Pós-Graduação em Ciência da Computação

"Estruturas de Aceleração para *Ray Tracing* em Tempo Real: um Estudo Comparativo"

Por

Artur Lira dos Santos

Dissertação de Mestrado

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE, MARÇO/2011

Catalogação na fonte
Bibliotecária Jane Souto Maior, CRB4-571

Santos, Artur Lira dos

Estruturas de aceleração para Ray Tracing em tempo real: um estudo comparativo / Artur Lira dos Santos - Recife: O Autor, 2011.

viii, 98 folhas: il., fig., gráf.

Orientador: Veronica Teichrieb.

Dissertação (mestrado) - Universidade Federal de Pernambuco. CIn, Ciência da Computação, 2011.

Inclui bibliografia.

1. Ciência da Computação. 2. Computação gráfica. 3. Ray Tracing. I. Teichrieb, Veronica (orientador). II. Título.

004 CDD (22. ed.) MEI2011 – 153

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ARTUR LIRA DOS SANTOS

als3@cin.ufpe.br

ESTRUTURAS DE ACELERAÇÃO PARARAY TRACING EM TEMPO REAL: UM ESTUDO COMPARATIVO

ESTA DISSERTAÇÃO FOI SUBMETIDA PARA O CENTRO DE INFORMÁTICA DA UNIVERSIDADE FEDERAL DE PERNAMBUCO COMO REQUERIMENTO PARCIAL PARA OBTER O GRAU DE MESTRE EM CIÊNCIA DA COMPUTAÇÃO.

ORIENTADORA: VERONICA TEICHRIEB.

RECIFE, FEVEREIRODE 2011.

Dissertação de Mestrado apresentada por **Artur Lira dos Santos** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título **"Estruturas de Aceleração para Ray Tracing em Tempo Real: um Estudo Comparativo"**, orientada pela **Profa. Verônica Teichrieb** e aprovada pela Banca Examinadora formada pelos professores:

Prof. Manoel Eusébio de Lima
Centro de Informática / UFPE

Prof. Alberto Barbosa Raposo
Departamento de Informática / PUC-RJ

Profa. Veronica Teichrieb
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 11 de março de 2011.

Prof. Nelson Souto Rosa
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.

Resumo

O poder computacional atual das GPUs possibilita a execução de complexos algoritmos massivamente paralelos, como algoritmos de busca em estruturas de dados específicas para *ray tracing* em tempo real, comumente conhecidas como estruturas de aceleração. Esta dissertação descreve em detalhes o estudo e implementação de dezesseis diferentes algoritmos de travessia de estruturas de aceleração, utilizando o *framework* de CUDA, da NVIDIA. Este estudo comparativo teve o intuito de determinar as vantagens e desvantagens de cada técnica, em termos de performance, consumo de memória, grau de divergência em desvios e escalabilidade em múltiplas GPUs. Uma nova estrutura de aceleração, chamada *Sparse Box Grid*, também é proposta, além de dois novos algoritmos de busca, focando em melhoria de performance. Tais algoritmos são capazes de alcançar *speedups* de até 2.5x quando comparado com implementações recentes de travessias em GPU. Como consequência, é possível obter simulação em tempo real de cenas com milhões de primitivas para imagens com 1408x768 de resolução.

Palavras-chave: *Ray Tracing*, GPU, Estruturas de Aceleração, *Sparse Box Grid*, RT^2.

Abstract

Current GPU computational power enables the execution of complex and parallel algorithms, such as ray tracing techniques supported by special data structures for 3D scene rendering in real time, commonly known as acceleration structures. The present dissertation describes in detail the study and implementation of sixteen different traversal algorithms for acceleration structures using the parallel framework NVIDIA CUDA, in order to point their pros and cons regarding performance, memory consumption, branch divergencies and scalability on multiple GPUs. In addition, a new acceleration structure, called Sparse Box Gridis proposed and also two new algorithms based on this analysis, aiming performance improvement. Both of them are capable of reaching speedup gains up to 2.5x when compared to recent and optimized parallel traversal implementations. As a consequence, real-time frame rates are possible for scenes with 1408x768 pixels of resolution and 3.6 million primitives.

Keywords: *Ray Tracing*, GPU, *Acceleration Structures*, *Sparse Box Grid*, RT^2.

Índice

Índice de Figuras

Índice de Algoritmos

Índice de Gráficos

1 Introdução

O crescimento acelerado da indústria de microprocessadores gráficos é consequência direta da incessante busca por melhorias gráficas em aplicativos de simulação 3D, como jogos e aplicações do tipo CAD (*Computer-Aided Design*). O próprio surgimento de *chips* gráficos [EBER03] no início da década de 90 é explicado pela necessidade do mercado de jogos em migrar de um *pipeline* gráfico 2D, baseado em sobreposição de imagens e *pixels*, para um capaz de simular cenas 3D eficientemente. Assim, *chips* gráficos foram projetados em conjunto com um novo *pipeline* gráfico 3D para a simulação de cenas virtuais. A parte principal desse *pipeline* é conhecida como rasterização (*rasterization*), *scanline* ou também *raster operations* [OGL05]. O termo vem do latim *rastrum*, relacionado ao ato de realizar um rastro ou esboço de uma representação 3D em um plano 2D utilizando apenas linhas e pontos. O objeto 3D é "escaneado" ou rastreado, de forma a encontrar quais são os pontos ou *pixels* da imagem que representam o objeto. Todo objeto 3D é representado de forma poligonal, sendo este um requisito básico para qualquer *pipeline* gráfico 3D baseado em rasterização. O OpenGL [OGL05] é um padrão que segue este paradigma.

