcional
`Material* matchMaterial()`	Cria o material especificado
`void matchSurface(Material*)`	Cria a superfície descrita e relaciona ao material informado no parâmetro
`void environment()`	Configura as variáveis de ambiente especificadas

Tabela 2.3: Classe `Parser`.

Os termos são lidos de acordo com uma gramática de descrição de cena que descrevemos no apêndice A.

Classe `Camera`

Uma câmera é descrita por um objeto da classe `Camera`, introduzida no diagrama 2.21. A posição da câmera, `Position` é a origem dos raios traçados em direção a cena, mais especificamente em direção ao centro de cada pixel da imagem. A classe fornece atributos que especificam, além da posição, o ângulo de abertura, altura, tipo de projeção e outros, detalhados na tabela 2.4.

Classe `Camera`	
Atributos	
`Vec3 Position`	Posição da câmera
`Vec3 directionOfProjection`	Direção de projeção
`Vec3 focalPoint`	Ponto focal
`Vec3 viewUp`	Vetor UP que indica a orientação da janela de projeção
`ProjectionType projectionType`	Tipo de projeção (perspectiva, paralela)
`REAL distance`	Distância de projeção
`REAL viewAngle`	Ângulo de visão da câmera
`REAL height`	Altura da câmera
`REAL aspectRatio`	Razão de aspecto (W/H, onde W é a largura e H a altura da janela de projeção)
`REAL F`	Distância do plano de corte de frente do volume de vista
`REAL B`	Distância do plano de corte de fundo do volume de vista
Métodos	
`void azimuth(REAL)`	Rotaciona a posição da câmera sobre o vetor UP, centrado no ponto focal
`void elevation(REAL)`	Rotaciona a posição da câmera sobre o produto cruzado da normal ao plano de vista e o vetor UP, centrado no ponto focal
`void rotateYX(REAL, REAL`	Composição do azimuth de ay com a elevação de ax (em graus)
`void roll(REAL)`	Rotaciona o vetor UP ao longo da normal ao plano de vista
`void yaw(REAL)`	Rotaciona o ponto focal sobre o vetor UP, centrado na posição da câmera
`void pitch(REAL)`	Rotaciona o ponto focal sobre o produto cruzado do vetor UP e a normal do plano de vista, centrado na posição da câmera

`void zoom(REAL)`	Muda o angulo de visão da câmera, de forma que a janela de vista projete mais ou menos informações. Um valor maior que 1 realiza zoom-in e menor zoom-out
`void move(REAL, REAL, REAL)`	Movimenta a câmera para as coordenadas fornecidas
`void move(Vec3&)`	Movimenta a câmera para as coordenadas fornecidas
`Vec3 worldToView(Vec3&)`	Converte coordenadas do mundo real para coordenadas da janela de projeção
`Vec3 viewToWorld(Vec3&)`	Converte coordenadas da janela de visão para coordenadas do mundo real

Tabela 2.4: Classe `Camera`.

A classe também fornece métodos para movimentação e rotação, zoom-in e zoom-out e transformações das coordenadas do mundo real para coordenadas de vista e vice-versa.

Classe `Scene`

Um objeto da classe `Scene` representa a cena lida do arquivo descritor de cena e contém uma coleção de atores e luzes, um volume que envolve toda a cena e atributos de cor, conforme ilustrado no diagrama de classe da figura 2.22.

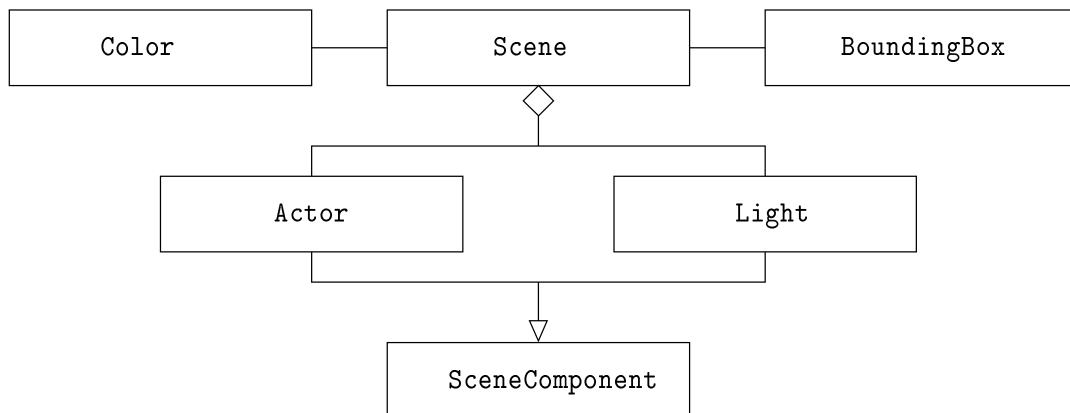


Figura 2.22: Diagrama da classe `Scene`.

A classe `Scene` fornece métodos para inserção e remoção de atores e luzes e um método para determinação do volume que envolve toda a cena. Na tabela 2.5 descrevemos os principais atributos e métodos da classe.

Classe `Scene`	
Atributos	
`Color backgroundColor`	Cor de fundo da cena
`Color ambientLight`	Cor da luz ambiente da cena
`BoundingBox boundingBox`	Volume envolvente da cena
`REAL IOR`	Coeficiente de reflexibilidade
`Actors actors`	Lista de atores da cena
`Lights lights`	Lista de luzes da cena
Métodos	
`int getNumberOfActors()`	Retorna o número de atores da cena

`int getNumberOfLights()`	Retorna o número de luzes da cena
`void addActor(Actor*)`	Adiciona um ator a lista de atores
`void deleteActor(Actor*)`	Remove um ator da lista de atores
`void addLight(Light*)`	Adiciona uma luz a lista de luzes
`void deleteLight(Light*)`	Remove uma luz da lista de luzes
`void deleteAll()`	Remove todos objetos da cena
`Actor* findActor(Model*)`	Procura por um ator na cena e retorna um ponteiro para o mesmo
`BoundingBox getBoundingBox()`	Retorna o volume envolvente relativo a cena
`BoundingBox computeBoundingBox()`	Calcula o volume envolvente capaz de enclausurar todos os objetos da cena

Tabela 2.5: Classe `Scene`.

Classe `SceneComponent`

A classe `SceneComponent` é uma classe utilizada como base para definição de outras classes, como `Actor` e `Light`, conforme ilustrado no diagrama de classe da figura 2.22, e contém um ponteiro para a cena, que é comum a todos os componentes da cena.

Classe `Actor`

Para cada ator descrito no arquivo de cena, um objeto da classe `Actor`, derivada da classe `SceneComponent`, é criado e adicionado a lista de atores da cena. Um ator é definido por um modelo geométrico e um atributo que define se o ator está visível ou não, se o mesmo é estático ou dinâmico. Definimos por estático um ator que não se altera ao longo do tempo, seja através de deformação ou mudança de posição. No diagrama de classe ilustrado na figura 2.23, representamos a classe `Actor` e seus principais relacionamentos. Na tabela 2.6 relacionamos os principais atributos e métodos da classe que define um ator da cena.

Classe `Actor`	
Atributos	
`int flags`	Atributo indica se um ator está vísivel e se é um ator dinâmico ou estático
`Model* model`	Modelo geométrico utilizado pelo ator
Métodos	
`Model* getModel`	Retorna o modelo do ator
`void setModel(Model model)`	Atribui um modelo ao ator

Tabela 2.6: Classe `Actor`.

Classe `Light`

Um objeto da classe `Light` representa uma fonte de luz presente na cena. Uma luz possui uma cor e pode ser do tipo pontual (que emite luz em todas as direções) ou direcional (que emite luz em uma direção específica). Conforme observado no diagrama de classe da figura 2.22, a classe `Light` é derivada da classe `SceneComponent`, portanto possui também um ponteiro para cena. Apresentamos na tabela 2.7, os principais atributos e métodos da classe `Light`.

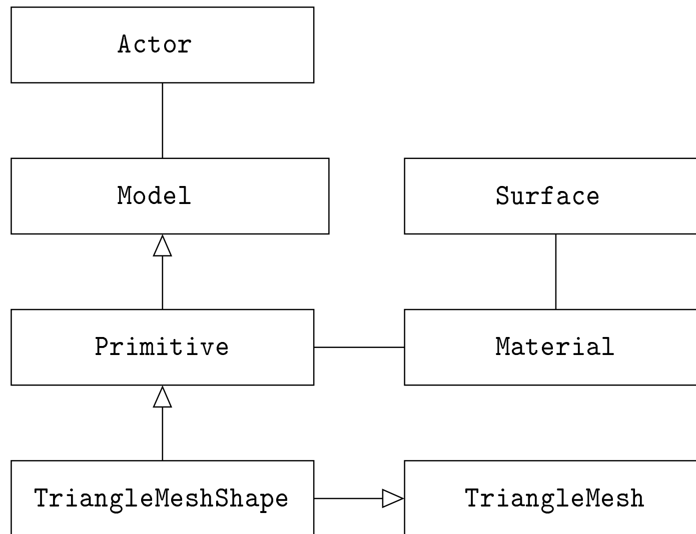


Figura 2.23: Diagrama da classe `Actor`.

Classe `Light`	
Atributos	
`Vec3 position`	Posição da fonte de luz
`Color color`	Cor da luz emitida
`bool directional`	Verdadeiro se a luz for direcional, caso contrário a luz é pontual
`int Flags`	Estado atual da luz (ligada/desligada)
Métodos	
`Color getScaledColor(REAL)`	Retorna um escalar que representa a cor
`void lightVector(Vec3&, Vec3, REAL &)`	Devolve o vetor luz (posição, cor, flags e distância)

Tabela 2.7: Classe `Light`.

Classe `Model`

A classe `Model` é uma classe que define métodos virtuais para cálculo de interseção de raios com o modelo, cálculo do vetor normal no ponto de interseção, devolução do volume envolvente do modelo e atribuição do material que compõe o modelo, conforme apresentamos na tabela 2.8.

Classe `Model`	
Atributos	
`virtual bool intersect(Ray&, IntersectInfo&)`	Método para cálculo de interseção de um raio com o modelo
`virtual Vec3 normal(IntersectInfo&)`	Devolve o vetor normal no ponto de interseção do raio com o modelo
`virtual Material* getMaterial()`	Devolve o material que compõem o modelo
`virtual BoundingBox getBoundingBox()`	Devolve o volume que envolve o modelo
`setMaterial(Material*)`	Atribui um material ao modelo

Tabela 2.8: Classe `Model`.

Classe `Primitive`

Surface surface	Superfície que compõe o material

Tabela 2.9: Classe Material.

O atributo surface contém os dados que descrevem as características da superfície do material.

Classe Color

Uma cor no sistema RGB é representada através de um objeto da classe Color. A classe Color faz uso do sistema $< r, g, b >$ de composição de cores, que indica respectivamente a intensidade de vermelho, verde e azul, para representar uma cor, e fornece métodos para soma, subtração, multiplicação, comparação e atribuição de cores.

Classe Surface

As características da superfície de um material são utilizadas durante o traçado de raios para determinar se novos raios (de reflexão e refração) devem ser gerados no ponto de interseção com o ator, para determinar a intensidade da luz refletida ou que tenha sofrido refração, etc. Estes valores alimentam as fórmulas derivadas da física e simplificadas no modelo de iluminação de Phong [34] que utilizamos neste trabalho. Representamos os principais atributos da classe Surface na tabela 2.10.

Classe Surface	
Atributos	
Color ambient	Cor ambiente
Color diffuse	Cor difusa
float shine	Brilho-Expoente especular local
Color spot	Cor local especular
Color specular	Cor especular
Color transparency	Cor de transparência
float IOR	Índice de refração

Tabela 2.10: Classe Surface.

Classe `BoundingBox`

Como podemos observar no diagrama 2.22, uma cena contém uma referência a um objeto da classe `BoundingBox`, que representa o volume que envolve toda a cena. Um objeto deste tipo possui os atributos `p1` e `p2`, para identificar o volume envolvente, conforme demonstrado na figura 2.24.

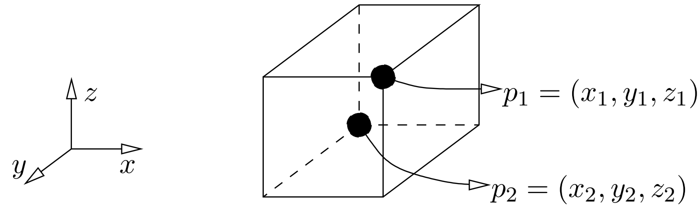


Figura 2.24: AABB.

Como já descrevemos o objetivo de termos um volume envolvente é promovermos a aceleração dos cálculos de interseção dos raios com os atores da cena, pois o cálculo de interseção de um raio com os planos que definem o volume é mais simples do que com os atores, e quando a interseção não é encontrada podemos descartar o teste com todos os atores que estiverem contidos no volume. A classe também fornece um método para cálculo de interseção de um raio com o volume.

Classe `Intersector`

A classe `Intersector` define métodos para cálculo da interseção dos raios gerados com os objetos da cena e serve de classe base para a classe `BVH`, que representa a estrutura de aceleração da cena, conforme ilustrado no diagrama de classe da figura 2.25.

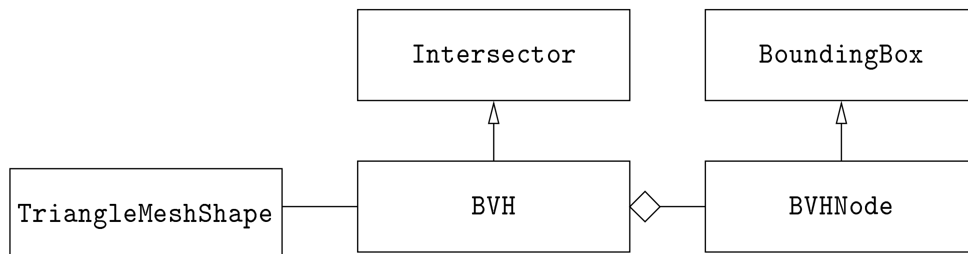


Figura 2.25: Diagrama da classe BVH.

Classe `BVH`

Após a geração da malha de triângulos que representa cada um dos atores, o algoritmo realiza a geração da lista de triângulos de toda a cena, ou seja, insere os triângulos de todos os atores em um único vetor de triângulos. Com o vetor de triângulos pronto, o algoritmo realiza a geração da estrutura hierárquica de aceleração (BVH), organizando os triângulos conforme sua localização espacial, para fornecer aceleração ao algoritmo de traçado de raios. Primeiramente o algoritmo enclausura todos os triângulos em um único volume, e posteriormente subdivide este volume de forma sucessiva e binária, formando novos volumes. A BVH é armazenada em memória na forma de um vetor de elementos do tipo `BVHNode`, e armazenada no atributo `nodes` da classe `BVH`, conforme demonstramos na figura 2.26.

Um nó `nodes`$[i]$, $0 \leq i \leq NB - 1$, é um nó interno, quando possuir nós filhos e enclausurar outros AABBs, ou um nó folha, quando o mesmo enclausurar apenas triângulos. Quando um nó

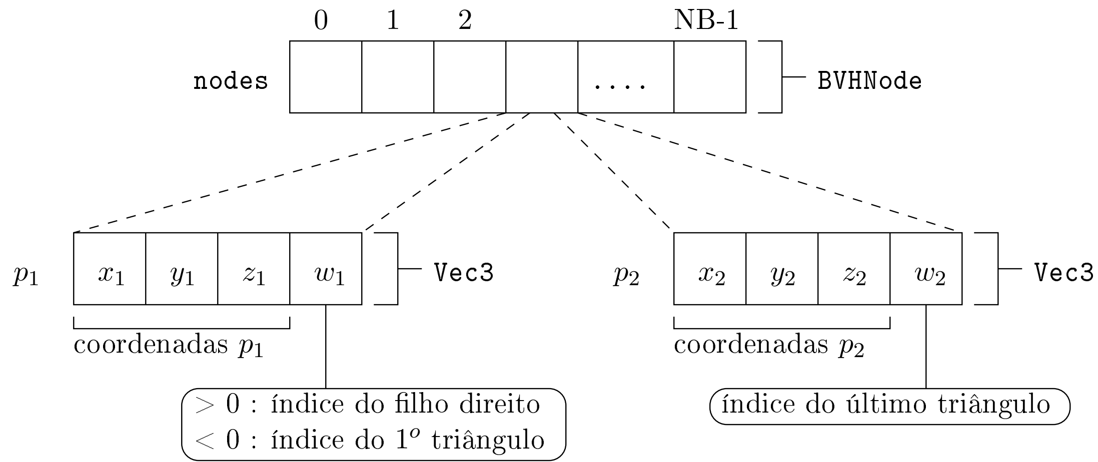


Figura 2.26: Estrutura de dados para BVH.

`nodes`$[i]$ for interno, armazenamos em w_1 o índice de seu filho direito e o filho esquerdo é conhecido por estar armazenado em `nodes`$[w_1 - 1]$. Quando um nó for folha, armazenamos em w_1, o índice do primeiro triângulo no vetor de triângulos e em w_2 o índice do último triângulo. Ilustramos na tabela 2.11 os principais atributos e métodos da classe `BVH`.

Classe `BVH`	
Atributos	
`int maxLevel`	Altura máxima da BVH
`TriangleMeshShape* mesh`	Malha de triângulos
`BVHNode* nodes`	Ponteiro para o nó raiz da BVH
Métodos	
`void recursiveBuild(BVNNode&, int)`	Constroi recursivamente a BVH, a partir do nó informado

Tabela 2.11: Classe `BVH`.

Classe `BVHNode`

A classe `BVHNode` é derivada da classe `BoundingBox`, conforme visualizamos na figura 2.25. Um objeto da classe `BVHNode` conhece seu filho esquerdo e direito e contém um AABB, conforme detalhes já fornecidos na descrição da classe `BVH` e na figura 2.26.

Classe `TriangleMesh`

A classe `TriangleMesh` define uma malha de triângulos. Uma malha de triângulos é composta por uma coleção de triângulos, vértices e vetores normais e fornece métodos para o cálculo de interseção de um raio com a malha e cálculo do vetor normal no ponto de interseção de um raio com um triângulo da malha.

Como observamos na figura 2.27, a malha de triângulos é representada por um conjunto de vetores. O vetor `triangle` é um vetor de objetos da classe `Triangle`, que é composto por dois

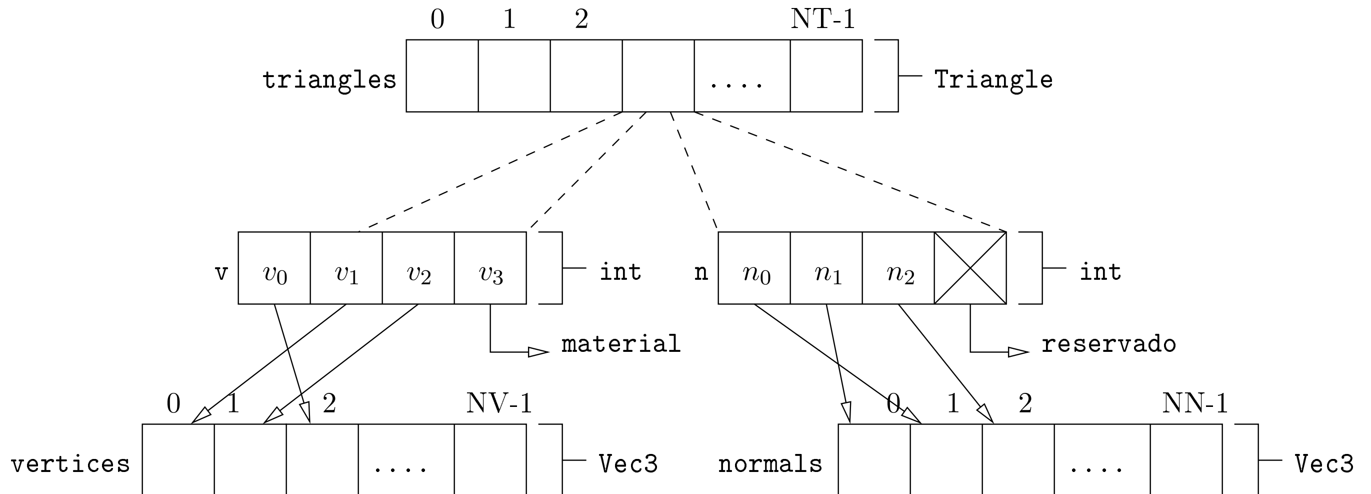


Figura 2.27: Estrutura de dados para malha de triângulos.

vetores de inteiros, `v`, que representa o conjunto de três vértices de cada triângulo, e `n`, que representa o conjunto das normais de cada um dos vértices. Os elementos de `v` e `n` não contém diretamente as informações dos vértices e respectivas normais, mas armazenam os índices dos vértices e das normais dos vértices, que estão representados nos vetores `vertices` e `normais` respectivamente.

Para representar um triângulo precisamos de apenas três vértices e, consequentemente, três normais de vértices, mas como observamos, um elemento foi adicionado a cada um dos vetores `v` e `n`, com o objetivo de obter alinhamento e acesso coalescido a memória. Aproveitamos o elemento v_3 para armazenar o índice do material relativo ao triângulo e o elemento n_3 é usado apenas para fins de alinhamento.

Os vetores `vertices` e `normals` são vetores de elementos do tipo `Vec3`, e contém todo o conjunto de vértices e normais de toda a malha. O tipo `Vec3` é fornecido pela extensão de CUDA, e é um vetor de quatro elementos de ponto flutuante (x, y, z, w), onde x, y, z representam as coordenadas tridimensionais de um ponto e w é utilizado para fins de alinhamento.

Classe `TriangleMeshShape`

A classe `TriangleMeshShape` é derivada das classes `Primitive` e `TriangleMesh` e é utilizada para representar uma malha de triângulos que contém um material. A classe fornece métodos para realização de uma cópia do objeto representado, obtenção do volume envolvente relativo a malha, cálculo de interseção com um raio, cálculo do vetor normal no ponto de interseção e atribuição das características relativas ao material que compõe a malha.

2.4.2 Aplicação

Classe `Renderer`

Um objeto da classe `Renderer`, ilustrado no diagrama de classe 2.28, é o objeto responsável pela renderização da cena.

Os principais atributos e métodos da classe são apresentados na tabela 2.12.

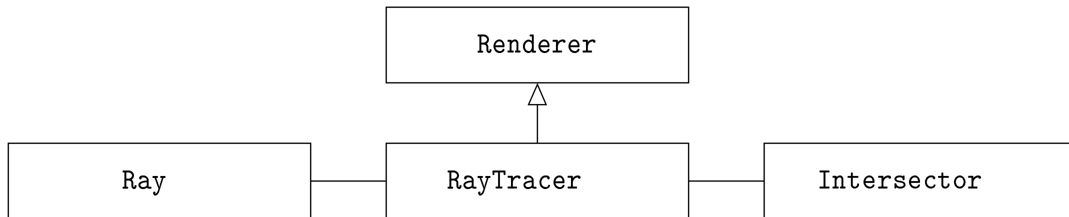


Figura 2.28: Diagrama da classe `RayTracer`.

Classe `Renderer`	
Atributos	
`Scene* scene`	Ponteiro para a cena
`Camera* camera`	Ponteiro para a câmera
`int W`	Largura da janela de renderização
`int H`	Altura da janela de renderização
`Camera* defaultCamera`	Câmera padrão
Métodos	
`virtual void render()`	Realiza a renderização da cena

Tabela 2.12: Classe `Renderer`.