Paralelamente, a técnica de *ray tracing* vem sendo utilizada, desde seu aparecimento com o trabalho de Appel [APPEL68] em 1968, como uma técnica robusta aplicada para a geração de imagens, baseando-se em fenômenos físicos naturais. Muitas pesquisas foram realizadas pela comunidade científica de forma a estabelecer técnicas baseadas ou que tiram proveito de *ray tracing* para renderização 3D realista, como *path tracing*[KJIY86] e *photon mapping*[JEN96]. Entretanto, como qualquer técnica de computação, uma abordagem como *path tracing* tem suas vantagens e desvantagens: enquanto os resultados visuais podem atingir uma qualidade próxima ao do olho humano, o poder computacional demandado o torna impraticável para seu uso em tempo real (taxa mínima de 30 quadros por segundo [PBRT07]) nos processadores atuais. Atualmente, os principais campos de aplicação de técnicas de *ray tracing* são os de animação em computação gráfica e efeitos especiais de cinema, campos que não requerem síntese de imagens em tempo real.

Entretanto, com a evolução acelerada dos microprocessadores, muitos trabalhos vêm sendo publicados [ALS09][JMMX10][OPT10][WCHT06] mostrando a possibilidade do uso da

técnica de *ray tracing* para simulação 3D em tempo real. Além de pesquisas no meio acadêmico, gigantes da indústria de microprocessadores como a Intel [INT10] e a NVIDIA [NVID10] visualizam num futuro próximo a utilização desta técnica em jogos e programas CAD, por oferecermelhor qualidade visual quando comparado com rasterização, mas sob a penalidade de maior custo computacional.

É importante observar que *ray tracing*a taxas acima de 30 quadros por segundo só se tornou possível a partir de estruturas de dados que organizam espacialmente ou agrupam de alguma maneira os objetos da cena, de forma a reduzir consideravelmente o número de primitivas a serem testadas. Tais estruturas de dados são comumente conhecidas como estruturas de aceleração. É no contexto de tais estruturas que este trabalho é definido.

1.1 Objetivos

O principal objetivo deste trabalho é a realização de um estudo comparativo de implementações em processadores modernos de várias estruturas de aceleração de *ray tracing*presentes na literatura. A partir deste estudo comparativo será possível definir qual ou quais estruturas são as mais adequadas em termos de performance para a realização de *ray tracing* em tempo real, levando em consideração parâmetros como o uso de memória e tempo de execução.

Além disso, tem-se objetivos específicos a serem alcançados, como:

- Pesquisa e experimentação com programação em GPGPU;
- Pesquisa de técnicas para alcançar melhorias qualitativas de síntese de imagens com *ray tracing*, como técnicas de *anti-aliasing*;
- Melhorias no *pipeline* do RT^2[ALSR09], biblioteca de *ray tracing* em tempo real utilizada como base para o estudo das estruturas de aceleração;
  Comparação do RT^2 com bibliotecas de *ray tracing* interativo ou em tempo real, como a OPTIX [OPT10], da NVIDIA [NVID10].

1.2 Organização da Dissertação

Os demais capítulos deste trabalho estão organizados da seguinte forma:

O segundo capítulo apresenta conceitos relacionados ao trabalho, incluindo revisão bibliográfica e definições teóricas.

O terceiro capítulo apresenta as estruturas de aceleração estudadas, fazendo parte da investigação sobre os trabalhos relacionados. Já o quarto capítulo apresenta o RT^2, biblioteca base para os estudos das estruturas de aceleração apresentadas no capítulo 3. O quarto capítulo também apresenta detalhes de implementação em GPU de algoritmos relacionados a *ray tracing*, como técnicas de *anti-aliasing* e paralelização em multi-GPU.

O quinto capítulo descreve a implementação dos dezesseis algoritmos de busca levantados na revisão bibliográfica. Tais algoritmos são parte das estruturas de aceleração descritas no capítulo 3, mas com modificações específicas para GPU. Assim, este capítulo também apresentadetalhes de implementação em GPU realizados neste trabalho.

O sexto capítulo apresenta os resultados encontrados e uma análise comparativa das estruturas implementadas.

O sétimo capítulo apresenta as conclusões, incluindo contribuições e trabalhos futuros.

Todas as referências utilizadas neste trabalho estão organizadas no capítulo de Referências.

Por fim, todos os algoritmos de travessia estudados neste trablaho estão descritos em pseudocódigo no Apêndice A.

2 Conceitos Básicos

O principal objetivo da Computação Gráfica (CG) é a de síntese de imagens a partir de dados que representam uma cena virtual. Tal síntese pode ser realizada de maneira *off-line*, mas com alto grau de realismo, ou em tempo real, com menor fidelidade gráfica, mas possibilitando interatividade. Dentro do contexto interativo, este capítulo apresenta os principais conceitos relacionados à CG para visualização 3D em tempo real. Esta descrição segue uma ordem cronológica de evolução da arquitetura dos principais *pipelines* gráficos estabelecidos no mercado, *pipelines* estes diretamente relacionados com o paradigma de rasterização poligonal. Este capítulo também descreve pesquisas relacionadas com técnicas de *ray tracing* em tempo real, implementadas em diferentes tipos de arquiteturas de processadores, relacionando suas vantagens e desvantagens.

2.1 Rasterização

No início da década de 90, fabricantes de microprocessadores disponibilizaram no mercado as primeiras placas gráficas com suporte à aceleração 3D [EBER03].Voltadas para o público doméstico, tais placas hoje são comumente chamadas de GPUs – *Graphics Processing Units*. Com este ganho computacional, proveniente principalmente da arquitetura de microprocessadores paralelos para operações massivas em ponto flutuante, tornou-se possível desenvolver jogos 3D interativos [WATT00], em que o jogador explora o ambiente virtual em três eixos ou dimensões, sendo assim uma experiência mais imersiva que a oferecida por jogos 2D. Esta interatividade atraiu o público de tal maneira que em menos de meia década[WATT00] se tornou padrão a presença de uma placa gráfica com suporte à aceleração 3D em um computador doméstico. Com esta forte demanda, o mercado de jogos cresce contemporaneamente aliado à evolução das GPUs. Além de jogos, aplicativos voltados para diversas áreas profissionais também passaram a tirar proveito das GPUs. Programas de modelagem 3D [3DS10] são hoje em dia muito utilizados em produções cinematográficas. Programas CAD (*Computer-Aided Design*) são aplicados na criação de desenhos técnicos, sendo parte fundamental de projetos em diversas áreas, como Arquitetura [ARCH10] e Design [ACAD10].

É importante observar que, para tornar possível a simulação de cenas 3D, não bastou apenas um *hardware*otimizado. Naturalmente, a definição de algoritmos e técnicas de geração de imagens de cenas 3D também foi necessária. Este conjunto de algoritmos é geralmente designado como técnicas de renderização. Tais GPUs oferecem então uma implementação otimizada em *hardware* de um *pipeline* de renderização por rasterização [WATT00]. As técnicas de rasterização, de maneira resumida, se baseiam principalmente nos seguintes procedimentos:

1. Transformação de todos os objetos da cena 3D em polígonos/vértices/linhas;
2. Ordenação dos vértices dos polígonos em relação ao espaço de câmera;
3. Definição dos fragmentos rastreados ou "rasterizados" no espaço de tela a partir dos polígonos da cena.

Para renderizar uma esfera, ela é transformada ou representada por um conjunto de polígonos, que posteriormente torna-se um grupo de triângulos e, por fim, dependendo de sua projeção no plano projetivo, tais triângulos são exibidos ou não na tela a partir de um processo de *scanline* ou *raster*.

É importante observar que, a rigor, o conceito de rasterização se limitaria apenas à etapa final de *raster*. Entretanto, por inicialmente ter sido a principal etapa do *pipeline* gráfico, o termo rasterização passou a ser utilizado também como um representante de todos os estágios (*pipeline*), se assemelhando com o próprio termo de renderização. O pipeline e seus estágios é ilustrado na Figura 2-1.

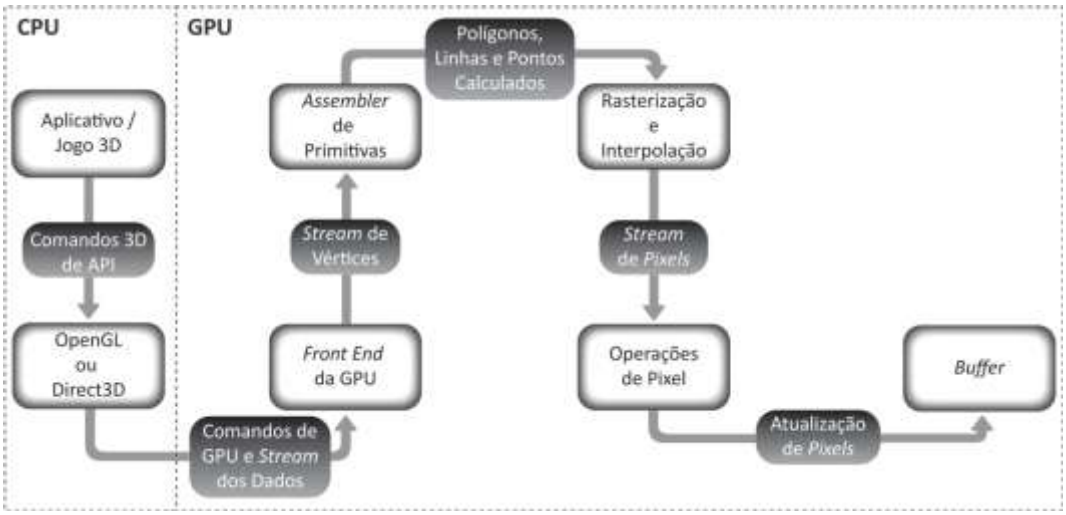


Figura 2-1. Pipeline fixo de renderização por rasterização.