Classe `Ray`

Definimos um raio de luz da cena através de um objeto da classe `Ray`, que contém uma origem (`Vec3 origin`) e uma direção (`Vec3 direction`).

Classe `RayTracer`

Um objeto da classe `RayTracer`, introduzida no diagrama 2.28, representa o traçador de raios, e é responsável pelo traçado dos raios de *pixel*, cálculos de interseção, traçado dos raios de sombra, refração, reflexão e determinação final da cor do *pixel*. Na tabela 2.13 apresentamos os principais atributos desta classe.

Classe `RayTracer`	
Atributos	
`Ray pixelRay`	Raio de luz
`int maxRecursionLevel`	Nível máximo de recursão
`REAL minWeight`	Peso mínimo de um raio para que o mesmo seja traçado
`Intersector* intersector`	Ponteiro para o objeto responsável pelo cálculo de interseção
Métodos	
`setPixelRay(REAL, REAL)`	Atribui o raio de pixel ao traçador
`virtual REAL trace(Ray&, Color&, int, REAL)`	Realiza o traçado de um raio, retornando a cor do mesmo
`virtual bool intersect(Ray&, IntersectInfo&, REAL)`	Devolve os dados relativos a interseção do raio traçado com a cena
`virtual bool notShadow(Ray&, IntersectInfo&, REAL, Color&, bool)`	Devolve falso se o raio intercepta um objeto antes de atingir a luz
`virtual Color shoot(REAL, REAL)`	Dispara um raio de *pixel* nas coordenadas indicadas e retorna a cor RGB deste raio
`virtual Color shade(Ray&, IntersectInfo&, int, REAL)`	Traça um raio de luz/sombra a partir do ponto de interseção
`virtual Color background()`	Devolve a cor de fundo da cena
`RayTracer(Scene&, Camera*=0,Intersector*=newBVH())`	Construtor da classe

Tabela 2.13: Classe `RayTracer`.

2.4.3 Visualização

O traçador de raios construído possui uma interface simplificada, que tem o objetivo de permitir a interação do usuário com o aplicativo e a apresentação da imagem gerada pelo traçado de raios. Apresentamos na figura 2.29 as principais classes relativas a visualização e seus relacionamentos.

A interface é composta por uma janela principal (objeto da classe `GLUTMainForm`, derivada da classe `GLUTWindow`) que pode conter um conjunto de janelas de visualização de imagens, representadas por objetos da classe `GLUTView`, derivada da classe `GLUTRendererWindow`. Uma janela de visualização contém um menu, representado por um objeto da classe `GLUTMenu`, que permite que o usuário realize uma série de escolhas, dentre elas : escolha da versão de traçado de raios a ser executada; execução do traçado de raios; geração da imagem e do volume envolvente da cena no formato fio de arame (realizada por um objeto da classe `PolyRenderer`; gravação da imagem gerada em arquivo externo. A janela de visualização também permite que o usuário, através do arrasto do mouse, movimente a posição da câmera. A janela possui um buffer para armazenamento da imagem gerada (objeto da classe `OpenGLImage`) e um ponteiro para o objeto da classe `RayTracer`, responsável pelo traçado dos raios.

2.5 Comentários Finais

Neste capítulo explicamos o modelo de renderização de imagens através do traçado de raios. Apresentamos os principais elementos que compõe este modelo, que é derivado do modelo que chamamos de "'orífico de alfinete'", como câmera, plano de projeção, atores e luzes, volume de vista, etc. A

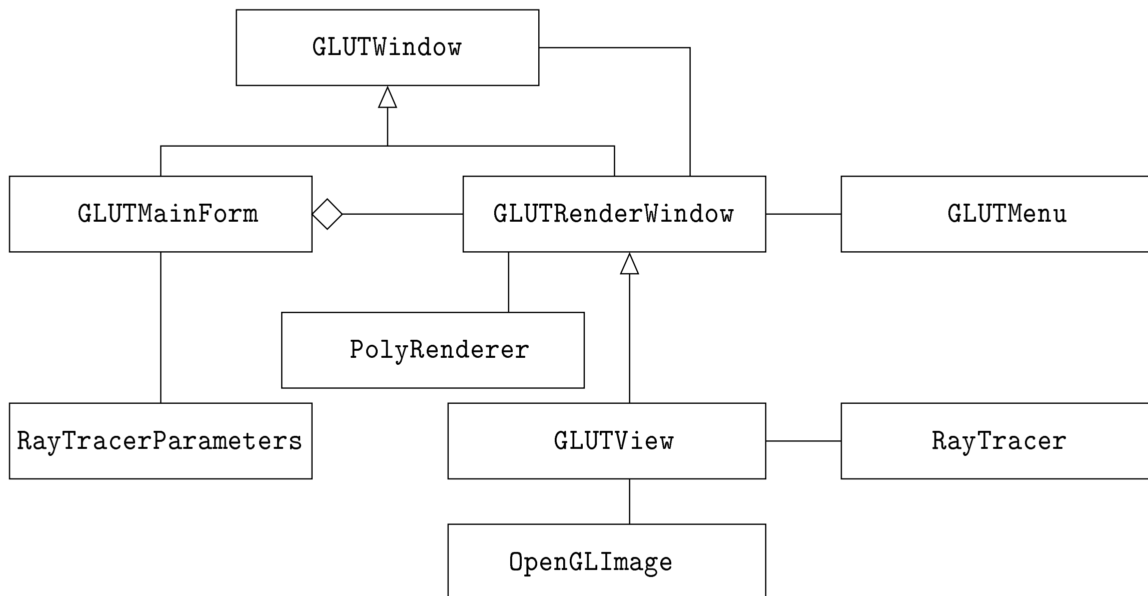


Figura 2.29: Diagrama da classe `GLUTMainForm`.

técnica de traçado de raios é uma técnica de amostragem e por conta deste fato, está sujeita a problemas como aliasing, que podem ser resolvidos através do traçado de raios distribuído. Também realizamos uma discussão sobre a performance de algoritmos de traçado de raios, onde observamos que, dado o volume de raios a serem traçados em uma cena e a quantidade possível de triângulos da mesma, o algoritmo de força bruta, ou seja, o teste de cada raio com cada um dos triângulos da cena, é uma solução inviável. Revisamos a literatura apresentando os principais trabalhos relacionados, principalmente aqueles ligados a adoção de estruturas hierárquicas para aceleração de cálculos de interseção destes raios, onde destacamos as estruturas BVH e kd-tree, como as principais estruturas de aceleração. A construção destas estruturas hierárquicas, que normalmente são binárias, ou seja, dividem a cena sempre em dois volumes a cada iteração, depende da adoção de alguma técnica para divisão dos volumes, para que seja possível produzir árvores binárias tão balanceadas quanto possível. Como também é inviável o uso de algoritmos de força bruta para determinação do plano de corte mais eficiente, dado o volume de cálculos necessários, adotamos a heurística SAH (Surface Area Heuristic) para escolha do plano de corte e construção da estrutura espacial. Posteriormente apresentamos os algoritmos de construção e percurso da estrutura hierárquica escolhida (BVH) e alguns exemplos. Também mencionamos, que dada nossa escolha de representação da BVH na forma de um vetor de nós que contém índices para um vetor de triângulos, o vetor de triângulos deveria ser reordenado a cada novo nó, para que fosse possível armazenar os triângulos de um nó de maneira consecutiva no vetor de triângulos e conseguir identificá-los apenas com dois índices, um para o primeiro triângulo e um para o último.

Na seção 2.4 descrevemos os aspectos relativos a implementação dos algoritmos mencionados, onde descrevemos o arquivo de entrada que é fornecido ao aplicativo, que é baseado na gramática definida no apêndice A, e descrevemos as principais classes de objetos que implementam o traçado de raios, e seus principais atributos e métodos. Dividimos a seção em três partes, uma dedicada às classes de leitura da cena, outra às classes responsáveis mais diretamente ao traçado dos raios e outra à visualização. O objetivo deste capítulo foi preparar o leitor para o capítulo seguinte, relativo a implementação do traçado de raios em GPU, através de algoritmos paralelos que procuram tirar proveito da grande capacidade aritimética destes dispositivos para aceleração do traçado de raios.

Capítulo 3

Traçado de Raios em GPU

3.1 Introdução

Como observado no capítulo 2, traçadores de raio são computacionalmente intensivos no aspecto aritmético, principalmente pela grande quantidade de cálculos de interseção raio/objeto. Tendo em vista a grande capacidade aritmética das GPUs atuais, sua constante evolução e o seu baixo custo, é natural pensarmos em desenvolver um traçador de raios capaz de aproveitar o poder computacional das GPUs. Em um primeiro momento, desenvolvemos versões onde a interpretação da gramática de cena e geração da BVH é realizada em CPU e o traçado de raios primários e secundários em GPU. Posteriormente, migramos também a geração da BVH para dentro da GPU e adicionamos a capacidade de tratamento de cenas dinâmicas. Podemos entender por cenas dinâmicas, cenas onde os atores alteram sua posição ou deformam-se ao longo do tempo. Neste capítulo discutiremos sobre a implementação do traçado de raios em GPU. Na seção 3.2 revisamos a literatura e descrevemos os principais trabalhos relacionados ao traçado de raios em GPU. Na seção 3.3 apresentamos um resumo da arquitetura CUDA, com o objetivo de fornecer ao leitor que desconhece esta arquitetura, um conhecimento suficiente para o entendimento das próximas seções. Na seção 3.4 descrevemos os detalhes de implementação do traçado de raios em GPU, onde apresentamos as principais estruturas de dados e todas as versões de traçado de raios implementadas. Na seção 3.5 apresentamos a descrição do algoritmo de construção da estrutura de aceleração de cálculo de interseção para GPU.

3.2 Trabalhos Relacionados

Nesta seção apresentamos os principais trabalhos já desenvolvidos em traçado de raios para GPU. Há várias abordagens para se mapear um traçador de raios para GPU. Purcell e outros [38, 39] fornecem uma discussão sobre traçadores de raios para um modelo de computação paralela particular para GPU e apresentam uma comparação entre um traçador de raios para GPU e um para CPU, também demonstrando uma análise dos principais requisitos de banda de memória e capacidade de processamento.

Os primeiros passos para a execução de um traçador de raios em GPU foram feitos em Ray Engine [7], o qual apresenta uma implementação em GPU das interseções raio/triângulo. O envio da geometria para a GPU também tornou-se um gargalo, tendo em vista que o processo de cópia é lento devido às características do hardware, o que pode ser contornado transportando-se todo o processamento para a GPU. Desta forma, a geração de raios primários, o percurso da estrutura de aceleração, o teste de interseção, o sombreamento e a geração de raios secundários são realizados na GPU, evitando o tempo gasto anteriormente com transferência entre a memória do host e o disposi-

tivo. Essa abordagem tornou-se a base de muitas outras implementações encontradas na literatura [8, 13, 24]. Essas tentativas possuíam desempenho inferior a implementações em CPU, tendo como principal obstáculo a capacidade limitada das GPU disponíveis e sua baixa programabilidade.

Inicialmente, a divisão espacial em grades regulares foi amplamente usada em traçadores de raios baseados em GPU, devido à sua simplicidade e relativa facilidade de paralelização. Um dos problemas enfrentados para programação de estruturas hierárquicas em GPU é a inexistência de recursão e a dificuldade de implementação, em GPUs não CUDA, de uma pilha para resolução deste problema. Ernst [13] demonstrou ser possível a construção de uma pilha em GPU, permitindo assim a transformação de algoritmos recursivos em iterativos, o que abriu caminho para implementação de outras estruturas de aceleração em GPU.

Posteriormente, Foley e Sugerman [14] apresentaram dois algoritmos para percurso de uma kd-tree em GPU, chamados *kd-restart* e *kd-backtrack*, que evitam o uso de uma pilha. Demonstrou-se que, para cenas com muitos objetos, em diferentes escalas, estes algoritmos apresentam aceleração de até oito vezes em relação às grades uniformes, porém ainda sem superar o desempenho dos traçadores de raio voltados para CPU. Ambos algoritmos apresentados, apesar de evitar o uso de pilha, acabam por revisitar alguns nós. Estes passos redundantes constituem um dos principais motivos do baixo desempenho obtido por estes dois algoritmos [36].

O trabalho de Horn e outros [22] estende o trabalho de Foley, fazendo uso das instruções de desvio e iteração disponibilizadas nas novas versões de dispositivos gráficos (ainda não CUDA) e apresentando um traçador de raios interativo com desempenho próximo a quinze milhões de raios por segundo. Tal desempenho foi obtido através da adição de uma pequena pilha para atravessamento da árvore, evitando assim a revisita à grande parte dos nós gerados nos algoritmos *kd-restart* e *kd-backtrack*. No mesmo ano, Popov e outros [36] apresentaram uma nova implementação, eliminando totalmente a necessidade de uma pilha e reduzindo o número de passos por raio para o atravessamento da árvore.

Com o argumento de que implementações baseadas em kd-trees parecem não ser adequadas para cenas dinâmicas, pela necessidade de reconstrução da árvore a cada quadro, e que kd-trees exigem mais memória que BVHs, Güenther e outros [18] apresentaram uma implementação baseada em BVH para GPU. A implementação faz uso de uma pilha compartilhada e pacotes de raios que atravessam a estrutura em conjunto. O método mostrou-se adequado para GPU por necessitar de menor quantidade de registradores e também usar menos de 1/3 da memória exigida por uma kd-tree, dependendo da cena. A implementação de Güenther tira proveito de recursos novos, disponibilizados a partir dos modelos G80 da NVIDIA (CUDA), como a possibilidade de leitura e escrita em qualquer posição da memória da GPU e a possibilidade de sincronização e comunicação entre threads.

Uma das características desejáveis para obtenção de performance é o uso de acesso coalescido a memória. Na GPU, uma memória lida por threads consecutivas de um mesmo warp pode ser combinada pelo hardware em várias leituras simultâneas. Para obter-se acesso coalescido (agrupado), as threads de um mesmo warp devem realizar a leitura da memória em ordem – Se tivermos por exemplo um vetor chamado $data[]$ e precisarmos ler vários elementos deste vetor, començando no deslocamento n, então a thread 0 do warp deve ler $data[n]$, a thread 1 deve ler $data[n + 1]$ e assim por diante. Leituras de 32 bits, realizadas desta forma, são executadas simultaneamente pelo dispositivos, e combinadas em várias leituras de 384 bits, a fim de produzir um acesso eficiente ao barramento de memória.

A maioria dos trabalhos sobre traçado de raios em GPU encontrados na literatura tem como foco a estrutura de dados usada para aceleração dos testes de interseção raio/objeto, o que é justificável, pois este é o principal gargalo do algoritmo. Contudo, a paralelização em si do traçado de raios em GPU não é menos importante, não somente por não ser possível a programação de funções recursivas

no dispositivo, mas principalmente porque, em CUDA, vários aspectos devem ser considerados para uma implementação eficiente, tais como acesso coalescido (citado acima) à memória global, conflitos de bancos na memória compartilhada, divergência no processamento de threads (notadamente do mesmo warp), transferência de dados entre host e GPU, uso de cache, entre outros.

Discussões sobre esses aspectos de implementação são mais escassas na literatura, porém negligênciá-los pode resultar na obtenção de baixo desempenho, por isto, consideramos esta discussão uma das contribuições deste trabalho.

3.3 CUDA

CUDA [33] é uma nova arquitetura de hardware e software desenvolvida pela NVIDIA que permite tratar de computação paralela sem a necessidade de um mapeamento para uma API gráfica. CUDA oferece às aplicações computacionalmente intensivas acesso aos recursos de processamento da GPU através de uma nova interface de programação, fornecendo maior desempenho e simplificando o software de desenvolvimento através do uso da linguagem C. A tecnologia CUDA está disponível para GPUs NVIDIA a partir da série 8 da GeForce [30] e também para os produtos Quadro FX 5600/4600 e GPU Tesla.

O software CUDA é composto de diversas camadas como ilustrado na Figura 3.1, formado por um controlador de hardware, uma API e duas bibliotecas matemáticas de alto nível: CUFFT e CUBLAS [31].

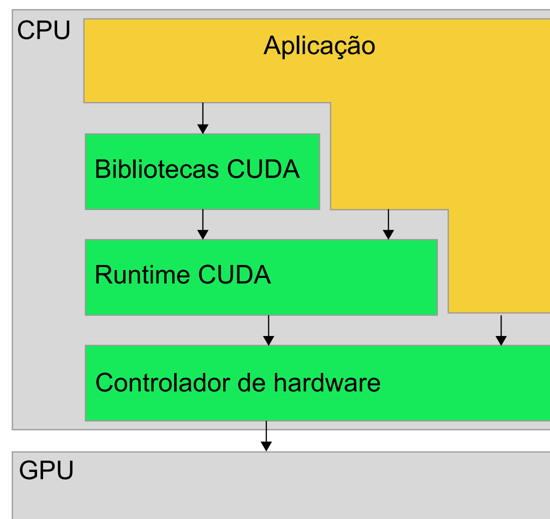


Figura 3.1: Camadas CUDA.

CUDA fornece endereçamento da memória de acesso aleatório dinâmico (DRAM) que permite uma maior flexibilidade na programação, possibilitando o uso de operações de leitura e escrita em qualquer parte da memória, como em CPU, o que era impossível de ser realizado em dispositivos anteriores. Também está disponível um cache de dados paralelos, ou memória compartilhada, com acessos de leitura e escrita bastante eficientes, os quais podem ser utilizadas pelos processadores para compartilhar dados.

Apesar da capacidade de processamento oferecido, a arquitetura CUDA possui a maioria dos seus transistores voltados para a unidade aritmética e lógica; assim, a programação deve privilegiar a computação aritmética, o balanceamento entre as threads e evitar a troca de dados entre a memória da CPU (o termo host será normalmente usado neste texto para se referir a CPU) e a memória

do dispositivo [32]. Uma introdução mais detalhada sobre a arquitetura CUDA e sua programação pode ser obtida em [31, 33].

3.3.1 A GPU como Coprocessador da CPU

Em CUDA, a GPU é vista como um dispositivo computacional capaz de executar um grande número de threads em paralelo, operando como um coprocessador da CPU. Assim, porções de dados (ou funções) paralelizáveis e que consumam grandes quantidades de recursos computacionais, são carregados na GPU, aliviando a carga da CPU. Para tal, é necessário que a função seja isolada e compilada para o hardware específico da GPU, resultando então em um programa denominado kernel (funções executadas no dispositivo são chamadas de kernels), que é mais tarde carregado e executado pelo dispositivo. Tanto o host quanto o dispositivo mantém suas próprias memórias, possibilitando no entanto, que se possa efetuar cópias de dados de uma memória para a outra, através de chamadas otimizadas presentes na API que utilizam o acesso direto à memória (DMA).

3.3.2 Grids, Blocos e Threads

Um grid é um conjunto de vários blocos responsável por executar um determinado kernel. Os blocos de um grid podem conter várias threads, como ilustrado na Figura 3.2. Em um bloco, as threads podem compartilhar dados através da memória compartilhada e sincronizar suas execuções para coordenar o acesso à memória. Podem ser especificados pontos de sincronização no kernel nos quais as threads do bloco são suspensas até que todas alcancem a mesma instrução.

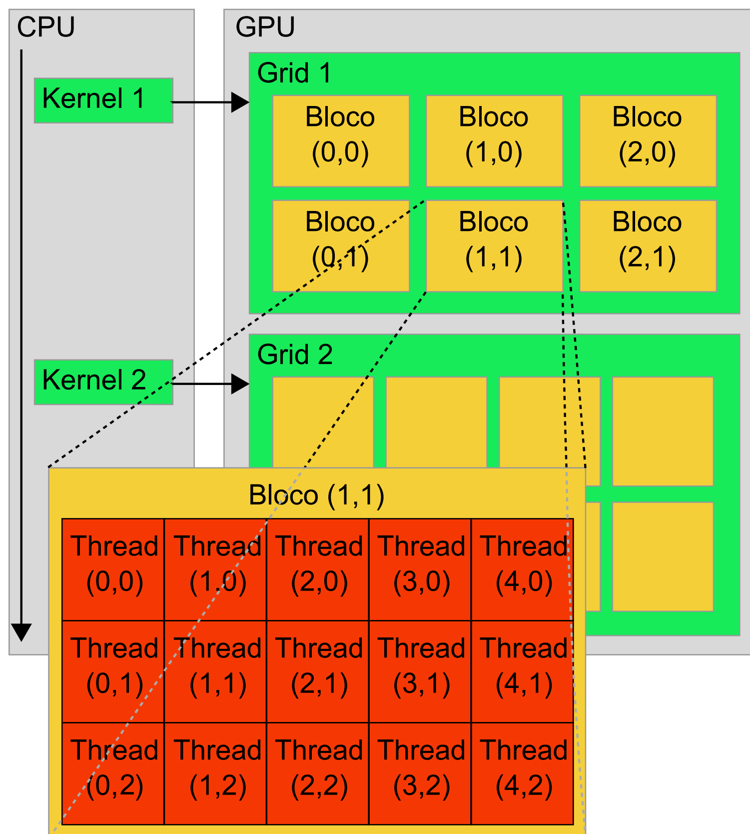


Figura 3.2: Blocos de threads.

Cada bloco possui um índice 1D ou 2D em relação ao grid ao qual pertence, e cada thread

possui um índice 1D, 2D ou 3D relativo ao bloco ao qual pertence. Estes índices são usados como identificadores de bloco ou thread, os quais podem ser empregados, por exemplo, na determinação de endereços de memória de dados manipulados por uma thread. Para um grid bidimensional de tamanho (D_x, D_y), o identificador do bloco de índices (x, y) (em relação ao grid) é $(x+y\times D_x)$. Para um bloco bidimensional de tamanho (D_x, D_y), o identificador da thread de índices (x, y) em relação ao bloco é $(x + y \times D_x)$ e, para um bloco tridimensional de tamanho (D_x, D_y, D_z), o identificador da thread de índices (x, y, z) é $(x + y \times D_x + z \times D_x \times D_y)$.

Há um limite máximo do número de threads que um bloco pode conter; no entanto, como blocos da mesma dimensão que executam o mesmo kernel são agrupados em um grid de blocos, o número de threads para uma única invocação de um kernel pode ser muito maior. Isso tem o custo de uma menor cooperação entre as threads, pois threads em blocos diferentes de um mesmo grid não podem se comunicar umas com as outras. A vantagem deste modelo é permitir que kernels sejam executados sem recompilação por diversos dispositivos com diferentes capacidades. Um dispositivo deve executar todos os blocos de um grid sequencialmente se não puder atender a todos em paralelo de uma única vez, ou deverá executá-los em uma combinação de execução paralela e sequencial.

3.3.3 Implementação do Hardware

Um dispositivo CUDA é formado por um conjunto de multiprocessadores (MPs) de arquitetura SIMD (*single instruction, multiple data*), os quais são constituídos de processadores escalares (SPs), como ilustrado na Figura 3.3. Em um dado ciclo de clock, cada processador do multiprocessador executa a mesma instrução, porém sobre dados diferentes.

Cada multiprocessador possui memória local de quatro tipos diferentes:

- Registradores locais (de 32 bits) por processador.
- Memória compartilhada acessível por todos processadores.
- Cache de memória constante, somente de leitura, compartilhado por todos os processadores.
- Cache de textura, somente de leitura, compartilhado por todos os processadores.