Apesar da evolução e sofisticação do *pipeline* de rasterização, existem diversos tipos de problemas de representação que geram artefatos visuais que incomodam a percepção visual humana, como o serrilhado e o *aliasing*[JMMX10], descritos em detalhes na seção 4.3.3. Entretanto, por ser bem estabelecida, além de otimizada em *hardware*, a rasterização é a técnica-base para a maioria dos aplicativos de simulação3D atuais.

2.1.1 Linguagem de Shading

Apesar do alto ganho de performance oferecido pelas primeiras placas de vídeo, desenvolvedores de jogos enfrentavam o problema de baixa ou nula flexibilidade de programação do *pipeline* gráfico. A adição de efeitos visuais específicos necessitava de muitas adaptações e aproximações, de forma que não eram obtidos satisfatoriamente. Fez-se então necessária a criação de placas de vídeo que suportassem maior flexibilidade de programação, oferecendo ao desenvolvedor maior controle do *pipeline* gráfico. Dentro deste contexto surgem as linguagens de *shading* ou *shading languages* (SL), extensões ao *pipeline* fixo que possibilitam alterações em algumas etapas, como a de *vertex*, *fragment* e *geometryshading*. A GeForce 3 TI[NVID10] foi a primeira placa disponível capaz de suportar *shading* programável, mas apenas nas etapas de *pixel* e *vertexshading*,através de uma linguagem similar à Assembly x86.

Posteriormente novas placas surgiram adicionando suporte às outras etapas do *pipeline*. Contudo, ainda que certa flexibilidade fosse possível, o código em uma linguagem de baixo nível como Assembly dificultava sua programação. Para contornar este problema, surgiramas linguagens de *shading* de alto nível [NvCG10] as quais baseiam suas estruturas na linguagem C e, consequentemente, retiraram uma grande parte da complexidade da implementação, bem como, diminuíram o tempo de aprendizagem da linguagem, por se basearem numa linguagem amplamenteconhecida e difundida. O diagrama da Figura 2-2 demonstra as etapas do *pipeline* programável.

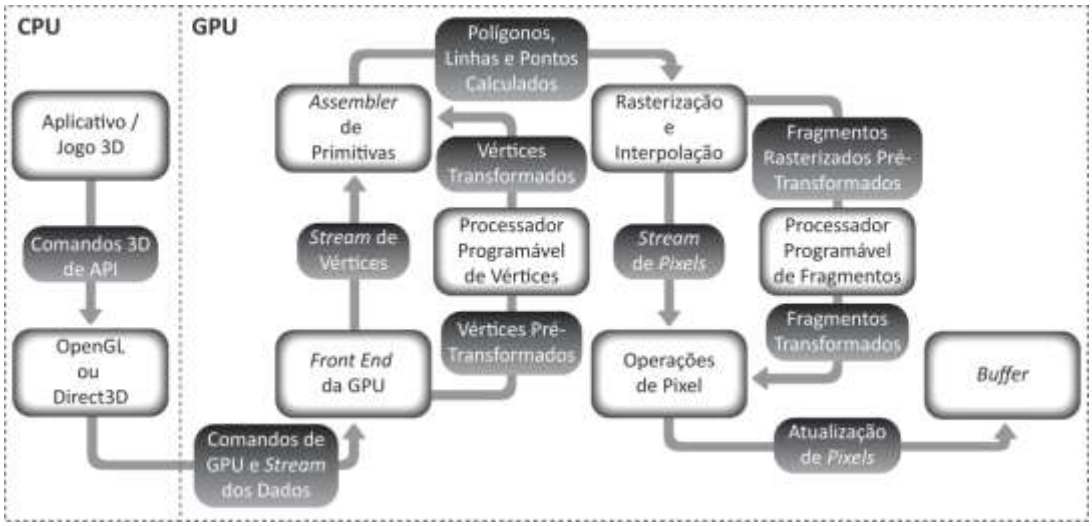


Figura 2-2. *Pipeline*programável com adição de *shaders*. Fonte: NVIDIA[NVID10].

A presença de *shaders*é essencial em um *gameengine*moderno, já que efeitos visuais complexos e de alta qualidade presentes em jogos são obtidos somente através de tais recursos. A etapa de iluminação e sombreamento é definida e realizada principalmente através de customização em SL. Como exemplo, efeitos dos mais simples como *Phong Illumination*[WATT00], aos mais complexos, como *Ambient Occlusion* (AO) [AO07], são exclusivamente implementados em *shaders*, por serem técnicas não presentes no *pipeline* fixo. A Figura 2-3 exibe as diferenças entre um modelo renderizado com e sem *shader* de AO. Observe os detalhes de profundidade no rosto do modelo, possível através de AO implementado em *shader*.

Figura 2-3. Esquerda:modelo sem *shader* de AO. Direita:modelo com *shader* de AO. Fonte: [UNI310].

Apesar de seus benefícios,linguagens de *shading*não suportam muitas das características de programação presentes em C[NVCG10], como recursão, empilhamento de dados e escrita/acesso em endereços arbitrários. Consequentemente, a customização do *pipeline*restringe-se às formas estáticas de acesso à memória. Além disso, uma técnica relacionada a um estágio do *pipeline* pode afetar outro, fazendo-se necessárias alterações em diversas etapas a cada adição de uma técnica. Este é o caso, por exemplo, de técnicas de mapas de sombra[WIDO06], que precisam levar em consideração casos de transparência, sendo técnicas executadas em estágios distintos do *pipeline*. A baixa flexibilidade no acesso à memória dificulta a troca de informação entre esses estágios e como consequência, adaptações não intuitivas precisam ser realizadas para que o resultado seja satisfatório.

Seguindo a trajetória de evolução das placas de vídeo, saindo de um *pipeline* fixo para um programável, é importante observar uma necessidade dos desenvolvedores gráficos por plataformas flexíveis, que ofereçam espaço para customizações de técnicas de CG. Por fim, também é relevante associar tal evolução com o amplo interesse acadêmico na GPU, com o possível uso desta arquitetura para aplicações de propósito geral, conhecido como GPGPU (*General Purpose computation on GPU*), descrito em maiores detalhes na seção 2.2.4.

2.2 Ray Tracing

A técnica de traçado de raios ou *ray tracing*, vem sendo utilizada por mais de três décadas[APPEL68]para a síntese de imagens baseando-se em simulações de fenômenos físicos naturais. No campo da CG, o *ray tracing*tem suas origens em 1968 com o trabalho de Arthur Appel [APPEL68], que utilizou *ray casting*(ver seção 2.2.1) para resolver o problema de teste de visibilidade. Appel é considerado o pai do algoritmo de*ray casting*. Entretanto, curiosamente tal técnica tem suas definições conceituais iniciais no trabalho do pintor Albrecht Dürer, durante o Renascimento, no século XVI [HOFF90]. Enquanto estudava conceitos de projeção e perspectiva, Dürer desenvolveu uma “máquina de perspectiva”, aparato com um funcionamento muito similar ao conceito de *ray casting*, como mostra a Figura 2-4.

Figura 2-4.“Máquina de perspectiva”. Albrecht Dürer, 1525[HOFF90]. A projeção perspectiva era simulada através de uma linha e agulha que marcava os pontos projetados no plano. O caminho percorrido pela linha seria análogo à travessia de um raio no *ray tracing*.

A ideia de *ray tracing* recursivo (ver seção 2.2.1), foi introduzida em 1980 por Turner Whitted [WHI80]. As primeiras imagens sintetizadas por *ray tracing* foram originadas a partir do trabalho de Whitted, como a imagem “*Spheres and Checkerboard*” (Figura 2-5). Whitted foi capaz de gerar imagens de cenas com fenômenos de reflexão, refração e sombras.

Figura 2-5."Spheres and Checkerboard".Turner Whitted, 1980 [WHI80].

2.2.1 Definição

Ray tracing é considerado como um ponto de entrada para o surgimento e utilização de técnicas de iluminação global [JEN96]. Seguindo uma abordagem baseada em fenômenos físicos, ao invés de “desenhar”triângulos na tela, como ocorre em algoritmos de renderização baseados em rasterização, a técnica de traçado de raios ou *ray tracing*[GLAS89] se baseia na emissão de raios pela cena. Tais raios se propagam no ambiente, sendo refletidos/refratados por objetos na cena. Mesmo utilizando os conceitos mais básicos da Física Ótica, *ray tracing* oferece uma imagem final com uma riqueza de detalhes maior quando comparado com cenas

rasterizadas, como é demonstrado naFigura 2-6. Materiais transparentes ou refletivos são representados com maior realismo, mesmo no mais simples *ray tracer*, do que utilizando as mais modernas técnicas de rasterização.

Figura 2-6. Níveis de realismo diferentes de uma cena renderizada com as técnicas de rasterização (esquerda) e *ray tracing* (direita).Fonte: [INT10].

Ao contrário da rasterização, *ray tracing* não depende da transformação dos objetos de toda a cena em polígonos. Portanto, tais objetos podem ser representados de forma mais precisa. A esfera, por exemplo, pode ser considerada como uma entidade algébrica que, a partir de sua equação espacial é possível obter informações suficientes para a renderização, realizando assim menos etapas de conversão contínuo-discreto e gerando menos erros. Esta forma diferenciada de representar os objetos é uma das grandes vantagens do *ray tracing*.

Para gerar imagens com qualidade gráfica superior, a técnica de *ray tracing* demanda um alto poder computacional se comparada com a rasterização, sendo geralmente utilizada apenas para renderizações *off-line* com um foco na qualidade da imagem. Entretanto, com o crescimento da performance tanto das CPUs como das GPUs, nos últimos anos começaram a surgir trabalhos utilizando *ray tracing* também para renderizações em tempo real. O *ray tracing* oferece uma melhor performance que a rasterização em cenas estáticas de alta complexidade [HAVR01], com mais de 100 milhões de polígonos visíveis, onde o algoritmo de rasterização se tornaria proibitivo. O principal argumento se refere à complexidade assintótica do *ray tracing*, que cresce de maneira sub-linear em relação ao número de objetos na cena, enquanto que na rasterização esse crescimento é linear.