3.3.4 Modelo de Execução

Um grid de blocos de threads é executado no dispositivo por escalonamento dos blocos para execução nos multiprocessadores. Cada multiprocessador processa grupos de blocos, sendo um grupo após o outro. Um bloco é processado por somente um multiprocessador, portanto o espaço de memória compartilhada reside na memória compartilhada do chip, tornando o acesso à memória mais eficiente.

A quantidade de blocos que cada multiprocessador pode processar em um grupo depende da quantidade de registradores por thread e quanto de memória compartilhada por bloco estão sendo requisitados por um dado kernel, visto que os registradores e a memória compartilhada do multiprocessador são divididos por todas as threads do grupo de blocos.

Os blocos processados por um multiprocessador em um grupo são referenciados como ativos. Cada bloco ativo é dividido em grupos SIMD de (trinta e duas) threads denominados *warps*. Os warps ativos — todas pertecentes aos blocos ativos — são escalonados periodicamente por um escalonador de threads que permuta de um warp para outro a fim de maximizar o uso dos recursos do multiprocessador. Um meio warp é a primeira ou a segunda metade de um warp.

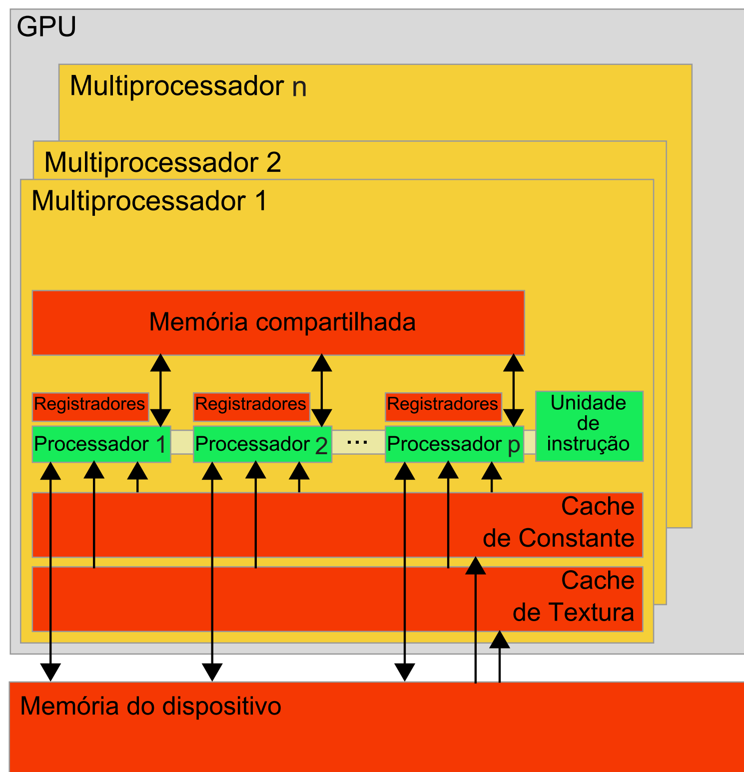


Figura 3.3: Dispositivo CUDA.

A maneira de como um bloco é dividido em warps é sempre a mesma: cada warp contém threads consecutivas, relacionado aos identificadores das threads, com o primeiro warp contendo o identificador de thread 0. A ordem do fluxo dos warps internos a um bloco é indefinido, porém suas execuções podem ser sincronizadas, coordenando acessos à memória global e compartilhada.

A ordem do fluxo dos blocos internos a um grid é indefinido e não há mecanismo de sincronização entre blocos; portanto, threads de blocos diferentes de um mesmo grid não podem se comunicar seguramente uma com a outra através da memória global durante a execução de um grid.

3.3.5 Modelo da Memória

Uma thread executada no dispositivo tem acesso à memória DRAM e memória local do chip somente através dos seguintes espaços de memória, ilustrados na Figura 3.4: registradores (leitura e escrita por thread); memória local (leitura e escrita por thread); memória compartilhada (leitura e escrita por bloco); memória global (leitura e escrita por grid); e memórias constante e de textura (somente leitura por grid). Os espaços de memória global, constante e de textura podem ser lidos ou escritos pelo host e são persistentes durante a execução do kernel na mesma aplicação.

Tendo em vista que a memória global tem maior latência e menor largura de banda comparada à memória local, essa deve ter acesso minimizado. Um padrão ideal de programação é armazenar, durante a execução, os dados da memória global na memória compartilhada, em outras palavras, cada thread de um bloco:

- Carrega os dados da memória global para a memória compartilhada;
- Sincroniza todas as threads do bloco para uma leitura correta dos dados;
- Processa os dados na memória compartilhada;

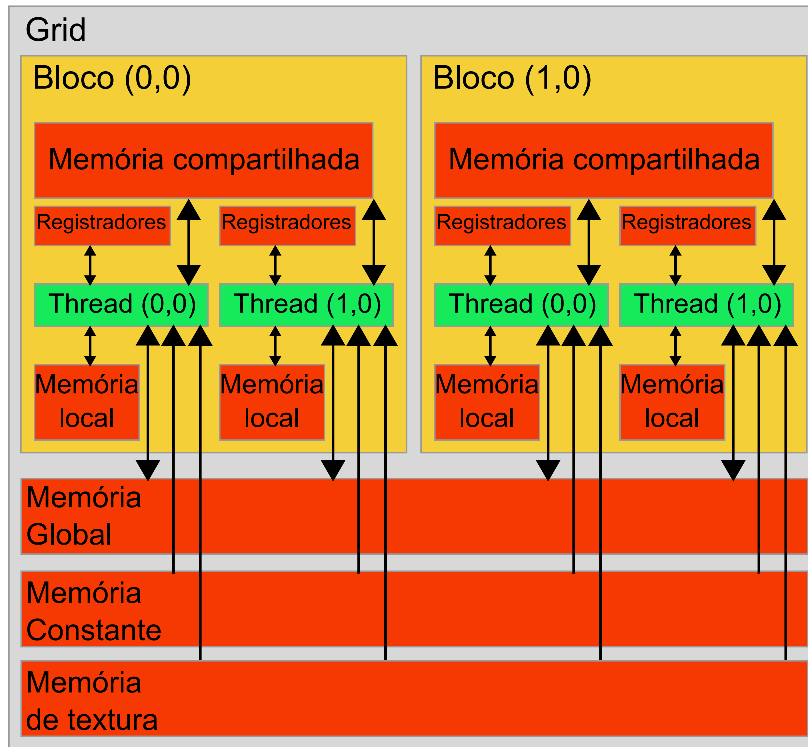


Figura 3.4: Modelo de memória.

- Sincroniza novamente, se necessário, para confirmar todos os resultados na memória compartilhada;

- Escreve os resultados de volta na memória global.

Para que esse padrão seja eficiente é necessário que as soluções desenvolvidas façam múltiplos acessos ao mesmo dado durante a execução do kernel, porém o que normalmente acontece é uma leitura única do dado, processamento e escrita, não havendo a necessidade de cópia do mesmo. Além disso, o padrão requer que o problema possa ser dividido em blocos, pois as threads são sincronizadas por bloco.

O espaço de memória constante possui cache, então um acesso a ela custa igual a um acesso à memória global somente quando ocorrer uma falha na cache. Em caso contrário o custo é o semelhante ao acesso a um registrador. O espaço de memória de textura funciona da mesma forma do supracitado e possui uma otimização da cache para localidade espacial 2D, fazendo com que threads do mesmo warp que lêem uma textura cujos endereços são próximos obtenham um melhor desempenho.

O espaço de memória compartilhada, por estar no chip, é mais rápido que os espaços de memória global e local. Desconsiderando a existência de conflitos de acessos, para todas as threads de um warp, acessar a memória compartilhada é tão rápido quanto acessar um registrador. O Registrador é a memória que possui o acesso mais eficiente, porém atrasos podem ocorrer devido às dependências de leitura após escrita e aos conflitos de acessos. As otimizações possíveis são feitas pelo compilador para evitar conflitos, diferentemente da memória compartilhada, onde quem deve evitar os conflitos, através de organização nos acessos, é o programador.

3.3.6 Interface de Programação

O objetivo da interface de programação CUDA é prover extensões da linguagem C a fim de que programadores familiarizados com a linguagem possam desenvolver mais facilmente programas para execução em GPU. Tais extensões consistem em um conjunto de declarações e sentenças que permitem ao programador selecionar quais partes de um código fonte são executados em GPU. A interface é definida também por uma biblioteca dividida em três partes:

- Componente do host, executado no host e que fornece funções para controlar e acessar um ou mais dispositivos a partir deste.
- Componente com funções específicas do dispositivo, executado na GPU.
- Componente comum a ambos (CPU e GPU), que fornece tipos de vetores e um subconjunto da biblioteca padrão C que pode ser executado tanto no código do host quanto no código do dispositivo.

Dentre as extensões da linguagem C estão os qualificadores do tipo de função, os quais servem para especificar onde uma função será executada, no host ou no dispositivo e de onde será a invocação, do host ou do dispositivo. Os qualificadores do tipo de função são:

- `__device__`: função executada no dispositivo e invocada a partir de outra função executando no dispositivo.
- `__global__`: declara a função como sendo um kernel, executada no dispositivo e invocada a partir do host.
- `__host__`: função executada no host e invocada somente pelo host. Declarar uma função somente com o qualificador `__host__` é equivalente que declarar sem nenhum qualificador; em ambos os casos a função será compilada somente para o host. Porém , se usado junto com o qualificador `__device__` a função será compilada para ambos, host e dispositivo.

Existem algumas restrições quanto aos qualificadores, sendo as mais as relevantes mencionadas a seguir:

- `__device__` e `__global__` não podem ser recursivas, não podem conter variáveis estáticas em seus corpos e não podem conter número variável de argumentos.
- `__global__` e `__host__` não podem ser usados juntos.
- Para uma invocação a uma função `__global__` é necessário especificar uma configuração de execução, onde se define o número de blocos e threads que irão executar a função (kernel). Esta invocação é assíncrona e retorna antes que o dispositivo tenha completado sua execução.

Além destes, existem também os qualificadores do tipo de variáveis:

- `__device__`: declara uma variável que reside no dispositivo. Pode ser utilizada juntamente com um dos qualificadores citados a seguir, para especificar a qual espaço de memória a variável pertence. Se nenhum estiver presente, a variável pertencerá ao espaço de memória global. Possui o tempo de vida da aplicação e é acessível por todas a threads do grid.
- `__constant__`: usada com `__device__` declara uma variável que reside na memória constante, possui o tempo de vida da aplicação e é acessível por todas as threads do grid.

- `__shared__`: usada com `__device__` declara uma variável que reside na memória compartilhada do bloco de threads, possui o tempo de vida do bloco e é acessível por todas as threads do bloco.

Uma outra extensão da linguagem C é a especificação da configuração de execução para uma chamada de uma função `__global__`. Essa configuração define as dimensões do grid e blocos para executação da função no dispositivo. A especificação é feita inserindo uma expressão na forma `<<<`D_g, D_b, N_s, S`>>>` entre o nome da função e a lista de argumentos, onde:

- D_g determina a dimensão do grid, tal que $D_g.x \times D_g.y$ é igual ao número de blocos; $D_g.z$ não é usado pois o grid é no máximo bidimensional.
- D_b especifica a dimensão do bloco de threads, tal que $D_b.x \times D_b.y \times D_b.z$ é igual ao número de threads por bloco.
- N_s e S são argumentos opcionais e indicam, resumidamente, a quantidade de memória compartilhada alocada dinamicamente por bloco e os dados na forma de stream utilizado no kernel, respectivamente. Mais detalhes em [31].

Como exemplo, a função declarada como:

```
__global__ void func(float* param);
```

deve ser invocada desta maneira:

```
func<<<Dg, Db>>>(param);
```

A função poderá falhar caso D_g ou D_b forem maiores que o máximo permitido para o dispositivo.

A quarta e última extensão são variáveis pré-definidas somente de leitura, acessíveis em toda função executada no dispostivo, que especificam as dimensões do grid e do bloco e os índices do bloco e da thread: `gridDim` (dimensão do grid); `blockIdx` (índice do bloco no grid); `blockDim` (dimensão do bloco); e `threadIdx` (índice da thread no bloco).

3.4 Traçado de Raios em CUDA

Ao programar para GPU, um dos principais aspectos a serem observados é que o acesso a memória deve se dar de forma coalescida, caso contrário o acesso a memória é serializado pela GPU e a performance é degradada. Outro aspecto diz respeito ao balanceamento de carga das threads, que como foi explicado na Seção 3.3, pode penalizar o desempenho, dado que uma thread não é liberada até que todas as threads do mesmo bloco tenham terminado sua execução. Isso significa que quando algumas threads terminam sua execução muito antes das outras, estas ficam ociosas até que todas as threads do mesmo bloco tenham finalizado.

A possibilidade de termos desbalanceamento pode ser observada em vários pontos do traçado de raios. Por exemplo, ao traçar os raios primários, alguns destes raios não interceptam objetos da cena e retornam cor de fundo. Neste caso, estas threads ficarão ociosas enquanto outras seguem o algoritmo para determinar se o ponto de interseção é iluminado ou está na sombra. Isso também ocorre durante o traçado dos raios secundários, pois nem todos os objetos interceptados irão gerar raios secundários, e desta forma também aguardarão o traçado dos raios que interceptaram objetos reflexivos ou transparentes e assim sucessivamente.

Nesta seção detalharemos as estruturas de dados usadas em GPU, que são comuns a todos os três algoritmos implementados e posteriormente apresentaremos os algoritmos e suas estruturas de dados particulares.

Como já citamos no capítulo 2, as malhas de triângulos de todos os atores da cena são agrupadas em uma única malha, contendo todos os vértices, normais de vértices e a lista de triângulos que definem a geometria da cena. Em GPU representamos a BVH através de um um objeto da classe `BVHData`, conforme descrevemos na tabela 3.1, que faz uso de estruturas de dados já definidas no capítulo 2.

Classe `BVHData`	
Atributos	
`TriangleMesh::Data`	geometry
`BVHNode* nodes`	Ponteiro para o nó raiz da BVH

Tabela 3.1: Classe `BVHData`.

A BVH contém os dados da malha de triângulos no atributo `geometry`, que contém um conjunto de vetores relativos aos vértices, normais e triângulos, e um ponteiro para o primeiro nó da BVH, o elemento `nodes[0]`.

Estes vetores são gerados em CPU e copiados diretamente para a memória global do dispositivo.

Os dados das luzes da cena são armazenados em memória global da GPU, na estrutura de vetores ilustrada na figura 3.5. Para a i-ésima luz da cena, $0 \leq i < \text{NL}$, onde NL é o número de luzes

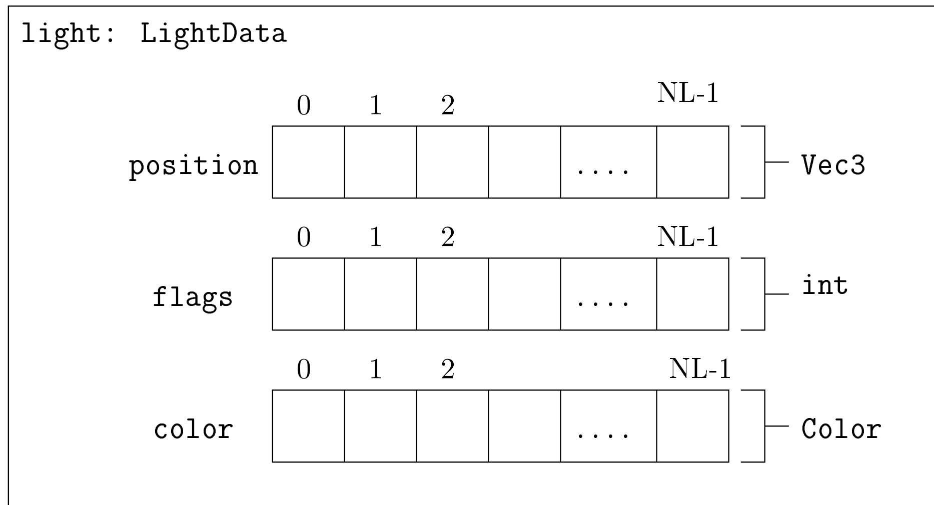


Figura 3.5: Estrutura de dados de luzes em GPU.

da cena, `position`[i] armazena a posição, `flags`[i] armazena o estado (ligada ou não, pontual ou direcional e o expoente de decaimento ou *falloff*) e `color`[i] armazena a cor da luz. Note que os elementos dos vetores da estrutura da figura 3.5 possuem quatro (`flags`) e dezesseis (`position` e `color`) bytes. Além disto, as funções de alocação de memória global de CUDA garantem que tais vetores estão alinhados em segmentos múltiplos de quatro e dezesseis, respectivamente. Isto garante acesso coalescido à memória global da GPU por threads de um mesmo *warp*, sendo este um dos principais aspectos para aumento de performance.

Os dados dos materiais, que são indexados pelos vértices dos triângulos, conforme a Figura 2.27, são armazenados na memória global da GPU em uma estrutura de vetores, conforme ilustrado na figura 3.6. Para o i-ésimo material, $0 \leq i < \text{NM}$, onde NM é o número de materiais da cena, `ambient`[i] armazena a cor de reflexão difusa ambiente, `diffuse`[i] armazena a cor de reflexão difusa, `shine`[i], o expoente de reflexão especular, `spot`[i], a cor de reflexão especular, `specular`[i] o coeficiente de reflexão, `transparency`[i], o coeficiente de transmissão ou refração da luz, `IOR`[i], o

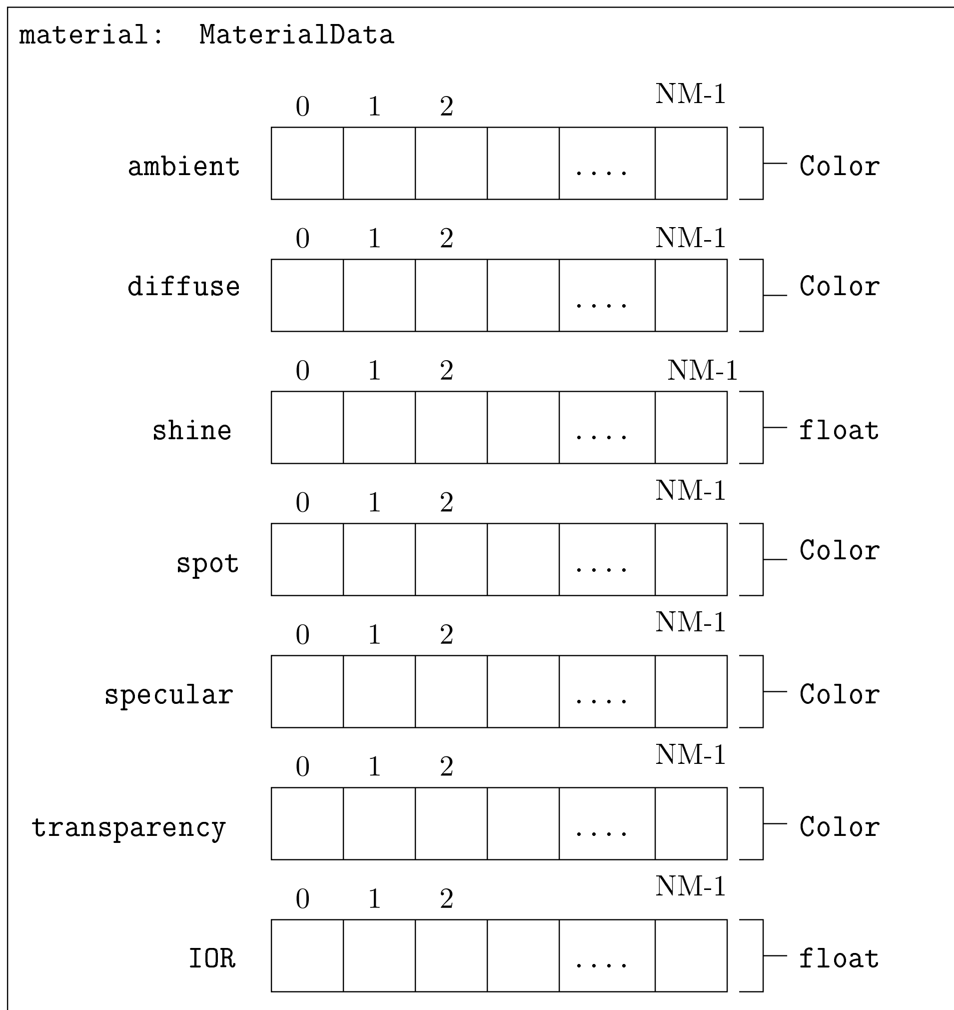


Figura 3.6: Estrutura de dados de materiais em GPU.

índice de refração.

Os dados da câmera são representados por um objeto da classe `CameraData`, como descrito na tabela 3.2, e também são armazenados na memória global da GPU.

Classe `CameraData`	
Atributos	
`Vec3 position`	Posição da câmera
`Vec3 viewPlaneNormal`	Vetor normal ao plano de visão
`Vec3 viewUP`	Vetor UP
`REAL distance`	Distância de projeção
`REAL height`	Altura da câmera
`uint projectionType`	Tipo de projeção (perspectiva ou paralela)

Tabela 3.2: Classe `CameraData`.

Os endereços dos vetores das estruturas `LightData` (figura 3.5), `MaterialData` (figura 3.6) e `CameraData`, bem como os endereços dos vetores que armazenam os dados de geometria (sopa de triângulos) e da BVH, são armazenados em uma estrutura mantida na memória constante da GPU. Com isto, tais dados podem ser acessados por qualquer kernel sem necessidade de passá-los como parâmetros, cada vez que um kernel é substituído.

3.4.1 Algoritmo 1

Esta versão foi a primeira versão desenvolvida, e realiza o tratamento da iluminação direta e indireta relativa a reflexão. Este algoritmo faz uso de um objeto da classe chamada `RayTracerData`, citada na tabela 3.3, para realizar a geração da estrutura de vetores e posteriormente copiá-los para GPU.

Classe `RayTracerData`	
Atributos	
`LightData lights`	Vetor que representa as luzes da cena
`MaterialData material`	Vetor de materiais da cena
`BVHData bvh`	Vetor que representa a BVH da cena
`RayData rays`	Vetor que representa os raios da cena
`IntersectData hits`	Vetor que contém os dados de interseção relativos aos raios
`int* validRays`	Vetor de inteiros que indicam quais raios de `rays` estão ativos
`int* raysToBeTraced`	Vetor de inteiros que contém os índices dos raios de `rays` que devem ser traçados
`int* raysToBeShaded`	Vetor de inteiros que contém os índices dos raios de `rays` que devem ser tonalizados

Tabela 3.3: Classe `RayTracerData`.

O traçador de raios em CPU realiza o preenchimento dos vetores contidos nos atributos `lights`, `material` e `bvh` e aloca espaço para os outros vetores que são usados pelo kernel e citaremos durante a descrição do algoritmo.

O algoritmo 1 faz uso de 5 kernels, conforme detalhamos a seguir :

Kernel 1 `makePixelRay`: **Geração dos raios de pixel**

Este kernel preenche os dados dos raios gerados, na estrutura de vetores indicada pelo atributo `rays`, da classe `RayData`, que ilustramos na figura 3.7.

O kernel usa $w \times h$ threads, onde a k-ésima thread é responsável pela geração do raio do pixel $k = i + j \times w$, $0 \leq i < w$, $0 \leq j < h$. O kernel faz uso de dois vetores de inteiros auxiliares, `validRays` e `raysToBeTraced`. Quando um raio k estiver ativo `validRays[k]` conterá o valor 1 e quando estiver inativo 0 e `raysToBeTraced` contem a lista dos índices dos raios ativos de `rays`.