Todo *ray tracer* inicia seu processamento emitindo no espaço 3D um conjunto de raios que se originam na posição da câmera virtual. Tais raios são direcionados para cortar o chamado espaço de tela ou "s*creen space*", que neste caso representará a imagem final, como é demonstrado na Figura 2-7. Os raios, por serem o ponto de entrada da renderização do *ray tracing*, são chamados de raios primários. Em um *ray tracer* simples, é emitido um raio

primário para cada *pixel* da imagem. O processo de emissão destes raios primários é chamado de *ray casting*.

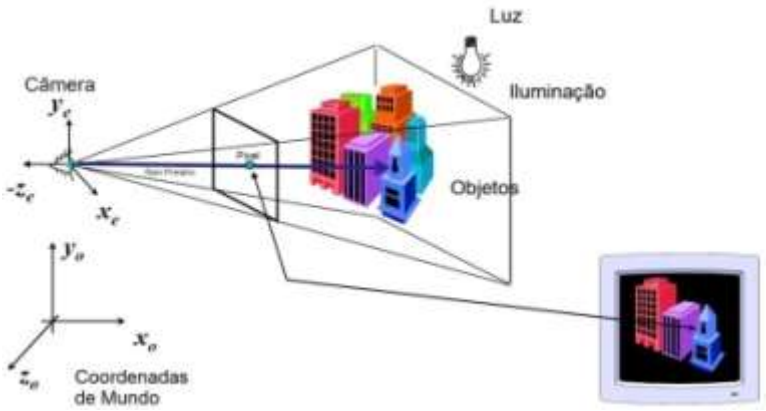


Figura 2-7. Processo de síntese de imagens por *ray tracing*. Fonte (inglês): [WATT00].

Quando um raio intersecta um objeto na cena, novos raios poderão ser emitidos, sendo estes raios refletidos, transmitidos, de sombra ou de iluminação. Tais raios são conhecidos como raios secundários. Os raios refletidos são gerados a partir da lei de reflexão [GLAS89], ilustrada na Figura 2-8.

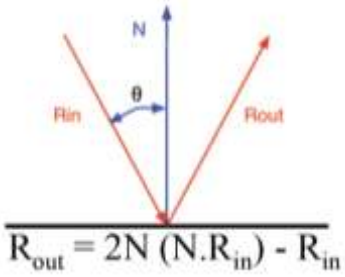


Figura 2-8. Cálculo do raio refletido.

É importante observar que estes casos de raio secundário podem ocorrer fora do campo de visão da câmera. Um exemplo disto é o caso de termos um espelho em frente à câmera, que irá exibir objetos atrás do olho (câmera) virtual. Neste caso, técnicas de *frustum culling*[OGL05], utilizadas em rasterização, necessitam de algumas modificações para funcionar corretamente em *ray tracing*. Isto significa que, ao invés de uma das faces do *frustum* representar o "*screen space*", temos que remover o *frustum* para um formato de regiões que seguem o eixo negativo do vetor de direção da câmera. Assim, objetos fora do campo de visão poderão ser refletidos em objetos visíveis e reflexivos. Uma abordagem utilizada, neste caso, é de, ao invés de definir um *frustum*, utilizar uma esfera ou um elipsóide para realizar a

operação de *culling*. Isto pode melhorar a performance, já que testes de intersecção entre a esfera e outras primitivas quadráticas são um dos testes mais baratos computacionalmente.

Cada raio gerado contribui com uma parcela da cor final do *pixel*. Esta contribuição depende principalmente dos materiais (texturas/cores) dos objetos intersectados e dos ângulos de incidência sobre tais objetos.

Outro fator para a contribuição de cor de um raio é o modelo de iluminação local [WATT00] em cada ponto de intersecção raio-objeto. Observe que, diferentemente do que ocorre em rasterização, no *ray tracing* o processo de *shading* pode ocorrer mais de uma vez por *pixel* para o cálculo final da cor, sendo esta quantidade de ocorrências definida pelo número de raios de iluminação que este *pixel* irá gerar. Entretanto, esta etapa envolve cálculos relativamente simples se comparado com todo o *pipeline* do *ray tracing*, não tendo assim um grande impacto na performance.

2.2.2 Ray Tracingem Processadores com SIMD

Aplicativos multimídia e de CG dependem de intensos cálculos algébricos e matemáticos, envolvendo muitas operações custosas computacionalmente, como cálculos de seno, cosseno e raiz quadrada. Assim, microprocessadores que antes apenas tinham unidades de cálculo em ponto fixo, passaram a adicionar e aprimorar suas unidades de cálculo em ponto flutuante. Muitas dessas operações algébricas são operações vetoriais 2D/3D, resultando em duas ou três instruções semelhantes em sequência, variando apenas os registradores de entrada. Definiu-se então extensões para unidade de cálculo em ponto flutuante, capazes de realizar simultaneamente uma mesma instrução em até quatro pares de registradores diferentes. Conhecidas como extensões SIMD (*Single Instruction, Multiple Data*) [WALD01], tais módulos se disseminaram principalmente na arquitetura x86. Em processadores Intel, componentes SIMD são conhecidos como SSE (*Streaming SIMD Extensions*). A Figura 2-9 demonstra a diferença de fluxo entre uma instrução do tipo SIMD e uma simples do tipo SISD (*Single Instruction, Single Data*).