Para projeção em perspectiva, os dados relativos ao k-ésimo raio são armazenados na posição k de cada um dos vetores de *rays*, onde `origin[k]` recebe a origem, `direction[k]` a direção do centro do pixel (i, j), `weight[k]` o peso do raio, que inicialmente é igual a 1, `weightingcolor[k]` a cor branca, `validRays[k]`$= 1$ para indicar que o raio está ativo, e `raysToBeTraced[k]`$= k$. Após a invocação do kernel, o número de raios a serem traçados é $n = w \times h$, e o nível de recursão é $r = 0$. Enquanto $n > 0$, isto é, enquanto houver raios a serem traçados, e r for menor que um dado nível máximo de recursão, invoque:

Kernel 2 `traceRay`: **Traçado de raios**

Este kernel traça todos os raios indexados por `RaysToBeTraced`. O kernel usa n threads, a i-ésima thread é responsável pelo traçado do raio de índice k=`RaysToBeTraced[i]`, $0 \leq i < n$.

Se o raio de índice k interceptar a cena, os dados da intersecção com o triângulo mais próximo à origem do raio são armazenados na posição k dos vetores contidos em `hits`. O atributo `hits` é um objeto da classe `IntersectaData`, conforme ilustrado na figura 3.8.

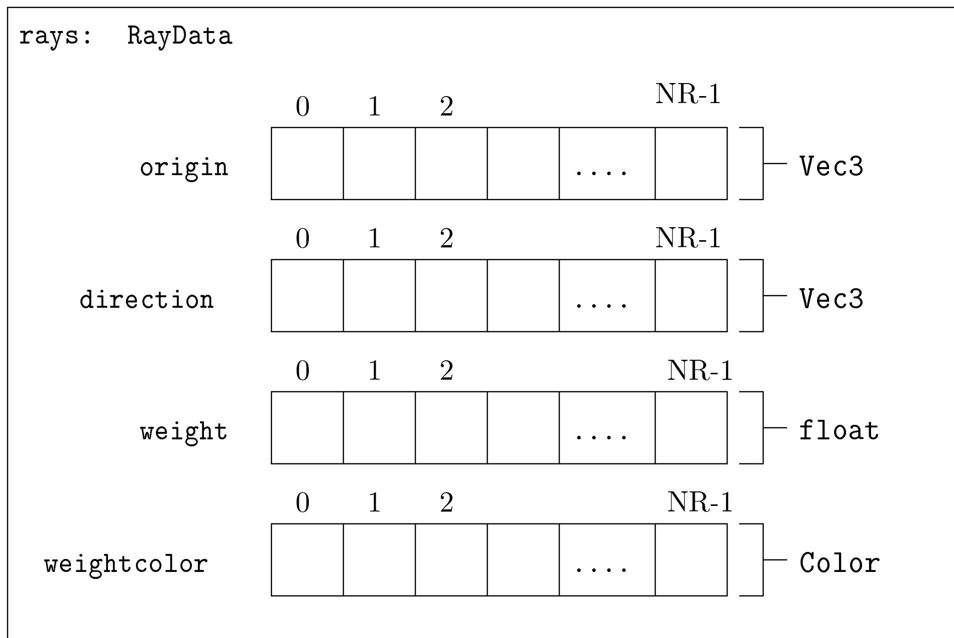


Figura 3.7: Estrutura de dados de raios em GPU.

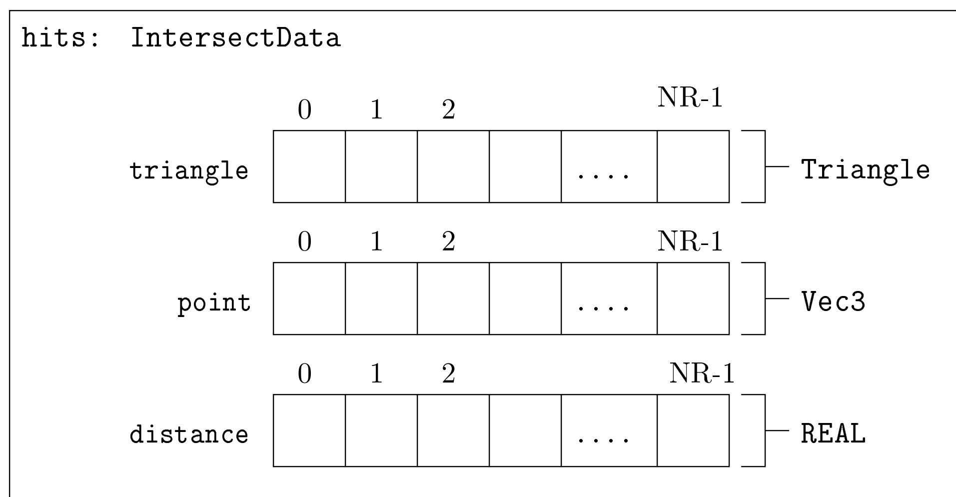


Figura 3.8: Estrutura de dados de interseção.

Em `hits`, para um raio de índice k, `triangle[k]` indica o triângulo interceptado pelo raio, `point[k]` o ponto de interseção e `distance[k]` a distância de interseção do raio com o triângulo mais próximo. O raio é ativo, ou seja, `validRays[i]`$= 1$, o que indica neste caso, que o raio deverá ser tonalizado. Se o raio não interceptar a cena, a cor de fundo da cena, ponderada pelo coeficiente do raio, é acumulada à cor do pixel em `frame[k]`, que é um vetor de cores, e o raio torna-se inativo, ou seja, `validRays[i]`$= 0$.

Kernel 3 `compactVectorT`: **Compactação do vetor de índices dos raios a serem traçados**

Este kernel efetua uma compactação, em paralelo, de todos os elementos do vetor `raysToBeTraced`, que são índices de raios ativos, ou seja, para os quais `validRays[i]`$> 0, 0 \leq i < s$, sendo tais índices armazenados no vetor de índices de raios a serem tonalizados `raysToBeShaded`. O número s de raios a serem tonalizados é dado pela soma prefixa inclusiva do vetor `validRays`, conforme exemplificamos na figura 3.9.

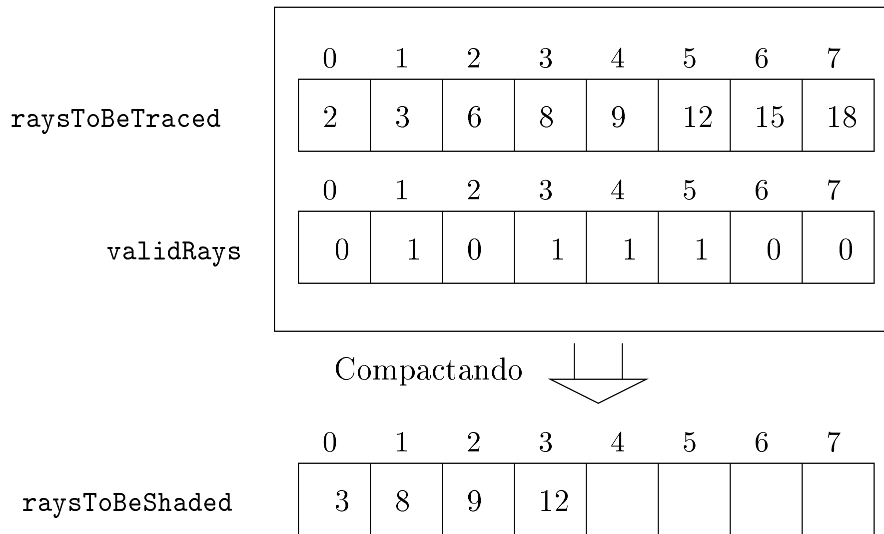


Figura 3.9: Exemplo de compactação de raios a serem traçados.

Após a invocação do kernel, o número de raios a serem traçados é $n = 0$, pois todos foram traçados. Se $s > 0$, ou seja, se houver raios a serem tonalizados, faça:

Kernel 4 `shadeRay` **Tonalização de raios**

Este kernel determina a cor devido à iluminação direta dos raios a serem tonalizados, bem como, dependendo do material, gera raios secundários devido à reflexão especular. O kernel usa s threads, a i-ésima thread é responsável pela tonalização do raio k=`raysToBeShaded[i]`, $0 \leq i < s$. O endereço do triângulo interceptado pelo raio de índice k, bem como o índice do material do triângulo e a distância da origem do raio ao ponto P de intersecção do raio com o triângulo estão armazenados na posição k dos vetores de hits. Com isso, calcula-se o ponto P, e a partir deste, traça-se um raio de sombra, na direção de cada uma das fontes de luz da cena, que conforme citado anteriormente, na figura 3.5, encontram-se armazenadas no conjunto de vetores indicado por `lights`. Se um raio de sombra interceptar algum outro objeto da cena, antes que o mesmo atinja uma fonte de luz, então sua cor é preta, pois a luz não contribui diretamente à tonalização do ponto P; caso contrário, a cor devido à contribuição da luz direta incidente em P é determinada pelo produto do coeficiente do raio de índice k, representado por `weightingcolor[k]`, pela cor da luz e acumulada à cor do pixel em `frame[k]`, conforme modelo de iluminação de Phong.

Se o material em P for especular e o peso do raio de índice k, indicado por `weight[k]` for maior que um dado peso mínimo, um novo raio secundário de reflexão deverá ser traçado, sendo seus dados armazenados na posição k dos vetores de `rays`. A origem do raio passa a ser o ponto P, a direção é a direção de reflexão, e o peso é igual ao produto do peso `weight[k]` com o componente de maior módulo do coeficiente de reflexão especular do material em P, o qual passa a ser o coeficiente do raio, e o raio é ativo, ou seja, `validRays[i]`$= 1$. Se o material em P não for especular, ou peso `weight[k]` não for maior que o peso mínimo, então um novo raio secundário não é gerado e o raio de índice k torna-se inativo, ou seja, `validRays[i]`$= 0$.

Kernel 5 `compactVectorS`: **Compactação do vetor de índices dos raios a serem tonalizados**

Este kernel efetua uma compactação, em paralelo, de todos os elementos do vetor `raysToBeShaded` que são índices de raios ativos, ou seja, para os quais `validRays[i]`> 0,

$0 \leq i < s$, sendo tais índices armazenados no vetor de índices de raios a serem traçados `raysToBeTraced`. O número n de raios a serem traçados é dado pela soma prefixa inclusiva do vetor `validRays`. Após a invocação do kernel o número de raios a serem tonalizados é $s = 0$.

Neste algoritmo, cada thread é responsável pelo traçado e tonalização de somente um raio. Como consequência, não há quaisquer conflitos de leitura ou escrita no conjunto de vetores indicados por `rays`, no vetor de raios a serem traçados, `RaysToBeTraced`, no vetor `RaysToBeToned`, nem no vetor `frame` que irá conter a imagem gerada, pois estes são armazenados na memória global da GPU e cada thread acessa elementos distintos destes vetores. Nesta versão temos apenas conflito de acesso de leitura da BVH, também armazenada em memória global, que é acessada por todas as threads, a fim de se detectar a interseção de cada raio com a cena.

Apesar do overhead inerente à carga de cada um dos kernels no dispositivo, o uso de várias threads, com a compactação intermediária dos vetores `raysToBeTraced` e `raysToBeShaded`, favorece o balanceamento de carga das threads, uma vez que conforme as threads terminam seu trabalho, estas são marcadas como inativas, saem do conjunto de raios a serem tratados ou tonalizados.

3.4.2 Algoritmo 2

Versão 2: Megakernel (com reflexão e refração)

A versão anterior trata dos raios primários e dos raios secundários gerados por reflexão especular, sem tratar os raios relativos a transparência que tratamos a partir deste algoritmo.

Dependendo das propriedades do material de um objeto, este pode ter características reflexivas e de transparência simultaneamente. Com este fato, dado que um raio r intercepte um objeto com tais características, no ponto de interseção P, temos a geração de dois novos raios a serem traçados, um para reflexão e outro para transparência, sendo que ambos devem contribuir para o cor do mesmo pixel. Nesta situação, se traçarmos ambos os raios em paralelo, haverá conflito de escrita na cor do pixel. Para contornar tal situação, esta versão faz uso de apenas um kernel, que chamamos de MegaKernel, onde uma thread realiza todas as etapas do traçado de raios, ou seja: geração dos raios de pixel; interseção do raio com a cena; determinação da iluminação direta; e geração e tratamento dos raios secundários de reflexão e refração que referem-se a um único pixel.

O algoritmo faz uso das estruturas de dados `rays` para armazenamento dos dados de cada raio, `hits` para armazenamento dos dados de interseção, `frame` para armazenamento da cor de cada pixel da imagem, todas já citadas no ínicio desta seção.

Para uma imagem de tamanho $w \times h$ este kernel usa $w.h$ threads, onde a k-ésima thread é responsável pelo tratamento do pixel k, $1 \leq k \leq w \times h$, da imagem.

Além das estruturas de dados que citamos, este algoritmo introduz o uso de uma pilha, representada por um objeto da classe `CUDARayStack`, conforme ilustrado na tabela 3.4, onde os raios pendentes de tratamento são empilhados para tratamento posterior.

Classe `CUDARayStack`	
Atributos	
`uint8 data[30 * sizeof(CUDARay)]`	Pilha de raios
`int top`	Topo da pilha
Métodos	
`void push(CUDARay& ray)`	Empilha um raio
`CUDARay pop()`	Desempilha um raio

Tabela 3.4: Classe `CUDARayStack`.

Sempre que um raio atingir um objeto com propriedades reflexivas e de transparência, o que exige o traçado de dois novos raios, um dos raios é empilhado para que seja tratado posteriormente, enquanto o outro raio é traçado até as condições de parada do algoritmo. Desenhamos a pilha de raios com capacidade para 30 raios.

O MegaKernel executa os seguintes passos:

Passo 1 **Geração dos raios de pixel**

O kernel usa $w \times h$ threads, onde a k-ésima thread é responsável pela geração, traçado e tonalização de todos os raios que contribuem para cor do pixel k, $k = i + j \times w$, $0 \leq i < w$, $0 \leq j < h$.

Passo 2 Se o raio de índice k, armazenado nas respectivas posições dos vetores de `rays`, não interceptar a cena, então a cor de fundo, ponderada pelo coeficiente do raio, é acumulada à cor do pixel em `frame[k]` e o algoritmo termina.

Passo 3 Caso ocorra interseção com a cena, um raio de sombra é disparado para cada fonte de luz da cena. Para cada raio de sombra disparado, se este interceptar algum outro objeto da cena, antes de encontrar a luz, então sua cor é preta, ou seja, a luz não contribui diretamente na tonalização do ponto P; caso contrário, a cor devido à contribuição da luz direta incidente em P é determinada pelo produto do coeficiente do raio de índice k, representado por `weightingcolor[k]`, pela cor da luz e acumulada à cor do pixel em `frame[k]`, conforme o modelo de iluminação de Phong. Se o material possuir propriedades reflexivas e/ou de transparência, no ponto P de interseção, o nível de recursão for menor que o nível máximo de recursão estabelecido e o raio possuir um peso mínimo, então um raio de reflexão e/ou refração são gerados e inseridos na pilha de raios a serem traçados.

Passo 4 Se houver raio na pilha, então um raio é retirado da pilha e armazenado na posição k dos vetores de `rays` e o passo 2 é executado, caso contrário, o algoritmo termina.

Nesta versão do algoritmo, temos apenas um kernel e cada thread é responsável pelo traçado e tonalização de todos os raios relativos à um único pixel, o que implica na ausência de conflito de leitura ou escrita nos vetores de raios ou no vetor que representa a imagem. Outro aspecto positivo é que, como não há troca de kernel, o algoritmo pode apresentar performance superior ao algoritmo 1, dependendo das características da cena em tratamento. De outro modo, se imaginarmos uma cena onde tenhamos um grande número de raios e uma grande parte deles terminem no primeiro nível de recursão e outra parte prossiga até o último nível de recursão, grande parte das threads ficarão ociosas até que outras finalizem o traçado. Acreditamos que neste tipo de cena, há grandes possibilidades de que o Algoritmo 1 obtenha uma melhor aceleração, mas lembramos que o Algoritmo 1 é incapaz de tratar raios de transparência, desta forma podemos considerar que este algoritmo é mais completo que o primeiro.

3.4.3 Algoritmo 3

Versão 3: Vários kernels (com reflexão e refração)

Nesta versão procuramos desenvolver um algoritmo que pudesse realizar o traçado de raios de reflexão e transparência, mas que pudesse reduzir a possibilidade de desbalanceamento de carga nas situações citadas.

Esta versão adota de maneira limitada, a idéia da compactação intermediária de vetores, inicialmente presente na primeira versão e, para contornar a escrita concorrente de mais de um raio sobre um mesmo pixel, adota a idéia do uso de uma pilha de raios, presente na segunda versão do algoritmo. Partindo da observação de que em uma cena podemos ter vários raios que finalizam seu processamento no primeiro estágio, ou seja, no traçado do raio de pixel, por conta do raio nem mesmo interceptar a cena, decidimos nesta versão realizar uma compactação intermediária logo após o tratamento dos raios primários e reduzir assim o desbalanceamento.

Este algoritmo faz uso das estruturas de dados já introduzidas, `rays`, `hits`, `validRays`, `raysToBeTraced`, `raysToBeShaded`, citadas no algoritmo 1 e da pilha de raios citadas no algoritmo 2, `CUDARayStack` e realiza o traçado e tonalização dos raios fazendo uso de 3 kernels, conforme explicado a seguir:

Kernel 1 `tracePixelRay`: **Traça raios de pixel**

Neste kernel, a k-ésima thread é responsável pela geração, traçado e tonalização do pixel k, $k = i+j \times w$, $0 \leq i < w$, $0 \leq j < h$. Cada thread k gera e traça um raio em direção a cena. Caso o raio intercepte algum objeto da cena, `validRays[k]`$= 1$ e `raysToBeTraced[k]`$= k$. Caso contrário, a cor de fundo é ponderada pelo coeficiente do raio e acumulada em `frame[k]` e `validRays[k]`$= 0$.

Kernel 2 `compactVectorT`: **Compactação do vetor**

Este kernel efetua uma compactação, em paralelo, dos elementos do vetor `raysToBeTraced`, que são índices de raios ativos, ou seja, para os quais `validRays[k]`> 0 e armazena os índices dos raios ativos no vetor `RaysToBeToned`. O número s de raios a serem tonalizados é dado pela soma prefixa inclusiva do vetor *validRays*. Após a invocação do kernel, o número de raios a serem traçados é $n = 0$.

Kernel 3 `shadeRay`: **Tonalização/traçado**

Este kernel determina a cor devida à iluminação direta e indireta (raios secundários). O kernel usa s threads, a i-ésima thread é responsável pela tonalização do raio $k = S[i]$, $0 \leq i < s$. Cada thread dispara então um raio de sombra para cada fonte de luz da cena.

Passo 1 Para cada raio de sombra disparado, se este interceptar algum outro objeto da cena, antes de encontrar a luz, então sua cor é preta, ou seja, a luz não contribui diretamente na tonalização do ponto P; caso contrário, a cor devido à contribuição da luz direta incidente em P é determinada pelo produto do coeficiente do raio de índice k, representado por `weightingcolor[k]`, pela cor da luz e acumulada com a cor do pixel em `frame[k]`, conforme modelo de iluminação de Phong.

Passo 2 Se o material possuir propriedades reflexivas e/ou de transparência, no ponto P de interseção, o nível de recursão for menor que o nível máximo de recursão estabelecido e o raio possuir um peso mínimo, então um raio de reflexão e/ou refração são gerados e inseridos na pilha de raios a serem traçados.

Passo 3 Se houver raio na pilha, então um raio é retirado da pilha e armazenado na posição k dos vetores de `rays` e o passo 2 é executado, caso contrário, o algoritmo termina.

Assim como no algoritmo 1, não há conflito de leitura ou escrita no vetor de raios, interseção, raios válidos, raios a serem traçados ou tonalizados e no vetor da imagem. Este algoritmo trata da possibilidade de desbalanceamento dos raios primários em primeiro nível, realizando a compactação do vetor de raios, que elimina os raios que partem da câmera e não atingem a cena. Entendemos que,

para cenas onde grande parte dos raios primários atinjam o fundo da cena, ao invés de atingirem objetos da cena, este algoritmo pode ser superior ao algoritmo 2, pois reduz o desbalanceamento. Dependendo da cena, este algoritmo também pode ser superior ao algoritmo 1, pois o overhead da carga dos kernels é inferior pois o número de kernels é menor.

3.5 Geração da BVH em CUDA

O algoritmo de construção da BVH de uma cena toma como entrada um vetor T contendo os nt triângulos de todos os atores da cena e um vetor B, com espaço para os nb nós da BVH. Cada i-ésimo elemento de B é um objeto com as seguintes propriedades: dois pontos p_1 e p_2 que definem o AABB do nó i, e dois índices b_l e b_r, $0 \leq b_l < b_r < nt$, que identificam em T, o primeiro e o último triângulo do nó, respectivamente, caso $B[i]$ seja um nó folha, ou identificam em B os nós filhos à esquerda e à direita do nó $B[i]$, respectivamente.

O primeiro passo executado é a criação em CPU do nó raiz da BVH (contendo todos os nt triângulos de T) e sua adição ao vetor B. Após esse passo, a BVH contém um único nível, e $NB = 1$, pois há apenas um nó na BVH, que precisa (ou não) ser subdividido. Em seguida, invoca-se um kernel para subdivisão dos nós de B (e, com isso, geração de um novo nível da BVH) que executa em NB blocos de 96 threads cada, sendo cada bloco i, $0 \leq i < nb$, responsável pela subdivisão do j-ésimo nó, como definido a seguir. Um bloco i pode gerar nenhum (caso o nó j seja classificado como nó folha) ou dois novos nós (filhos do nó j). O kernel toma como parâmetros de entrada o vetor B (com os nós da BVH até um nível d) e um vetor I contendo os índices dos nós de B, do nível d, que devem ser avaliados e subdivididos se for o caso. Os parâmetros de saída do kernel são um vetor O (com espaço para $2nb$ elementos) com os índices dos nós filhos gerados pelo kernel e um vetor de inteiros A (com espaço para $2nb$ elementos). Se um bloco i não gerar um novo nó da BVH (porque o nó $j = I[i]$ é classificado como nó folha), então $A[2i] = A[2i+1] = 0$; senão, $A[2i] = A[2i+1] = 1$ e $O[2i] = cl$ e $O[2i+1] = cr$, onde $cl = 2j+1$ e $cr = 2j+2$ são os nós filhos à esquerda e à direita resultantes da subdivisão do nó j, respectivamente. Após a invocação do kernel de subdivisão, o vetor O ou está vazio ou contém (eventualmente em posições não consecutivas) os pares de índices de nós filhos que formam o nível $d+1$ da BVH, os quais precisam ou não ser subdivididos. A seguir, invoca-se uma função para compactação do vetor O com base no vetor de índices A. Os elementos resultantes da compactação de O são armazenados em I e o kernel de subdivisão é invocado novamente. Estes passos são repetidos até que o vetor I seja vazio. O kernel de subdivisão executa em blocos de 96 threads, 32 threads (um warp) para cada eixo cartesiano. Cada thread de um warp é responsável pela avaliação, em paralelo, da função SAH em cada um dos $k = 32$ planos de corte considerados, em cada direção, usando para isto primitivos paralelos como soma prefixa e redução. No final do kernel, se o nó correspondente ao bloco necessitar ser subdividido, apenas a thread correspondente ao plano de corte de custo mínimo escreve nos vetores de saída O e A.

3.6 Comentários Finais

Neste capítulo apresentamos a possibilidade de uso da GPU, como dispositivo paralelo genérico, para aceleração do traçado de raios, tendo em vista a grande capacidade computacional destes dispositivos. Primeiramente realizamos uma revisão da literatura, apresentando os principais trabalhos relacionados a implementação de traçado de raios em GPU, principalmente aqueles que tratam do percurso e construção de estruturas de aceleração, que são intensamente utilizados em traçado de raios para determinar a interseção dos raios com objetos da cena, e como já citado no capítulo 2, constituem o principal gargalo deste tipo de aplicação. Visualizamos nesta revisão que as principais

estruturas de aceleração adotadas para o traçado de raios em GPU também são BVH e kd-tree e que estas possuem performance similar, sendo a BVH considerada a estrutura de aceleração mais adequada para GPU, principalmente quando necessário o tratamento de cenas dinâmicas.

Posteriormente, apresentamos uma introdução sobre a arquitetura CUDA, onde citamos algumas das principais características do modelo de execução e do modelo de memória adotado pela GPU. Com explicado, damos o nome de kernel, ao código que é executado no dispositivo. Um kernel é executado em grids de blocos de threads numa arquitetura SIMD. O modelo de memória da GPU foi detalhado, onde pudemos observar que a memória do dispositivo é composta por: a) Memória de textura; b) Memória constante; c) Memória global; d) Memória compartilhada; e) Memória local; e memória relativa ao registrador, sendo estas últimas pertinentes a cada bloco e as outras com possibilidade de acesso por qualquer thread. Citamos também alguns aspectos ligados a latência de cada um destes tipos de memória. Apesar da possibilidade de acesso de qualquer thread a qualquer região da memória, por exemplo à memória global, citamos a importância de que o acesso ocorra de maneira coalescida, ou seja, que sejam acessadas sempre em endereços multiplos de 4 ou 16 bytes, por threads de um mesmo warp, e que não haja acesso concorrente. O acesso coalescido é um dos principais fatores para obtenção de performance, pois evitamos que a GPU serialize o acesso a memória.

A interface de programação fornece extensões à linguagem C para que seja possível programar o dispositivo. Dentre estas extensões, citamos os qualificadores de tipo de função, qualificadores de tipos de variáveis e especificação de configuração de execução.

Na seção 3.4 apresentamos as principais estruturas de dados construídas para a execução do traçado de raios em GPU que são comuns aos três algoritmos implementados. Apresentamos os três algoritmos implementados, discutindo as principais características, vantagens e desvantagens de cada um em relação ao outro e as estruturas de dados particulares de cada versão. Ao final, apresentamos a descrição do algoritmo para construção da BVH em GPU, que visa proporcionar ao algoritmo aceleração para o tratamento de cenas dinâmicas. Quando um ator da cena é dinâmico, ou seja, quando este ator se move na cena, ou sofre deformação, a malha de triângulos que o representa sofre mudanças, o que consequentemente torna a BVH inválida. Quando isto ocorre, pode haver necessidade da reconstrução da BVH, o que realizamos em GPU.

Capítulo 4

Resultados

4.1 Introdução

Neste capítulo apresentamos os resultados alcançados pelos algoritmos de traçado de raios implementados. Na seção 4.2 relacionamos o conjunto de cenas geradas pelo traçador de raios construído, onde relacionamos as principais características que são relevantes à análise de performance, como número de triângulos e número de luzes. Na seção 4.3 analisamos, sob diversos aspectos, a performance do traçador de raios, quando utilizado para o *ray casting*, e na seção 4.4, analisamos os resultados de performance para o *ray tracing*. Na seção 4.5 apresentamos os resultados obtidos pelo algoritmo de construção da BVH em CPU e GPU, e na seção 4.6 apresentamos as imagens geradas pelo traçador de raios.

Os testes foram realizados em um computador Dell XPS720, processador Intel Core 2 Duo 6300 1.6GHz com 2046MB de memória RAM e uma placa de vídeo NVIDIA GeForce 8800 GTX com 768MB de memória, com sistema Windows XP Professional SP2.

4.2 Descrição das Cenas

Como já descrevemos anteriormente, as cenas são descritas através de uma gramática de cena, disponível no apêndice A, onde o usuário pode especificar os diversos primitivos da cena, como luzes, câmera, configurações do ambiente, resolução da imagem a ser produzida e níveis de recursão a ser usado pelo traçador de raios. Na tabela 4.1 descrevemos o conjunto das cenas utilizadas, e para cada uma descrevemos o número de luzes, número de triângulos da malha de triângulo produzida e quantidade de nós necessários para representar a cena na BVH.

As configurações de 1 a 6 da tabela contém uma cena representada pelo mesmo conjunto de esferas coloridas e diferenciam-se apenas pelo número de luzes. As cenas de 6 a 10 são compostas por um coelho entre planos perpendiculares e espelhos e também se diferenciam apenas pelo número de luzes. As cenas de 11 a 14 possuem apenas uma luz na cena e diferenciam-se pelo número de triângulos da cena, que é variado acrescentando-se mais coelhos a cena. As figuras 15 a 17 são utilizadas como exemplos complementares.

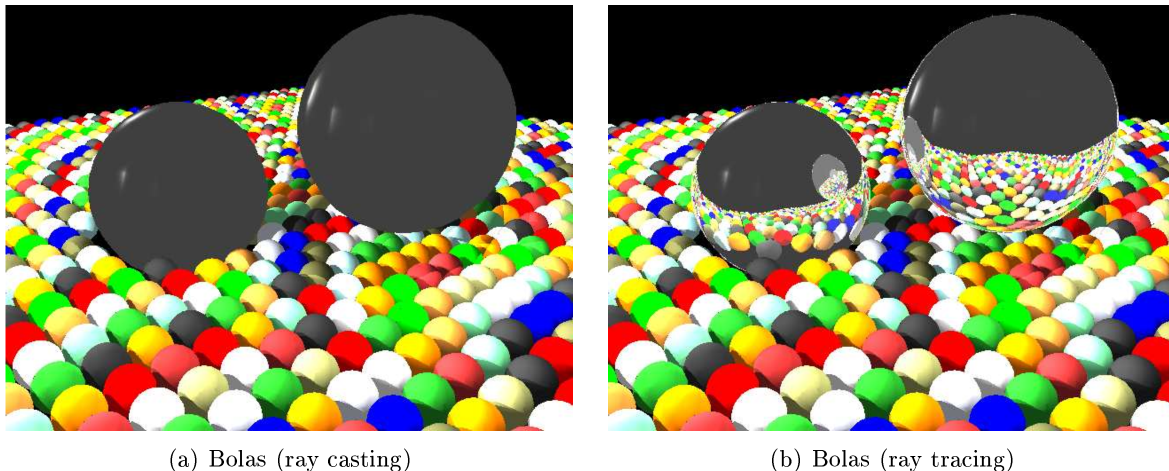
4.3 Traçado de Raios Primários

Esta seção é dedicada a análise de performance dos quatro algoritmos de traçado de raios, para o processo de *ray casting*, nome dado ao processo de traçado de raios primários. Entendemos por

Tabela 4.1: Relação de cenas.

#	Cena	Luzes	Triângulos	Nós
1	Bolas1	1	708.108	390.865
2	Bolas2	2	708.108	390.865
3	Bolas4	4	708.108	390.865
4	Bolas8	8	708.108	390.865
5	Bolas16	16	708.108	390.865
6	Coelhos0101	1	70.212	23.842
7	Coelhos0102	2	70.212	23.842
8	Coelhos0104	4	70.212	23.842
9	Coelhos0108	8	70.212	23.842
10	Coelhos0116	16	70.212	23.842
11	Coelhos0201	1	139.876	47.510
12	Coelhos0401	1	279.204	94.846
13	Coelhos0801	1	557.860	189.394
14	Coelhos1601	1	1.115.172	378.864
15	Coelhos1616	16	1.115.172	378.864
16	Golfinho	5	8.018	2.622
17	Xadrez	2	234.962	82.166
18	Anel	1	2.084	728
19	Conference	1	282.759	101.184

raios primários os raios que partem da câmera em direção a cena e que, ao interceptarem um objeto, seguem até as luzes da cena para identificar se o objeto está visível ou não, mas as propriedades de reflexão ou transparência dos objetos da cena não são consideradas. Apenas para efeito de comparação, ilustramos na figura 4.1, a diferença entre imagens geradas por *ray casting* e *ray tracing*.



(a) Bolas (ray casting) (b) Bolas (ray tracing)

Figura 4.1: Traçado de raios primários × traçado de raios.

Realizamos a geração das imagens das cenas de 1 a 17, da tabela 4.1, utilizando o algoritmo para CPU e os três algoritmos para GPU, com uma resolução de 640×480 pixels, e apresentamos os tempos obtidos por cada um dos algoritmos na tabela 4.2. Observamos na tabela 4.2 que nenhum dos algoritmos pode ser considerado o melhor para todos os casos. Observamos que o algoritmo 2

Tabela 4.2: Traçado de raios primários: performance CPU × GPU (resolução 640 × 480).

#	CPU (ms)	Alg_1 (ms)	Alg_2 (ms)	Alg_3 (ms)
1	66.482	933	897	977
2	80.253	1.101	1.045	1.148
3	108.385	1.359	1.312	1.410
4	167.026	1.976	1.916	2.030
5	283.013	3.186	3.099	3.250
6	23.420	560	567	549
7	34.873	751	745	742
8	54.631	1.131	1.143	1.118
9	98.715	1.919	1.946	1.911
10	184.785	3.397	3.389	3.398
11	24.152	549	535	516
12	24.640	640	621	598
13	29.225	672	639	636
14	28.561	665	640	638
15	228.049	3.926	4.029	3.923
16	53.897	895	842	886
17	14.137	558	589	531

foi aquele que obteve o melhor tempo para as cenas de 1 a 5, relativa as bolas, mas este fato não se repetiu para todas as cenas. Analisaremos a seguir os dados relativos a aceleração e eficiência. Medimos a aceleração comparando-se o tempo obtido por cada um dos algoritmos em GPU, em relação ao tempo obtido pelo algoritmo para CPU. O cálculo da aceleração é fornecida pela equação 4.1, onde o tempo obtido em CPU, representado por T_C, é dividido pelo tempo do algoritmo para GPU, representado por T_G.

$$SP = \frac{T_C}{T_G} \tag{4.1}$$

A eficiência alcançada pelos algoritmos é obtida dividindo-se a aceleração produzida, pela quantidade de processadores do dispositivo, em nosso modelo 128 processadores. O cálculo da eficiência é dado pela equação 4.2.

$$EF = \frac{SP}{NP} \tag{4.2}$$

Os valores relativos a aceleração e eficiência alcançados pelos algoritmos são apresentados na tabela 4.3. As colunas SP_n e EF_n representam, respectivamente, os valores de aceleração e eficiência obtidos pelo algoritmo n. Observamos nesta tabela que a maior aceleração foi de 91, obtida pelo algoritmo 2 na cena 5, responsável por uma eficiência de 0,71, porém observamos também que o pior resultado foi produzido pelo mesmo algoritmo, na cena 17, o que confirma a afirmação de que não conseguimos eleger um algoritmo que seja o melhor. Observando as cenas de 1 a 5, onde a complexidade do traçado de raios é crescente devido ao acréscimo no número de luzes na cena, percebemos que a aceleração obtida também é crescente para todos os algoritmos, o que nos leva a concluir que quanto maior a complexidade da cena, maior aproveitamento obtemos da capacidade aritimética da GPU para suplantar o overhead da carga dos kernels. Na figura 4.2 apresentamos o gráfico da aceleração obtida por cada um dos três algoritmos, para as cenas de 1 a 5, em relação ao algoritmo para CPU, para o *ray casting*, quando aumentamos a quantidade de luzes em uma cena. Neste gráfico consideramos as cenas de 1 a 5.

Analisamos a seguir, o comportamento da aceleração quando aumentamos o número de triângulos. Selecionamos as cenas 6 e 11 a 14, onde o número de triângulos é crescente para ilustrarmos a

Tabela 4.3: Traçado de raios primários: aceleração e eficiência.

#	SP_1	EF_1	SP_2	EF_2	SP_3	EF_3
1	71	0,55	74	0,57	67	0,53
2	72	0,56	76	0,59	69	0,54
3	79	0,62	82	0,64	76	0,60
4	84	0,66	87	0,68	82	0,64
5	88	0,69	91	0,71	87	0,68
6	41	0,32	41	0,32	42	0,33
7	46	0,36	46	0,36	46	0,36
8	48	0,37	47	0,37	48	0,38
9	51	0,40	50	0,39	51	0,40
10	54	0,42	54	0,42	54	0,42
11	43	0,34	45	0,35	46	0,36
12	38	0,30	39	0,30	41	0,32
13	43	0,33	45	0,35	45	0,35
14	42	0,33	44	0,34	44	0,34
15	58	0,45	56	0,44	58	0,45
16	60	0,47	63	0,49	60	0,47
17	25	0,19	23	0,18	26	0,20

aceleração no gráfico 4.3. Nas cenas utilizadas no gráfico 4.3, o algoritmo 3 se mostra sempre como o algoritmo que obtém a melhor performance. Também observamos que o aumento de triângulos, nem sempre implicou em aumento da aceleração.

Analisamos a seguir, a relação entre o aumento da resolução da imagem gerada e a aceleração produzida. Quando aumentamos a resolução, um efeito direto é o aumento do número de raios na mesma proporção, o que implica em maior trabalho para o traçado de raios. Na tabela 4.4, demonstramos o comportamento da aceleração dos algoritmos quando, sob uma mesma cena, neste caso a cena 3, aumentamos a resolução. O gráfico relativo a tabela 4.4 é apresentado na figura

Tabela 4.4: Traçado de raios primários: aceleração $\times$ resolução das imagens (cena 3).

Resolução	CPU	Alg_1	SP_1	Alg_2	SP_2	Alg_3	SP_3
300 x 300	31.282	438	71	421	74	452	69
400 x 400	55.748	728	76	689	80	755	73
800 x 800	222.039	2618	84	2.463	90	2.722	81
1024 x 1024	364.662	4.294	84	4.100	88	4.515	80

4.4. Observamos no gráfico 4.4 que a aceleração é geralmente crescente, quando aumentamos a resolução da cena, e que para a cena 3 escolhida, o algoritmo 2, que faz uso de apenas um kernel, foi o que obteve a melhor performance, em todas as resoluções, provavelmente pelo fato de que a melhoria no balanceamento das cargas entre as threads, fornecido pelos outros algoritmos, não foi suficiente para superar o overhead da troca dos kernels.

Como citamos anteriormente, a eficiência dos algoritmos é obtida de forma direta, a partir da aceleração. A eficiência ótima, ou seja, quando o valor da eficiência atinge 1, evidentemente não pode ser alcançada, porém, quanto mais próximos estivermos deste número, melhor será o aproveitamento dos processadores pelo algoritmo da GPU. Apresentamos no gráfico da figura 4.5 a eficiência apresentada pelos três algoritmos. No gráfico podemos observar que, em geral, a eficiência

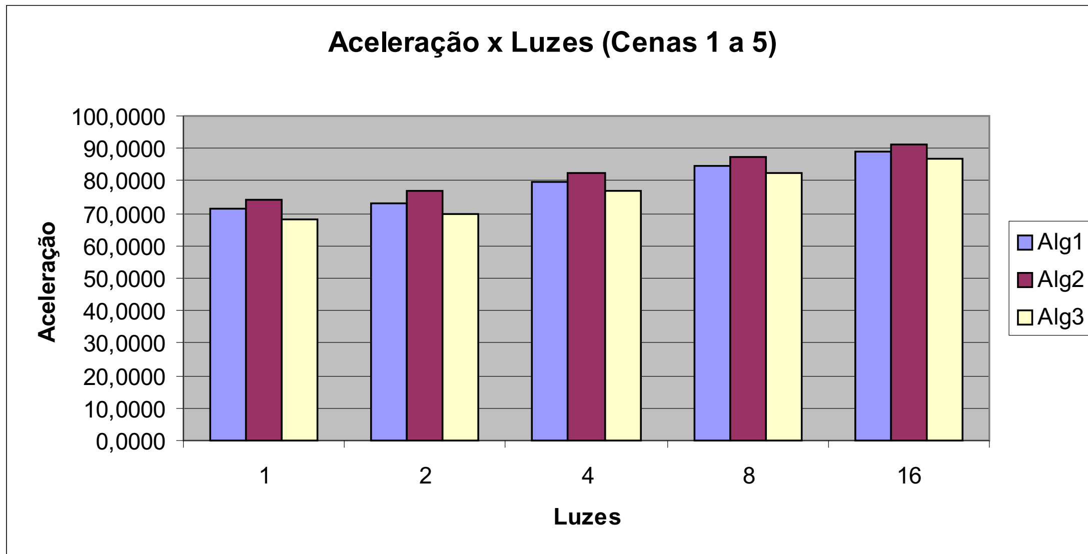


Figura 4.2: Traçado de raios primários: aceleração × número de luzes.

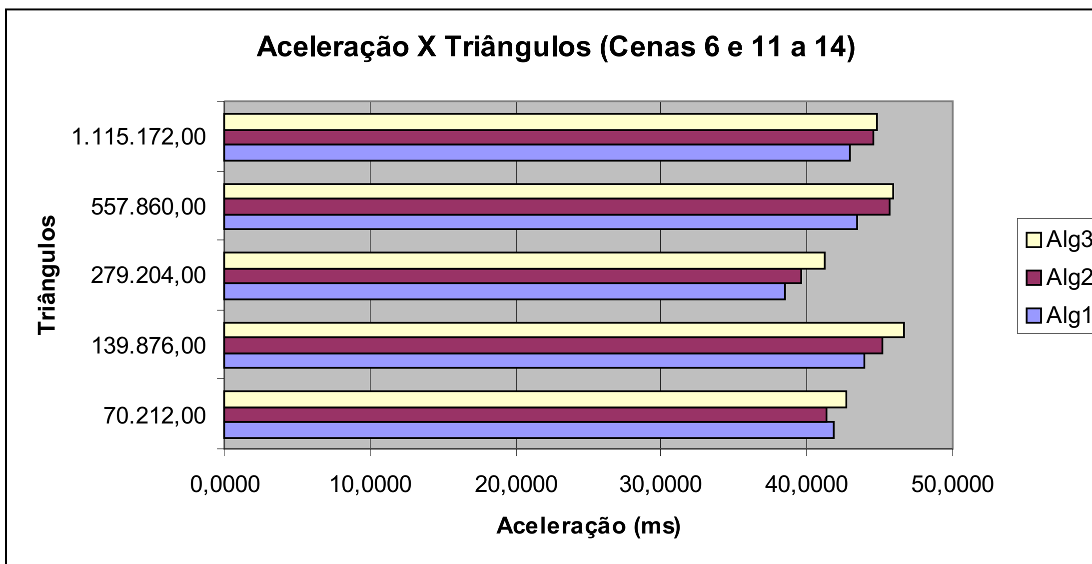


Figura 4.3: Traçado de raios primários: aceleração × número de triângulos (cenas 6 e 11 a 14).

é crescente conforme aumentamos a complexidade da cena.

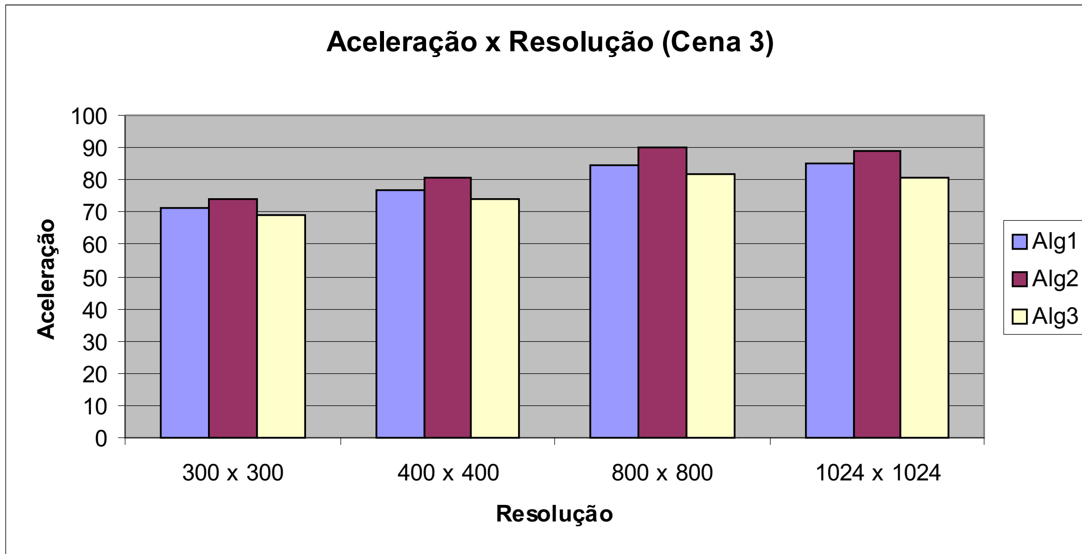


Figura 4.4: Traçado de raios primários: aceleração × resolução das imagens (cena 3).

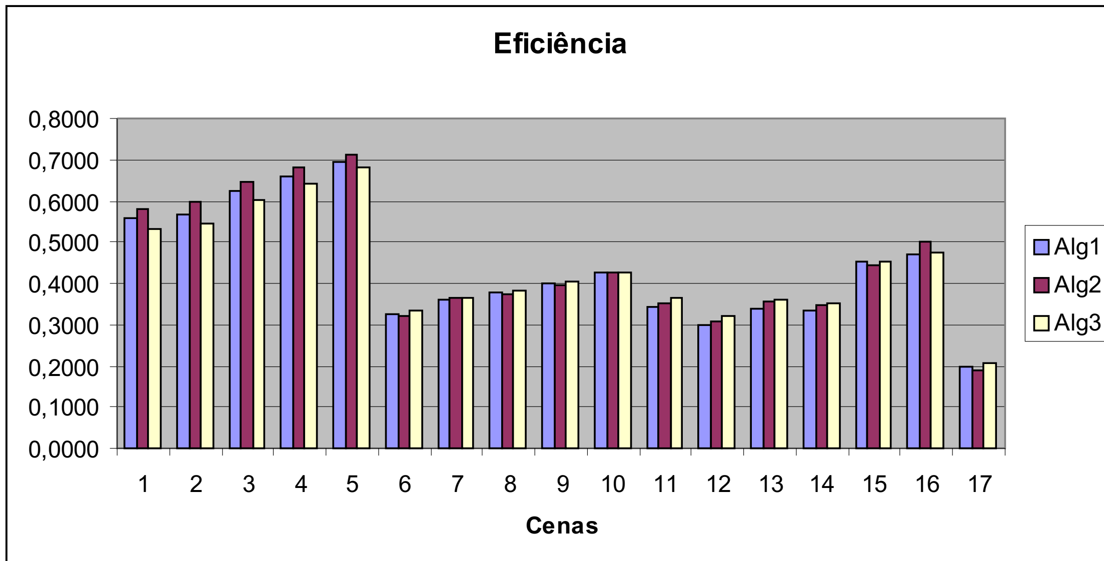


Figura 4.5: Traçado de raios primários: eficiência.