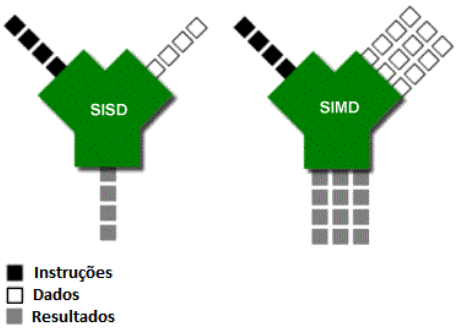


Figura 2-9. Unidade SISD vs. SIMD.

Com a adição destas extensões, alguns tipos de algoritmos obtiveram um ganho de performance significativo, sendo este o caso de técnicas de travessia de estruturas (ver capítulo3) para *ray tracing*. Como raios de *pixels* vizinhos originados da câmera tendem a seguir caminhos semelhantes, a computação realizada durante a travessia de tais raios também será parecida. Seguindo este conceito, Wald et al. [WALD01] demonstraram o uso de SIMD para a realização de tais instruções repetidas, definindo métodos de travessia por pacotes. Através de pacotes, a travessia de raios vizinhos ocorre em paralelo. Com SIMD, cada instrução da travessia é executada por quatro ou mais raios. Caso os raios do pacote comecem a divergir ao seguirem caminhos diferentes, primeiramente os raios que convergem serão processados. Quando tais raios forem completados, a travessia em pacotes segue no caminho dos raios restantes, até o último grupo de raios convergentes seja processado. Assim, através de *ray tracing* com SIMD, uma única *thread* executando em um único processador é capaz de realizar a travessia em mais de um raio de maneira paralela.

O ganho computacional oferecido pela adição de SIMD gerou resultados de *ray tracing* com taxas de renderização de 1 a 3 quadros por segundo em imagens de baixa resolução [WALD01]. Apesar de ser uma taxa aquém em uma ordem de magnitude da considerada como tempo real, atingir esta performance se mostrou no momento como um indicativo da possibilidade de *ray tracing* interativo em arquiteturas futuras.

2.2.3 Ray Tracingem Processadores Multi-Core

O surgimento de processadores com múltiplos núcleos (*multi-core*) é explicado como uma segunda via à resolução do problema de saturação de performance de um único núcleo. O desenvolvimento, seguindo uma arquitetura *single-core* sofre atualmente (devido a

limitações tecnológicas [HEPAT00]), sérias restrições no aumento de sua performance, principalmente no que se diz respeito ao seu aquecimento, devido à necessidade do aumento de sua freqüência de operação. Assim, paralelizar processamento com a adição de mais núcleos de menor freqüência se tornou a alternativa adotada na maioria das arquiteturas em CPU atuais.

Seguindo o conceito *multi-core*, um algoritmo paralelizável passa a ter maior relevância que um puramente serial. Felizmente, *ray tracing*faz parte do conjunto de técnicas altamente paralelizáveis, já que o resultado de cada *pixel* a ser processado é completamente independente de seus vizinhos. Assim, é possível paralelizar ao distribuir de maneira simples os *pixels* a serem processados entre os processadores. Com a granularidade de *pixels* estando na ordem de milhões numa imagem em alta definição (1920 x 1080), mesmo processadores com dezenas de núcleos são capazes de processar *ray tracing* com boa escalabilidade.

Daniel Pohl *et. al*[INT09] demonstraram em 2008 a técnica de *ray tracing* em tempo real (acima de 24 quadros por segundo) e em alta definição com o uso de múltiplos *cores*, portando o jogo Quake Wars[ID09] para um *pipeline* completo de *ray tracing*. Esta pesquisa tinha expectativas de que a arquitetura Larrabee [INT09], da Intel, fosse utilizada como plataforma de *ray tracing* em tempo real em videogames. Entretanto, em 2009 [CNET09] a Intel anunciou que a arquitetura seria modificada e desenvolvida com foco em computação científica de alta performance.

2.2.4 Ray Tracing em GPU

A GPU (*Graphics Processing Unit*) faz parte da família de microprocessadores especializados no processamento gráfico, sendo assim encontrada em videogames, computadores pessoais e estações de trabalho, situada na placa de vídeo ou integrada diretamente à placa-mãe.

As GPUs são diferenciadas da CPU principalmente no aspecto de poder de cálculo, medido em GFLOPS/s (1 bilhão de operações de soma/multiplicação em ponto flutuante por segundo) e demonstrado no gráfico da Figura 2-10. Ademais, oferecem alta vazão no barramento de dados, sendosuperiores às CPUs em ambos os aspectos. Como exemplo, o processador Intel Core i7 Quad Core 3.2GHz, possui picos de 51.2 GFLOPS e 64 GB/s, enquanto que a placa de vídeo GeForce GTX 295 possui picos de 1788 GFLOPS e 223.8 GB/s. É

importante lembrar que em ambas arquiteturas, estes desempenhos foram obtidos em situações ideais para cada tipo de processador. Assim, existem casos em que a performance de uma CPU pode ser superior à de uma GPU. Isto está relacionado principalmente ao grau de paralelismo que a aplicação oferece e seu modelo de acesso à memória.