4.4 Traçado de Raios

Nesta seção analisamos os resultados obtidos pelos algoritmos implementados, para o traçado de raios primários e secundários (reflexão e refração). A introdução do tratamento dos raios secundários traz consigo, a possibilidade de que o desbalanceamento entre as threads seja aumentado, uma vez que temos raios que colidem com objetos com características reflexivas ou transparentes, e portanto geram novos raios a partir do ponto de interseção, e temos raios que interrompem seu traçado porque os objetos atingidos não possuem tais propriedades. Além deste fato, a redução gradual do peso dos raios também é responsável por fazer com que raios interrompam seu processamento em momentos distintos. De outro modo, o introdução de mais níveis de recursão, imposta de maneira obrigatória pelo traçado de raios secundários, implica em aumento da complexidade do traçado de raios, fato que pode beneficiar a aceleração obtida pelos algoritmos em GPU. Na tabela 4.5 apresentamos os números de performance de cada algoritmo para cada uma das cenas já introduzidas. Para as

Tabela 4.5: Traçado de raios: performance CPU $\times$ GPU (10 níveis)-(resolução 640 $\times$ 480).

#	CPU (ms)	Alg_1 (ms)	Alg_2 (ms)	Alg_3 (ms)
1	77.327	1.274	1.363	1.266
2	94.692	1.516	1.591	1.501
3	135.672	1.916	2.044	1.873
4	211.475	2.810	2.991	2.722
5	364.242	4.563	4.860	4.390
6	40.431	964	935	979
7	60.119	1.284	1.266	1.320
8	94.114	1.934	1.875	1.977
9	169.192	3.307	3.270	3.381
10	323.640	5.834	5.808	5.987
11	42.000	917	861	944
12	44.560	1.070	987	1.083
13	55.367	1.178	1.065	1.237
14	56.386	1.266	1.162	1.314
15	460.536	8.315	8.113	8.341
16	190.453	5.522	5.625	4.012
17	16.785	660	651	620

cenas 1 a 5, observamos que o Algoritmo 3 apresenta os melhores tempos, diferenciando-se do que é observado no *ray casting*, onde o algoritmo 2 se mostrou superior. É provável que a compactação do vetor de raios, que elimina os raios que finalizam o processamento na primeira iteração do algoritmo tenha proporcionado um balanceamento de carga mais eficiente e proporcionado o melhor resultado. Apesar disto, o algoritmo 2 continua a apresentar a melhor performance, para uma número considerável de cenas. Na tabela 4.6 apresentamos os dados relativos a aceleração e eficiência dos algoritmos de traçado de raios, para 10 níveis de recursão. Observamos que a melhor aceleração é produzida pelo algoritmo 3, na cena 5, e os piores resultados são sempre produzidos na cena 17, sendo o algoritmo 1 o pior deles para esta cena.

Analisamos a seguir, a relação entre a aceleração dos algoritmos e uma série de outros fatores. Primeiramente analisamos a relação entre o número de luzes de uma cena e a aceleração. Na figura 4.6, para as cenas de 1 a 5, apresentamos o gráfico da aceleração obtida por cada um dos três algoritmos, em relação ao algoritmo para CPU, quando aumentamos a quantidade de luzes presentes na cena. No gráfico 4.6 observamos que quanto maior o número de luzes, maior a aceleração obtida,

Tabela 4.6: Traçado de raios: aceleração e eficiência (10 níveis).

#	SP_1	EF_1	SP_2	EF_2	SP_3	EF_3
1	60	0,47	56	0,44	61	0,47
2	62	0,48	59	0,46	63	0,49
3	70	0,55	66	0,51	72	0,56
4	75	0,58	70	0,55	77	0,60
5	79	0,62	74	0,58	82	0,64
6	41	0,32	43	0,33	41	0,32
7	46	0,36	47	0,37	45	0,35
8	48	0,38	50	0,39	47	0,37
9	51	0,39	51	0,40	50	0,39
10	55	0,43	55	0,43	54	0,42
11	45	0,35	48	0,38	44	0,34
12	41	0,32	45	0,35	41	0,32
13	46	0,36	51	0,40	44	0,34
14	44	0,34	48	0,37	42	0,33
15	55	0,43	56	0,44	55	0,43
16	34	0,26	33	0,26	47	0,37
17	25	0,19	25	0,20	27	0,21

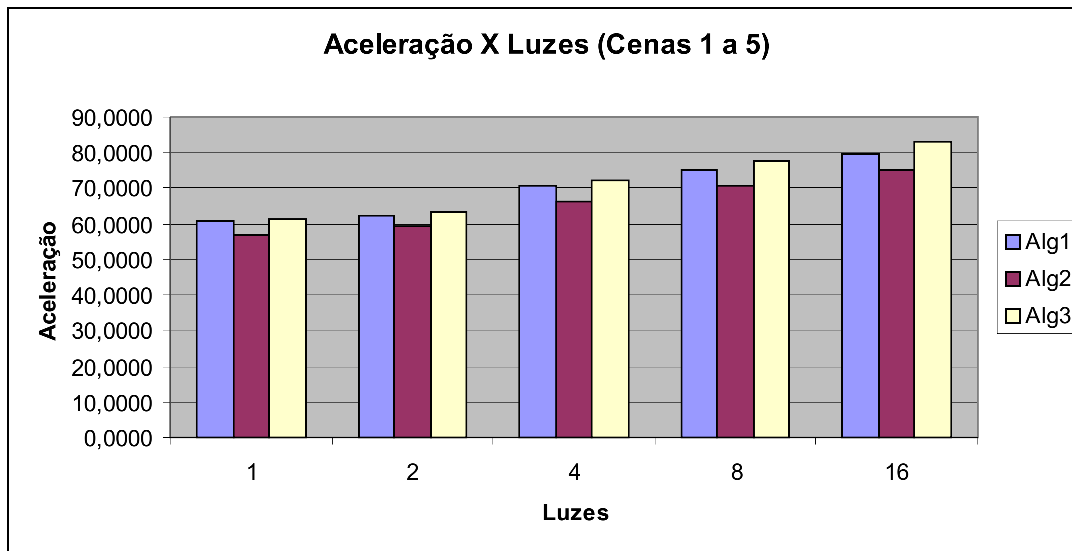


Figura 4.6: Traçado de raios: aceleração × número de luzes.

o que mais uma vez deve ser causado pelo melhor aproveitamento da GPU em cenas mais complexas. Além disto, observamos que o algoritmo 2, que faz uso de um único kernel sempre apresenta a pior aceleração, fato que atribuímos a ausência das compactações intermediárias que eliminam as threads que tratam de raios que finalizam seu processamento a cada iteração.

Passamos a análise da relação do número de triângulos das cenas e da aceleração dos algoritmos. Na figura 4.7 apresentamos o gráfico da aceleração obtida por cada um dos três algoritmos, quando aumentamos a quantidade de triângulos. Acreditamos que o algoritmo 2, que apresentou o melhor desempenho no gráfico 4.7, tenham alcançado este resultado pelo fato de que as cenas possuem apenas uma luz, e portanto, o tempo utilizado na carga de vários kernels superou o ganho de tempo obtido com o melhor balanceamento das threads. Não conseguimos obter uma relação exata sobre o

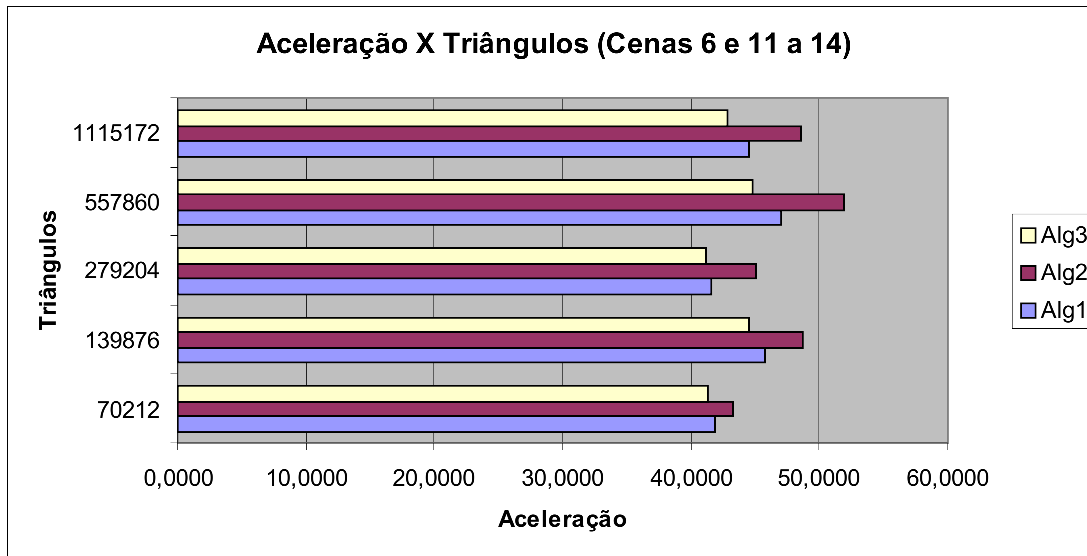


Figura 4.7: Traçado de raios: aceleração × número de triângulos (cenas 6 e 11 a 14).

aumento de triângulos da cena e a aceleração obtida, e acreditamos que neste caso especifico, talvez a inserção de alguns triângulos tenha obstruídos a interseção de raios com objetos reflexivos.

Apresentamos na tabela 4.7 o comportamento da aceleração dos algoritmos quando, sob uma mesma cena, neste caso a cena 3, aumentamos a resolução. O gráfico relativo a tabela 4.7 é

Tabela 4.7: Traçado de raios: aceleração × resolução (cena 3).

Resolução	CPU	Alg_1	SP_1	Alg_2	SP_2	Alg_3	SP_3
300 × 300	41.841	677	61	667	62	684	61
400 × 400	73.386	1.105	66	1.029	71	1.121	65
800 × 800	293.372	4.128	71	3.914	74	3.774	77,73
1024 × 1024	482.690	6.759	71	6.390	75	6.149	78

apresentado na figura 4.8. No gráfico podemos observar que, para resoluções mais baixas, o algoritmo 2 produz os melhores resultados, porém para cenas de maior resolução, o algoritmo é superado pelo de número 3.

No processo de *ray tracing*, devido as propriedades de reflexão e transparência dos objetos, os raios podem ricochetear pela cena inúmeras vezes. O número de vezes que estes raios podem realizar tal operação é controlado pelo número de níveis de recursão, ou seja, quanto mais níveis de recursão utilizamos, maior a complexidade para o traçado de raios e melhor será a qualidade da imagem produzida. A seguir analisaremos o comportamento de performance dos algoritmos quando variamos os níveis de recursão (ver tabela 4.8). O gráfico relativo a variação da aceleração mediante aumento dos níveis de recursão é demonstrado no gráfico da figura 4.9. Observamos que para poucos níveis (0 e 1), o algoritmo 2 apresenta a melhor aceleração, chegando a atingir o valor de 93, que é a melhor aceleração obtida dentre todos os algoritmos em todos os níveis apresentados. No entanto, a partir de 4 níveis de recursão, o algoritmo 3 se mostra a melhor opção para esta cena. Acreditamos que isto se deve ao fato de que, nos algoritmos 2 e 3, threads que finalizam seu traçado em algum momento, liberam o processador para que outras threads trabalhem. Para completar a análise fornecemos no gráfico 4.10 a eficiência apresentada pelos três algoritmos, em todas as cenas citadas.

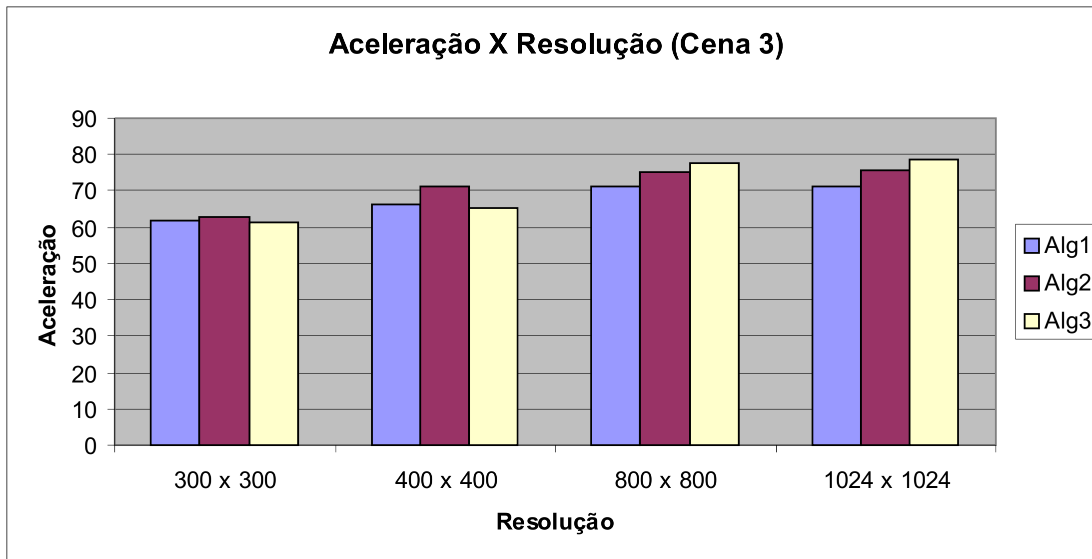


Figura 4.8: Traçado de raios: aceleração × resolução das imagens (cena 3).

Tabela 4.8: Traçado de raios: aceleração × níveis de recursão (cena 5).

Níveis	CPU	Alg_1	SP_1	Alg_2	SP_2	Alg_3	SP_3
0 nível	283.013	3.186	88	3.099	91	3.250	87
1 nível	355.619	3.949	90	3.800	93	4.044	87
2 níveis	360.304	4.272	84	4.336	83	4.163	86
4 níveis	361.065	4.502	80	4.661	77	4.283	84
8 níveis	361.162	4.557	79	4.859	74	4.377	82
10 níveis	364.242	4.563	79	4.860	74	4.390	82

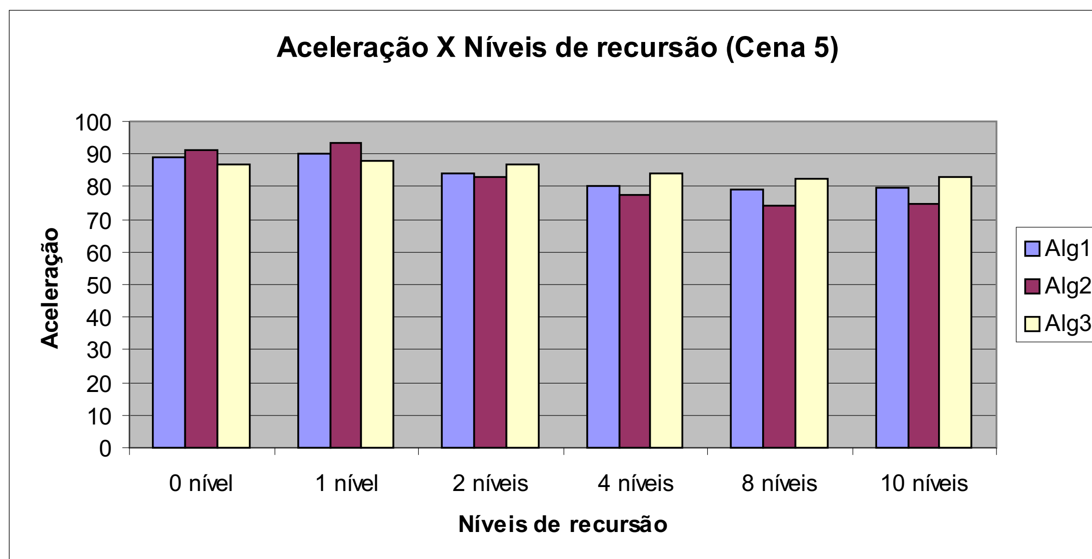


Figura 4.9: Traçado de raios: aceleração × níveis de recursão (cena 5).

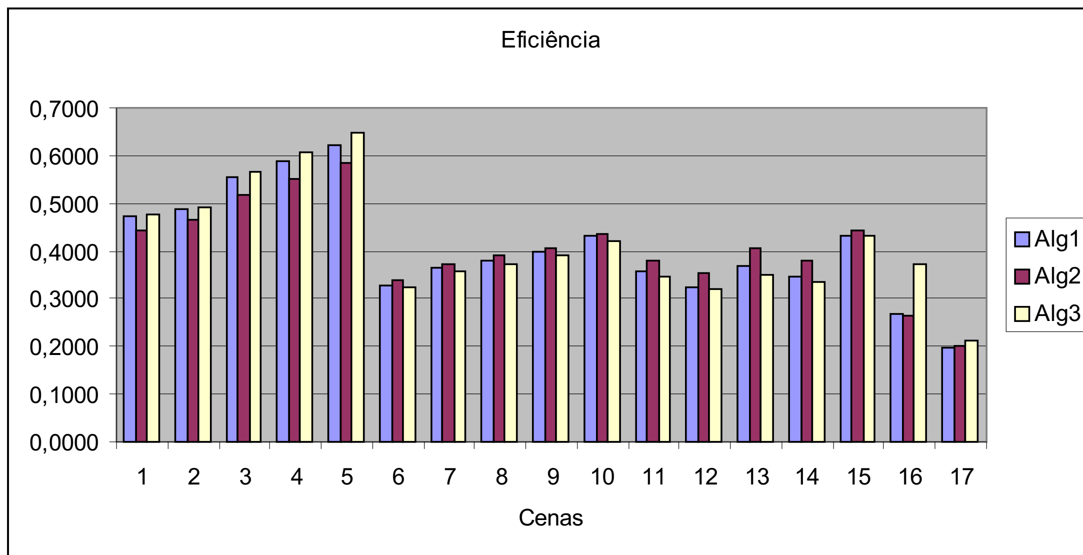


Figura 4.10: Traçado de raios: eficiência.

4.5 Construção da BVH

Nesta seção realizamos uma análise da performance obtida pelo algoritmo de construção da BVH em GPU. Na tabela 4.9 apresentamos os tempos obtidos pelo algoritmo de construção da BVH em CPU e GPU, bem como a aceleração obtida para algumas das cenas já relacionadas.

Tabela 4.9: Relação de cenas para construção em GPU.

Cena	Tempo CPU (ms)	Tempo GPU (ms)	Aceleração
18	38	59	0.64
6	1.717	2.489	0.68
11	3.582	4.803	0.74
17	6.041	5.696	1.06
12	7.601	9.643	0.78
19	7.624	9.188	0.82

Na figura 4.11 podemos observar que há uma tendência de obtenção de melhores acelerações para cenas com maior número de nós na BVH. Este fato está provavelmente relacionado a utilização de um maior número de processadores da GPU, em determinado nível de divisão. Este comportamento porém não é observado para todos os casos, possivelmente por conta de diferenças na distribuição de triângulos na cena. Também observamos que, dentre as cenas observadas, apenas a cena de número 17 produziu uma aceleração positiva.

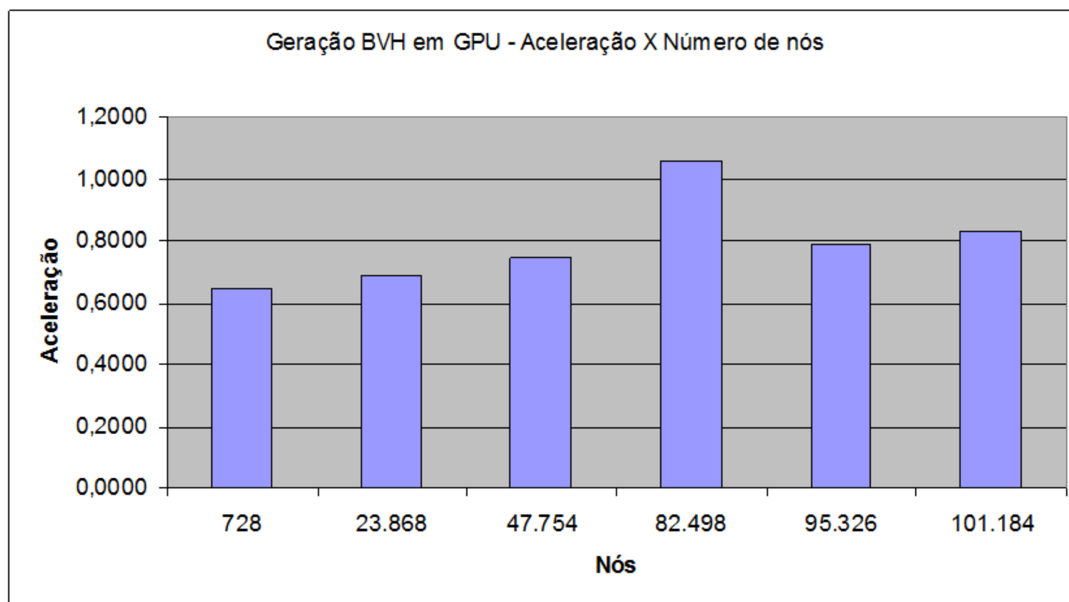


Figura 4.11: Geração BVH em GPU - Aceleração X número de nós.

Na figura 4.12 fazemos uso da cena de número 12 para verificar o comportamento da divisão dos nós em cada nível. Visualizamos que nos primeiros níveis temos poucos nós a serem divididos, portanto teremos poucos processadores da GPU em utilização. Além do fato de termos poucos processadores trabalhando nos primeiros níveis, cada um destes processadores está lidando com um grande número de triângulos. Observamos um crescimento exponencial do número de nós gerados, provocado pelo aumento do número de processadores em uso e redução do número de triângulos em tratamento por cada um deles, que ocorre a cada nível de divisão. A partir do nível 18 o número de nós gerados é decrescente, o que ocorre porque alguns nós não devem mais sofrer divisão.

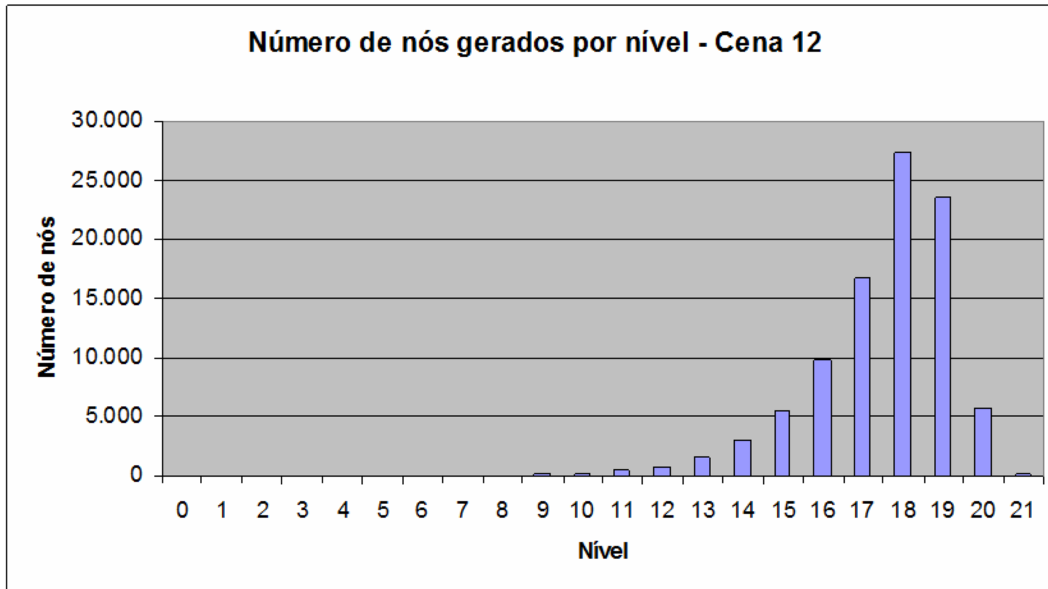


Figura 4.12: Número de nós gerados por nível (cena 12).

Na figura 4.13 visualizamos o tempo de geração dos nós da BVH para cada um dos níveis, bem como o número de nós produzidos em cada um dos níveis representados. Percebemos que, inicialmente, o tempo de geração de cada nível se reduz a medida que aumentamos o nível e que a partir de um determinado ponto, o tempo se mantém praticamente constante.

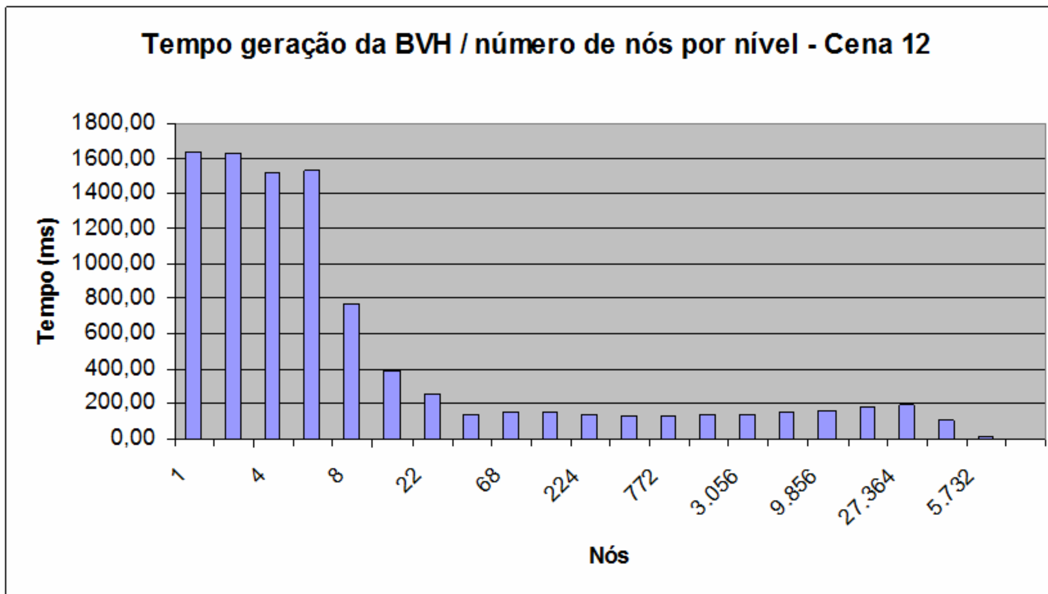


Figura 4.13: Tempo de geração da BVH X número de nós por nível (Cena 12).

A figura 4.14 apresenta um gráfico muito similar ao gráfico 4.11. De fato, o número de nós de uma BVH é geralmente proporcional ao número de triângulos e assim a aceleração se comporta de forma muito similar também.

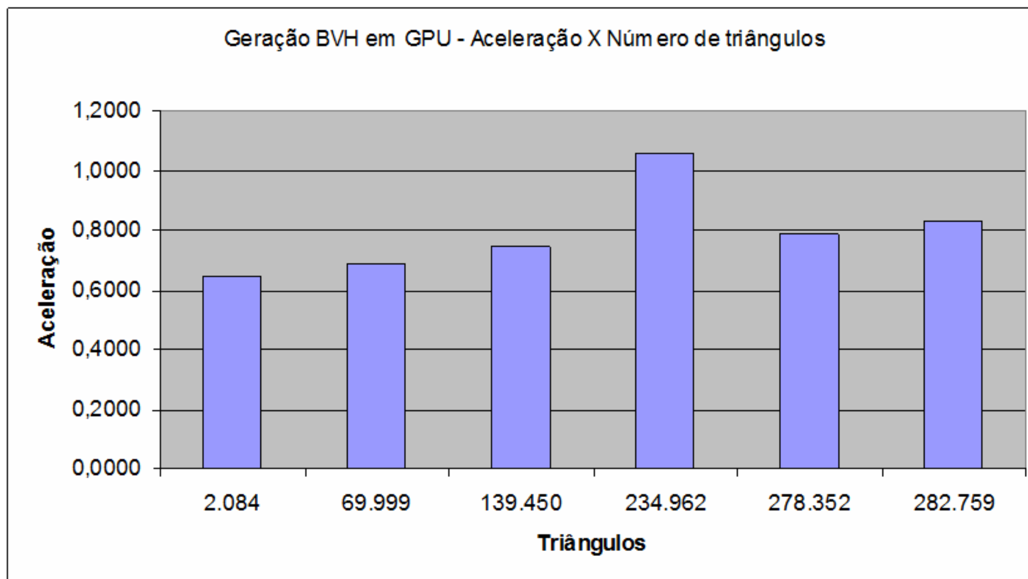


Figura 4.14: Geração BVH em GPU - Aceleração X número de triângulos.

4.6 Imagens

Nesta seção apresentamos as imagens relativas a todas as cenas descritas neste capítulo. Na figura 4.15 apresentamos a cena 1 e na figura 4.16 as cenas de 2 a 5, onde variamos o número de luzes.

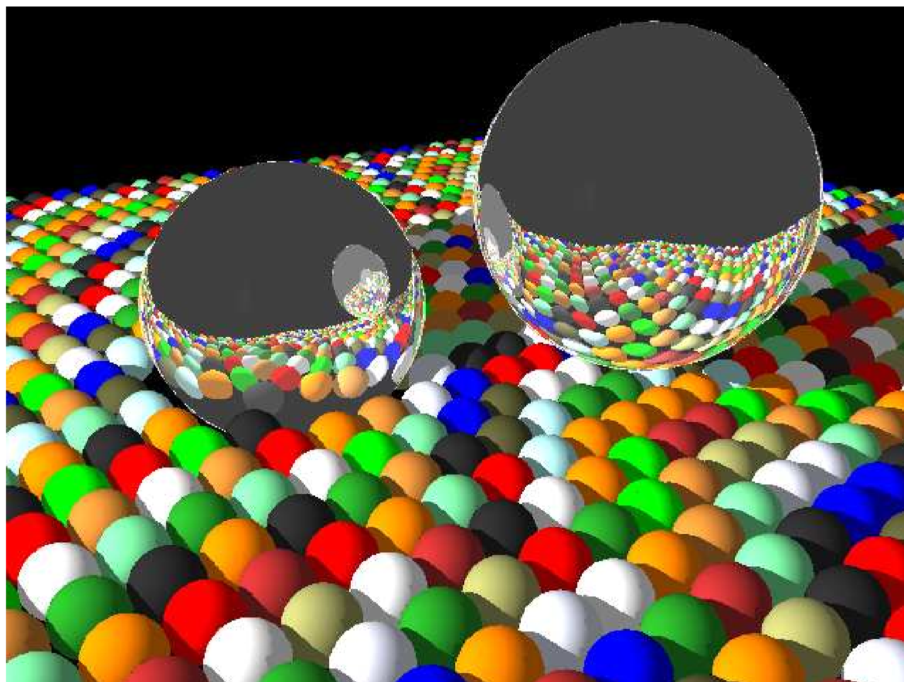
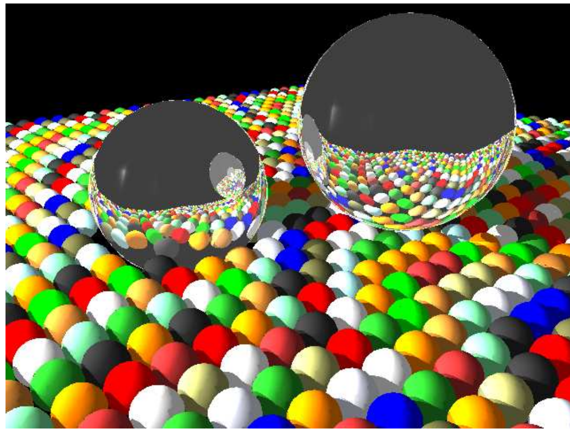
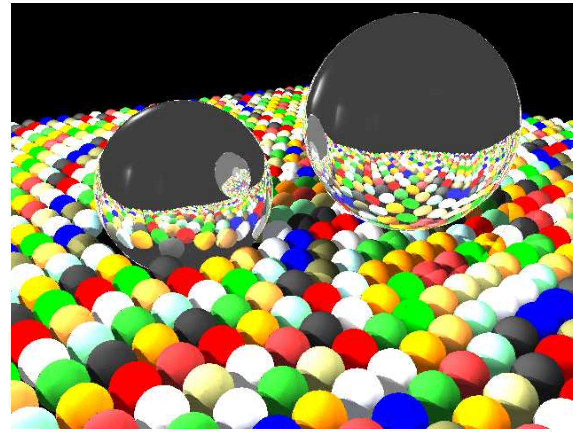


Figura 4.15: Cena 1.

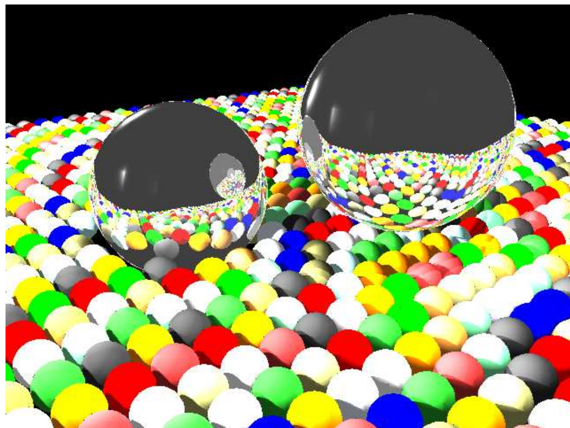
Nas figuras 4.17 e 4.18 apresentamos as cenas de número 6 a 15, que são representadas por coelhos entre espelhos e na figura 4.19 representamos as cenas 16 e 17,relativas ao golfinho e ao tabuleiro de xadrez.



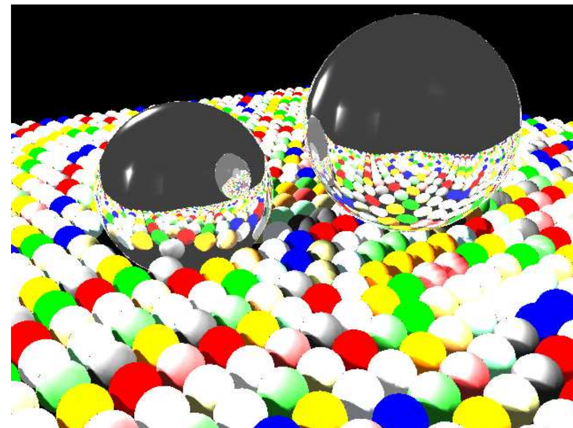
(a) Cena 2



(b) Cena 3



(c) Cena 4



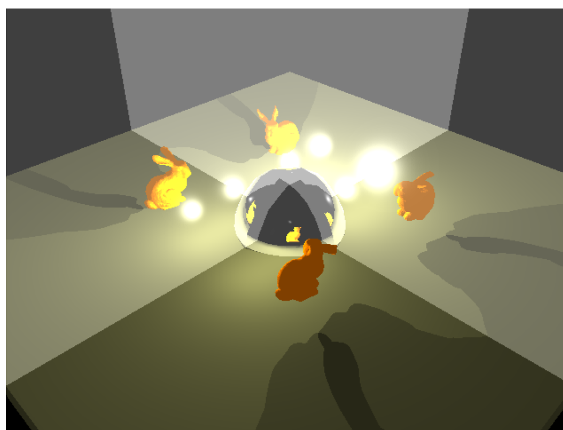
(d) Cena 5

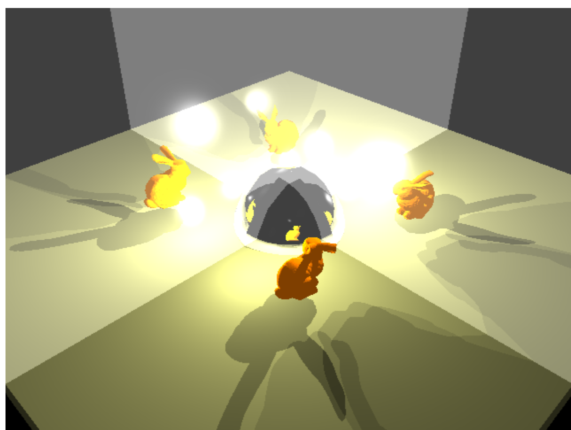
Figura 4.16: Cenas 2 a 5.

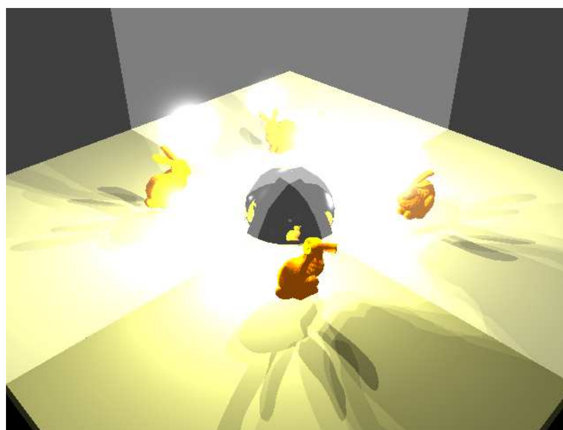
4.7 Comentários Finais

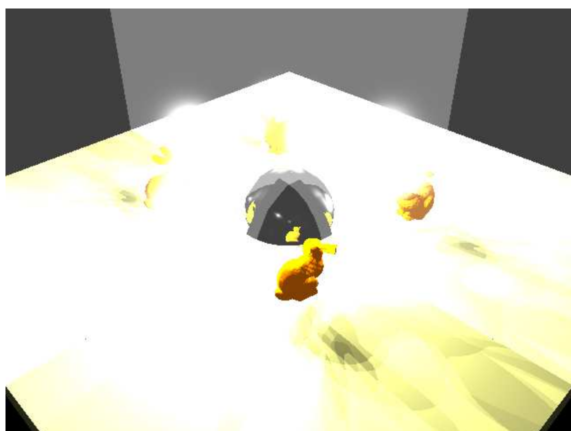
Neste capítulo apresentamos uma coleção de imagens produzidas pelo traçador de raios construído, bem como os resultados relativos a performance (tempos, aceleração e eficiência) para geração das mesmas. A cenas descritas tem por objetivo fazer uso intenso do traçado de raios, ou seja, procuram descrever elementos com grande número de triângulos em sua composição, elementos com propriedades reflexivas para aumento do número de raios a ser traçado, e número de luzes suficiente para medirmos as diferenças entre os algoritmos. As análises buscam comparar a performance dos três algoritmos em diversas situações, dentre elas, variando o número de luzes, variando o número de triângulos das cenas, o número de níveis de recursão e a resolução da imagem a ser produzida. Os números obtidos foram também apresentados na forma de gráficos para facilitar a leitura e estabelecimento de uma possível relação entre os resultados, que realizamos através de comentários pontuais. Como observamos no capítulo 3, os algoritmos possuem características particulares, principalmente no tocante ao número de kernels e balanceamento de carga, o que pode beneficiar um ou outro algoritmo, dependendo das características de cada cena, o que faz com que não haja um algoritmo que seja melhor que o outro em todas as situações. Também apresentamos neste capítulo os resultados obtidos pelo algoritmo de construção da BVH em GPU.

(a) Cena 6


(b) Cena 7

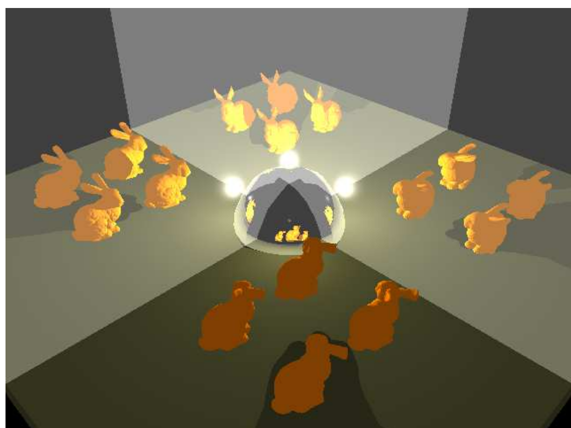

(c) Cena 8

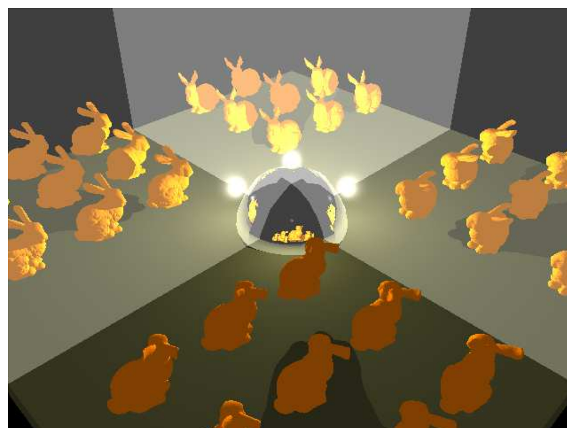

(d) Cena 9

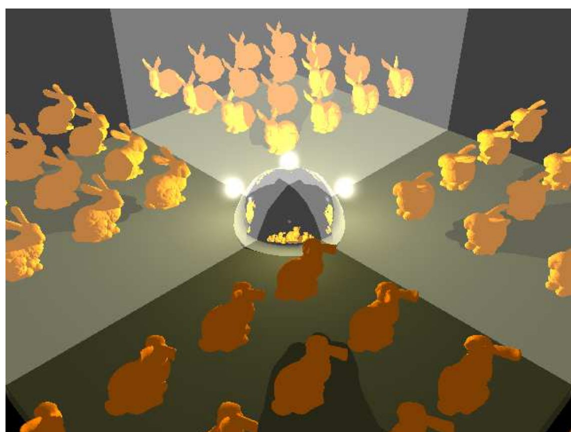

(e) Cena 10

(f) Cena 11

Figura 4.17: Cenas 6 a 11.

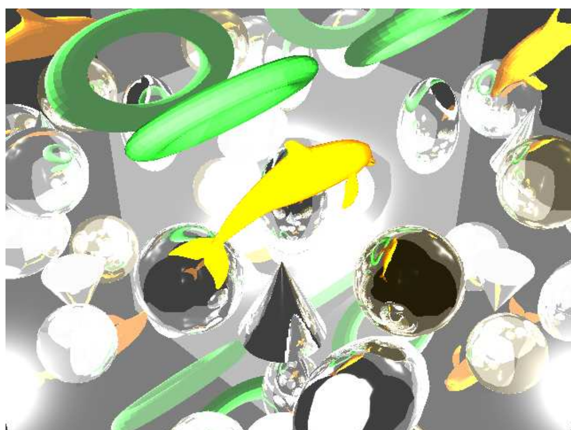

(a) Cena 12


(b) Cena 13


(c) Cena 14

(d) Cena 15

Figura 4.18: Cenas 12 a 15.


(a) Cena 16

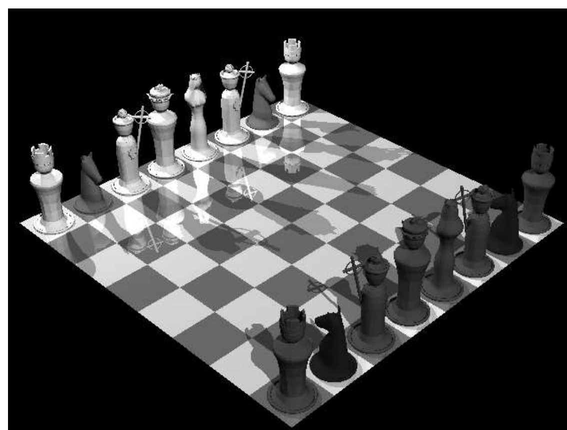

(b) Cena 17

Figura 4.19: Cenas 16 e 17.

Capítulo 5

Conclusão

5.1 Discussão dos Resultados Obtidos

O objetivo geral deste trabalho foi o desenvolvimento de um traçador de raios para cenas dinâmicas em CUDA. Primeiramente desenvolvemos um traçador de raios voltado para CPU, para que o mesmo pudesse ser utilizado para comparação de performance e também para verificação da validade dos resultados produzidos pelo traçador de raios em GPU.

Os objetivos específicos foram:

- *Estudo e implementação de uma estrutura de dados espacial para aceleração dos cálculos de interseção de raios com atores de cenas dinâmicas.* A revisão bibliográfica nos apresentou diversas possibilidades de estruturas de dados para aceleração do traçado de raios. Os estudos demonstravam que, para a maioria das situações, as estruturas de dados espaciais hierárquicas eram as mais apropriadas, dentre elas, especialmente kd-tree e BVH. A kd-tree consiste de uma estrutura hierárquica de aceleração que realiza uma subdivisão espacial da cena, e permite que um primitivo esteja contido em mais de um volume, enquanto uma BVH é composta por uma hierarquia de volumes envolventes, que é criada a partir dos triângulos da cena e onde um triângulo pode estar contido em apenas um volume.

  Para subdivisão dos nós da BVH optamos pelo uso de uma heurística (SAH), a fim de se produzir árvores de melhor qualidade, ou seja, mais balanceadas na média. A implementação e percurso destas estruturas de aceleração em CPU é facilitada pela possibilidade de recursão, mas em GPU, devido à ausência de recursão, deve ser realizada através de uma pilha. Concluiu-se que a BVH seria a estrutura de dados mais apropriada para tratamento de cenas dinâmicas em GPU, principalmente pela possibilidade de estimativa da memória exigida e de sua possibilidade de adaptação para cenas dinâmicas.

- *Projeto e implementação de algoritmos paralelos de traçado de raios em GPU.* Realizamos o projeto e implementação de três algoritmos de traçado de raios para GPU. Um dos requisitos para obtenção de performance na GPU é obter um considerável balanceamento de carga entre as threads, ou seja, devemos buscar manter sempre o maior número de threads ocupadas. No algoritmo 1, este objetivo foi alcançado através da compactação intermediária dos vetores de raios a serem traçados e tonalizados. No entanto, o algoritmo 1 não é capaz de tratar raios de reflexão e refração simultaneamente, pois em um mesmo ponto de interseção do raio com o objeto, podem ser gerados dois novos raios e isto não é suportado por este algoritmo.

  Para realizar o tratamento de reflexão e transparência, o algoritmo 2 foi desenvolvido em um único kernel (megakernel) e faz uso de uma pilha de raios. Quando dois novos raios são

gerados, o algoritmo insere um dos raios na pilha, para posterior tratamento, e prossegue com a execução do primeiro raio gerado. Quando um raio em tratamento atinge alguma das condições de parada do algoritmo de traçado de raios (peso mínimo ou nível máximo de recursão), um raio é retirado da pilha e traçado. Neste algoritmo, cada thread é responsável pelo traçado de todos os raios relativos à tonalização de um único pixel e a thread finaliza seu trabalho quando a pilha de raios estiver vazia.

Como o algoritmo 2 faz uso de um único kernel, a possibilidade de desbalanceamento de carga entre as threads é aumentada, tendo em vista que threads que terminam o traçado de raios precocemente são finalizadas apenas após a finalização de todas as outras threads do mesmo bloco. Dependendo da cena que esteja sendo gerada, pode ocorrer que grande parte dos raios finalizem seu traçado logo na primeira iteração, como exemplo tomamos raios que não interceptam nenhum objeto da cena e retornam cor de fundo. Em contraste, pode-se ter raios sendo tratados por threads deste mesmo bloco, que interceptam objetos reflexivos sucessivamente, e podem ser traçados até o último nível de recursão configurado. Neste cenário, todas as threads possuirão tempo de execução igual ao pior caso.

O algoritmo 3 procura tratar o desbalanceamento de carga das threads que é gerado neste primeiro nível de recursão, quando o raio não intercepta a cena. Após a geração e traçado dos raios primários, o algoritmo realiza uma compactação do vetor de raios, a fim de eliminar os raios que não atingiram objetos da cena. Para tanto, este algoritmo faz uso de um kernel para geração e traçado dos raios primários, um kernel para compactação do vetor de raios primários e posteriormente um kernel que realiza o traçado e tonalização dos raios secundários. O algoritmo faz uso de uma pilha de raios, assim como no algoritmo 2, para permitir o traçado de raios de reflexão e transparência.

A estrutura de dados de aceleração, BVH, pode ser gerada tanto em CPU quanto em GPU. O algoritmo de criação da BVH em GPU recebe como entrada um vetor T contendo a sopa de triângulos da cena e um vetor B contendo o primeiro nó da BVH e espaço suficiente para geração de todos os nós da BVH a ser gerada. O algoritmo faz uso de apenas um kernel para geração da BVH, onde cada cada bloco é responsável pela subdivisão de um nó da BVH por vez. O kernel de subdivisão executa em blocos de 96 threads, 32 threads (um warp) para cada eixo cartesiano. Cada thread de um warp é responsável pela avaliação, em paralelo, da função SAH em cada um dos $k = 32$ planos de corte considerados, em cada direção, usando para isto primitivos paralelos como soma prefixa e redução. Como a subdivisão de um nó pode não ocorrer em um determinado nível da árvore, o algoritmo realiza a compactação do vetor de nós ativos, com o objetivo de reduzir o desbalanceamento entre as threads, e o kernel é invocado novamente para o conjunto de nós produzido na iteração anterior. Em um nível d da árvore podemos ter no máximo 2^d nós a serem divididos, portanto, nos primeiros níveis da árvore teremos poucas threads sendo ocupadas, em um nível intermediário possivelmente teremos o maior número de threads trabalhando e após o nível intermediário, teremos nós da BVH que já terão terminado seu processo de subdivisão.

- *Projeto e implementação de um algoritmo paralelo para construção da estrutura de aceleração, BVH, em GPU*

- *Testes de desempenho do traçador de raios* Realizamos os testes de performance do algoritmo para CPU e dos três algoritmos para GPU. Nos testes buscamos obter uma relação a fim de justificar os resultados alcançados por cada algoritmo e motivar a implementação de novas soluções. Os resultados são avaliados através da variação de luzes de uma cena, resolução da imagem final, níveis de recursão e número de triângulos.

- *Testes de desempenho da construção da BVH em GPU*

Podemos concluir, então, que todos os objetivos estabelecidos foram alcançados e o traçado de raios em GPU é uma alternativa aplicável para aceleração da renderização de imagens.

Com relação aos resultados de performance obtidos, no traçado de raios primários, fazendo uso do algoritmo 2, atingimos uma *speedup* de 91 em relação ao traçado de raios primários em CPU (obtido com a cena de número 5). Para o traçado de raios com 10 níveis de recursão, fazendo uso do algoritmo 3, obtivemos um *speedup* de 83 (também para a cena de número 5). O maior *speedup* observado foi 93, obtido pelo algoritmo 2, para o traçado de raios com 1 nível de recursão (para a cena 5).

Como o modelo de GPU utilizado em nossos testes não é atualmente um modelo topo de linha, acreditamos que resultados relativamente superiores podem ser obtidos através do uso de dispositivos mais atuais. Apesar deste fato, os algoritmos desenvolvidos, para algumas cenas, produzem imagens a taxas aceitáveis para aplicações interativas.

Outros autores relatam ter renderizado imagens em tempo real, o que não foi possível obter em nosso traçador de raios, com o hardware disponível para os testes. Estes trabalhos se concentram na descrição e no algoritmo de percurso das estruturas de dados de aceleração dos cálculos de interseção dos raios com os objetos da cena e fornecem pouca ou nenhuma informação sobre os detalhes de implementação do traçado de raios em GPU propriamente dito. Isto impediu que os números apresentados nestes trabalhos pudessem ser comprovados. Além disto, como já citamos, a performance do traçado de raios está intimamente relacionada às características das cenas utilizadas, que em alguns casos, não apresentava a totalidade dos detalhes necessários para sua reprodução exata. Frequentemente, as descrições das cenas contidas nos trabalhos estudados descrevem o número de triângulos, número de nós da estrutura de aceleração (BVH ou kd-tree), mas omitem por exemplo, a quantidade de luzes da cena, a posição da câmera, o coeficiente de decaimento dos raios disparados, os índices de reflexibilidade dos atores, etc.

5.2 Trabalhos Futuros

Como trabalhos futuros sugerimos:

- A realização de traçado de raios distribuído para tratamento de anti-aliasing.
- A inclusão de mapa de textura para texturização dos objetos da cena.
- A implementação e testes de mecanismos adaptativos para construção da BVH, onde o número de cestos considerados seja variável conforme o número de triângulos ou espaçamento entre eles. É possível que estas implementações possam produzir a BVH em menor tempo sem grandes prejuízos na qualidade da BVH, produzida.

Apêndice A

Gramática do Leitor de Cenas

A.1 Especificação de Cenas

Uma cena é especificada em um arquivo texto contendo as descrições de atores e, opcionalmente, luzes, câmera, ambiente e ajustes globais, bem como declarações para uso posterior de modelos, materiais, transformações e expressões envolvendo vetores, cores e números reais. Nos trechos de gramática a seguir, símbolos terminais correspondentes a caracteres são escritos entre aspas simples (por exemplo, '{') e os demais, como literais e palavras reservadas, em negrito (por exemplo, **sphere**). Itens opcionais são delimitados por (e)? (por exemplo, (Expressão)? denota expressão opcional). Itens delimitados por (e)* e por (e)+ denotam, respectivamente, zero ou mais e uma ou mais repetições (por exemplo, Vetor (',' Vetor)* significa uma seqüência de um ou mais vetores ou pontos separados por vírgula, e (inteiro Vetor)+ significa uma sequência não vazia de inteiros sucedidos por vetores ou pontos).

A.2 Atores e Modelos

Um ator é um objeto de cena caracterizado por um modelo geométrico primitivo. Um primitivo é uma malha de triângulos definida pelas seguintes produções:

```
Primitivo:
  Malha
  | Caixa
  | Esfera

Malha:
  mesh
  '{'
      (Arquivo_OBJ | Dados_De_Malha)
      (Modificador_De_Objeto)*
  '}'

Arquivo_OBJ:
  file '"' nome_de_arquivo '"'

Dados_De_Malha:
  vertices
```

```
  '{'
      Inteiro
      (Vetor)+
  '}'
  (normals
  '{'
      Inteiro
      (Vetor)+
  '}')?
  triangles
  '{'
      Inteiro
      (Três_Índices ('/'Três_Índices)?)+
  '}'

Três_Índices:
  '<' inteiro ',' inteiro ',' inteiro '>'

Caixa:
  box
  '{'
      center Vetor
      orientation Vetor
      scale Vetor
      (Modificador_De_Objeto)*
  '}'

Esfera:
  sphere
  '{'
      center Vetor
      radius Real
      (Modificador_De_Objeto)*
   '}'
```

As duas últimas produções definem malhas de triângulos em forma de caixa e esfera, respectivamente. Malhas podem ser lidas de arquivos no formato Wavefront OBJ, como a seguir :

```
mesh { file "f-16.obj" }
```

Além disso, uma malha pode ser definida explicitamente por vértices, normais de vértices (opcionais) e triângulos. Um triângulo tem seus três vértices dados por uma tripla de inteiros (a partir de 0), cada inteiro sendo o índice de um vértice. As normais aos vértices de um triângulo, se especificados, também são dadas por uma tripla de inteiros (a partir de 0) e separadas por '/' da tripla de índices dos vértices. O exemplo a seguir define a malha de triângulos de um cubo unitário:

```
mesh
{
  vertices
```

```
  {
    8
    <-0.5, -0.5, 0.5>
    <0.5, -0.5, 0.5>
    <0.5, 0.5, 0.5>
    <-0.5, 0.5, 0.5>
    <-0.5, -0.5, -0.5>
    <0.5, -0.5, -0.5>
    <0.5, 0.5, -0.5>
    <-0.5, 0.5, -0.5>
    normals
    {
      6
      <0, -1, 0>
      <1, 0, 0>
      <0, 1, 0>
      <-1, 0, 0>
      <0, 0, 1>
      <0, 0, -1>
    }
    triangles
    {
      12
      <0, 4, 1>/<0, 0, 0>
      <4, 5, 1>/<0, 0, 0>
      <1, 5, 2>/<1, 1, 1>
      <5, 6, 2>/<1, 1, 1>
      <2, 6, 3>/<2, 2, 2>
      <6, 7, 3>/<2, 2, 2>
      <3, 7, 0>/<3, 3, 3>
      <7, 4, 0>/<3, 3, 3>
      <0, 1, 2>/<4, 4, 4>
      <2, 3, 0>/<4, 4, 4>
      <4, 6, 5>/<5, 5, 5>
      <6, 4, 7>/<5, 5, 5>
    }
}
```

Um modificador de objeto é a especificação do material da superfície do objeto ou uma sequência de transformações aplicadas ao objeto :

```
Modificador_De_Objeto:
  Material
  | Transformação
```

Um primitivo pode ser especificado em uma declaração para posteriormente ser utilizado. O exemplo a seguir declara uma (malha de triângulos na forma de uma) esfera chamada `bola`, com centro na origem e raio unitário. A declaração cria globalmente a variável `bola`, mas não adiciona o objeto à cena.

```
define bola sphere
{
  center <0,,0,0>
  radius 1
}
```

Um ator é definido pela produção :

```
Ator:
  Objeto

Objeto:
  Primitivo
  | object nome
  | object '{' nome (Modificador_De_Objeto)* '}'
```

As duas últimas produções adicionam à cena atores cujos modelos geométricos são instâncias de objetos anteriormente declarados. Por exemplo,

```
object bola
```

adiciona à cena um ator cujo modelo geométrico é uma instância do objeto `bola`. O exemplo a seguir define dois atores elipsoidais a partir do objeto `bola` anteriormente declarado :

```
object
{
  bola
  transform
  {
    scale <0,0,0> <2,1,1>
    translate <-1,-1,0>
  }
}

object
{
  bola
  transform
  {
    scale <0,0,0> <1,2,1>
    translate <+1,+1,0>
  }
}
```

A.3 Luzes

Os dois tipos de fontes de luz possíveis são : luz pontual e luz direcional. Uma fonte de luz pontual emite luz de uma determinada cor, a partir de determinado ponto, em todas as direções. Uma fonte de luz direcional emite luz de determinada cor, em direção constante. Uma fonte de luz é definida pelas seguintes produções :

```
Luz:
  Luz_Pontual
  | Luz_Direcional
  | light nome

Luz_Pontual:
  light
  '{'
    position Vetor
    (color Cor)?
    (falloff Inteiro)?
  '}'

Luz_Direcional:
  light
  '{'
    direction Vetor
    (color Cor)?
  '}'
```

Uma luz também pode ser declarada para uso posterior. O exemplo a seguir declara uma luz pontual branca, localizada na origem chamada `luz1`:

```
define luz1 light
{
  position <0,0,0>
  color rgb(1,1,1)
}
```

Para adiconar `luz1` à cena, escreve-se :

```
light luz1
```

A.4 Câmera

Uma câmera é definida pelas seguintes produções :

```
Câmera:
  camera
  '{'
    Tipo_de_Projeção
    position Vetor
    to Vetor /* ponto focal */
    up Vetor
    (angle Real)?
  '}'
  | camera nome

Tipo_de_Projeção:
```

```
  parallel
  | perspective
```

A.5 Ambiente

O ambiente de uma cena é caracterizado pela cor de fundo, cor de luz ambiente e índice de refração do meio no qual os objetos da cena estão inseridos. O ambiente é definido pela seguinte produção :

```
Ambiente:
  environment
  '{'
    (background Cor)?
    (ambient Cor)?
    (ior Real)?
  '}'
```

A.6 Ajustes Globais

Ajustes globais são valores de algumas variáveis de controle do programa de traçado de raios, definidos pelas seguintes produções:

```
Ajustes_Globais:
  settings
  '{'
    (image_width Inteiro)?
    (image_height Inteiro)?
    (min_weight Inteiro)?
    (max_recursion_level Inteiro)?
  '}'
```

A.7 Materiais

O material da superfície de um objeto é definido pelas seguintes produções :

```
Material:
  material
  '{'
    color Cor
    (Fatores_De_Pigmento)?
  '}'
  | material nome

Fatores_de_Pigmento:
  finish
  '{'
    (ambient Real)?
```

```
    (diffuse Real)?
    (spot Real)?
    (shine Real)?
    (specular Real)?
    (transparency Real)?
    (ior Real)?
  '}'
```

A.8 Transformações

Uma transformação é definida pelas seguintes produções :

```
Transform:
  transform
  '{'
    (Transform_Afim)*
  '}'
  | transform nome

Transform_Afim:
  translate Vetor
  rotate Vetor Vetor Real /* <base> <dir> <ângulo> */
  rotate_x Real
  rotate_y Real
  rotate_z Real
  scale Vetor Vetor /* <base> <fatores> */
```

A.9 Declarações

Declarações (globais) são definidas pelas seguintes produções :

```
Declaração:
  define nome ('=')? Definição

Definição:
  Objeto
  | Luz
  | Câmera
  | Material
  | Transform
```

A.10 Expressões

Expressões são definidas pelas seguintes produções :

```
Ponto:
  Expressão
```

```
Real:
  Expressão

Inteiro:
  Expressão

Cor:
  Expressão

Expressão:
  Termo
  | Expressão '+' Termo
  | Expressão '-' Termo

Termo:
  Fator
  |Termo '*' Fator
  | Termo '/' Fator

Fator:
  '(' Expressão ')'
  | '+' Fator
  | '-' Fator
  | nome
  | inteiro
  | real
  | '<' Expressão ',' Expressão ',' Expressão '>' /*vetor*/
  | rgb '(' Expressão ',' Expressão ',' Expressão ')'
```

A.11 Inclusão de Arquivos

A inclusão de um arquivo contendo definições em uma cena é definida pela seguinte produção :

```
Inclusão_de_Arquivo:
  include '"' nome_de_arquivo '"'
```

Bibliografia

[1] Amanatides, J., and Woo, A. A fast voxel traversal algorithm for ray tracing. In *In EUROGRAPHICS 87* (1987), pp. 3–10.

[2] Appel, A. Some techniques for shading machine renderings of solids. In *AFIPS Spring Joint Computing Conference* (1968), pp. 37–45.

[3] Arvo, J., and Kirk, D. A Survey of Ray Tracing Acceleration Techniques. In *An Introduction to Ray Tracing* (London, UK, UK, 1989), Academic Press Ltd., pp. 201–262.

[4] Ashdown, I. *Radiosity: a programmer's perspective.* John Wiley & Sons, Inc., New York, NY, USA, 1995.

[5] Blinn, J. F. Models of Light Reflection for Computer Synthesized Pictures. *SIGGRAPH Comput. Graph. 11*, 2 (1977), 192–198.

[6] Cabral, B., Max, N., and Springmeyer, R. Bidirectional Reflection Functions from Surface Bump Maps. In *SIGGRAPH '87: Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1987), ACM, pp. 273–281.

[7] Carr, N. A., Hall, J. D., and Hart, J. C. The Ray Engine. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (Aire-la-Ville, Switzerland, Switzerland, 2002), EUROGRAPHICS Association, pp. 37–46.

[8] Christen, M. Implementing Ray Tracing on GPU. Tech. rep., Diploma thesis, University of Applied Sciences, 2004.

[9] Cline, D., Steele, K., and Egbert, P. Lightweight Bounding Volumes for Ray Tracing. *Journal of Graphics Tools 11*, 4 (2006), 61–71.

[10] Cook, R. L., and Torrance, K. E. A Reflectance Model for Computer Graphics. In *SIGGRAPH '81: Proceedings of the 8th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1981), ACM, pp. 307–316.

[11] CUDA-zone. CUDA Zone. *http://www.nvidia.com/object/cuda_home_new.html*. Acessado em 10 de março de 2008.

[12] Eisemann, M., Grosch, T., Magnor, M., and Mueller, S. Automatic Creation of Object Hierarchies for Ray Tracing Dynamic Scenes. In *WSCG Short Papers Post-Conference Proceedings* (1 2007), V. Skala, Ed., WSCG, pp. 57–64.

[13] Ernst, M., Vogelgsang, C., and Greiner, G. Stack Implementation on Programmable Graphics Hardware. In *VMV04* (2004), pp. 255–262.

[14] Foley, T., and Sugerman, J. Kd-tree Acceleration Structures for a GPU Raytracer. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (New York, NY, USA, 2005), ACM, pp. 15–22.

[15] GlassnerIntro, A. S., Ed. *An Introduction to Ray Tracing.* Academic Press Ltd., London, UK, UK, 1989.

[16] Goldsmith, J., and Salmon, J. Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics and Applications 7*, 5 (1987), 14–20.

[17] GPGPU-org. GPGPU.org. *http://gpgpu.org.* Acessado em 10 de março de 2008.

[18] Günther, J., Popov, S., Seidel, H.-P., and Slusallek, P. Realtime Ray Tracing on GPU with BVH-based Packet Traversal. In *Proceedings of the IEEE/EUROGRAPHICS Symposium on Interactive Ray Tracing 2007* (Sept. 2007), pp. 113–118.

[19] Havran, V. *Heuristic Ray Shooting Algorithms.* Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[20] Havran, V., Prikryl, J., and Purgathofer, W. Statistical Comparison of Ray-Shooting Efficiency Schemes. Tech. Rep. TR-186-2-00-14, Institute of Computer Graphics and Algorithms, Vienna University of Technology, May 2000. human contact: technical-report@cg.tuwien.ac.at.

[21] He, X. D., Torrance, K. E., Sillion, F. X., and Greenberg, D. P. A Comprehensive Physical Model for Light Reflection. In *SIGGRAPH '91: Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1991), ACM, pp. 175–186.

[22] Horn, D. R., Sugerman, J., Houston, M., and Hanrahan, P. Interactive kd-tree GPU Ray Tracing. In *I3D '07: Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2007), ACM, pp. 167–174.

[23] Kajiya, J. T. Anisotropic Reflection Models. In *SIGGRAPH '85: Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1985), ACM, pp. 15–21.

[24] Karlsson Filip, L. C. J. Ray Tracing Fully Implemented on Programmable Graphics Hardware. Master's thesis, Chalmers University of Technology, 2004.

[25] Larsson, T., and Akenine-möller, T. Strategies for Bounding Volume Hierarchy Updates for Ray Tracing of Deformable Models. Tech. rep., Centre, Maelardalen University, 2003.

[26] Lauterbach, C., eui Yoon, S., and Manocha, D. RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs. In *In Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 39–45.

[27] Lauterbach, C., Garland, M., Sengupta, S., Luebke, D., and Manocha, D. Fast BVH Construction on GPUs. *Comput. Graph. Forum 28*, 2 (2009), 375–384.

[28] MacDonald, D. J., and Booth, K. S. Heuristics for Ray Tracing using Space Subdivision. *Vis. Comput. 6*, 3 (1990), 153–166.

[29] Miller, G. S. P. From Wire-frames to Furry Animals. In *Proceedings on Graphics Interface '88* (Toronto, Ont., Canada, Canada, 1988), Canadian Information Processing Society, pp. 138–145.

[30] NVIDIA. GeForce 8800 GPU Architecture Overview, 2006. *http://www.nvidia.com/object/IO_37100.html*. Acessado em 10 de março de 2008.

[31] NVIDIA. CUDA Programming Guide, 2007. *http://developer.download.nvidia.com/compute/cuda/1_1/nvidia_cuda_programming_guide_1.1.pdf*. Acessado em 15/9/2009.

[32] NVIDIA. Optimizing CUDA, 2008. *http://gpgpu.org/wp/wp-content/uploads/2009/06/04-OptimizingCUDA.pdf*. Acessado em 15/9/2009.

[33] NVIDIA. CUDA Basics, 2009. *http://gpgpu.org/wp/wp-content/uploads/2009/06/02-CUDA_basic.pdf. Acessado em 15/9/2009.*

[34] Phong, B.-T. Illumination for Computer Generated Pictures. *j-CACM 18*, 6 (1975), 311–317.

[35] Popov, S., Günther, J., Seidel, H.-P., and Slusallek, P. Experiences with Streaming Construction of SAH kd-trees. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (Sept. 2006), pp. 89–94.

[36] Popov, S., Günther, J., Seidel, H.-P., and Slusallek, P. Stackless kd-tree Traversal for High Performance GPU Ray Tracing. *Computer Graphics Forum 26*, 3 (September 2007), 415–424.

[37] Poulin, P., and Fournier, A. A Model for Anisotropic Reflection. In *SIGGRAPH '90: Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1990), ACM, pp. 273–282.

[38] Purcell, T. J. *Ray tracing on a Stream Processor*. PhD thesis, Stanford University, Stanford, CA, USA, 2004. Adviser-Hanrahan,, Patrick M.

[39] Purcell, T. J., Buck, I., Mark, W. R., and Hanrahan, P. Ray Tracing on Programmable Graphics Hardware. *ACM Transactions on Graphics 21*, 3 (July 2002), 703–712.

[40] Smits, B. Efficiency Issues for Ray Tracing. *J. Graph. Tools 3*, 2 (1998), 1–14.

[41] Smits, B. Efficiency Issues for Ray Tracing. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses* (New York, NY, USA, 2005), ACM, p. 6.

[42] Stewart, N., Leach, G., and John, S. An Improved Z-Buffer CSG Rendering Algorithm. In *In HWWS 98: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics hardware* (1998), pp. 25–30.

[43] Szirmay-Kalos, L., Havran, V., Balázs, B., and Szécsi, L. On the Efficiency of Ray-shooting Acceleration Schemes. In *SCCG '02: Proceedings of the 18th Spring Conference on Computer Graphics* (New York, NY, USA, 2002), ACM, pp. 97–106.

[44] Wald, I. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004.

[45] Wald, I. On fast construction of SAH-based bounding volume hierarchies. In *In Proceedings of the 2007 IEEE/EG Symposium on Interactive Ray Tracing. IEEE* (2007), pp. 33–40.

[46] Watkins, C., Coy, S., and Finlay, M. *Photorealism and Ray Tracing in C, with Disk*. Henry Holt and Co., Inc., New York, NY, USA, 1992.

[47] Woop, S., Marmitt, G., and Slusallek, P. Bkd-trees for Hardware Accelerated Ray Tracing of Dynamic Scenes. In *GH '06: Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware* (New York, NY, USA, 2006), ACM, pp. 67–77.

[48] Wächter, C., and Keller, A. Instant Ray Tracing: The Bounding Interval Hierarchy.

[49] Zhukov, S., Inoes, A., and Kronin, G. An Ambient Light Illumination Model. In *Rendering Techniques 98* (1998), G. Drettakis and N. Max, Eds., EUROGRAPHICS, Springer-Verlag Wien New York, pp. 45–56.