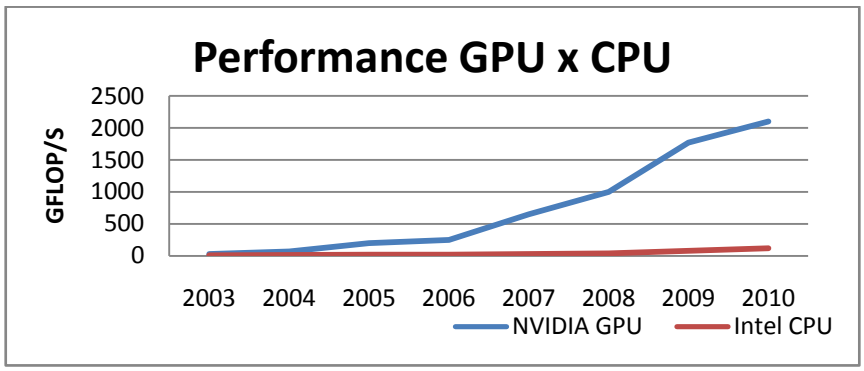


Figura 2-10: Evolução das GPUs e CPUs em termos de GFLOP/s. Fonte: NVIDIA [NVID10].

Originalmente a GPU foi concebida para processamento gráfico, incluindo rasterizaçãoe tarefas de sintetização de imagens. São então processadores orientados para a execução paralela de instruções, otimizados para processar operações escalares do tipo SIMD, variando de 4 a 32 pares de registradores em paralelo para cada unidade SIMD. Comparando com uma CPU comum, a arquitetura de uma GPU é projetada de modo a incluir mais transistores dedicados ao processamento de dados, em detrimento de *caching* de dados e complexidade do fluxo de controle, como ilustrado naFigura 2-11.

Figura 2-11: Arquitetura de CPU vs. GPU. NVIDIA [NVID10].

Com esta arquitetura de cálculo massivamente paralelo, uma GPU consegue lidar melhor que a CPU com problemas que demandam uma grande quantidade de operações sobre dados. Supondo que uma mesma operação é realizada sobre cada elemento dos dados, não existe a necessidade de um fluxo de controle sofisticado. Estas características permitem que latências no acesso à memória sejam amenizadas através da grande vazão de cálculos.

Como descrito na seção 2.1.1, linguagens de *shading*[NVCG10] possibilitaram a adição de efeitos gráficos que não podiam ser obtidos utilizando o *pipeline* gráfico fixo. Apesar das melhorias, a flexibilidade de uma linguagem de *shading* é limitada, restringindo a performance de algoritmos complexos. Entretanto, as primeiras implementações [FOL05] [HORN07] de *ray tracing* em GPU foram através de tais linguagens. Como exemplo, Horn *et al.*[HORN07] implementaram um *whittedray tracer* em GPU utilizando *pixel-shader*, capaz de sintetizar imagens de resolução 1024x1024, a taxas de 12-14 quadros por segundo em cenas de média complexidade, com sombra e *Phong shading*. Entretanto, dois atributos de programação importantes para a eficiência de *ray tracing* é o uso eficiente de pilhas (para as estruturas de aceleração, descritas em detalhes no próximo capítulo) e a possibilidade de recursão (para o próprio *ray tracer*). Tais atributos não estão presentes nativamente em uma linguagem de *shading*, precisando de muitas adaptações que comprometem consideravelmente a performance do algoritmo. Como consequência, os resultados de Horn *et al.* foram inferiores à implementações em CPUs *multi-core* da época.

Tendo em vista as limitações da linguagem de *shading*, novos modelos de programação em GPU foram recentemente propostos [HORN07], sendo abordagens que seguem o conceito de programação de propósito geral em GPU, conhecido como GPGPU. Assim, tal paradigma propõe o uso de uma placa de vídeo não apenas para a implementação de aplicativos gráficos, ampliando a gama de uso para outras áreas da computação, através de subconjuntos de linguagens já conhecidas, como C/C++. CUDA [CUDA10] e OpenCL [OPCL10]são exemplos de arquiteturas voltadas para GPGPU.

O modelo GPGPU, por sua maior flexibilidade, possibilitou implementações eficientes de *ray tracing*[ALS09][PPOV07]em GPU. Com um melhor aproveitamento da arquitetura, tais implementações ultrapassaram em performance as abordagens em CPUs *multi-core*, alcançando um ganho de performance de 10x a 40x [ALS09].

A maioria das técnicas de *ray tracing* implementadas neste trabalho foi implementada em GPU, particularmente em CUDA. Apesar de OpenCL se mostrar como um modelo GPGPU multi-plataforma, atualmente encontra-se nas primeiras versões, com pouca documentação. CUDA já se encontra em sua terceira versão, além de oferecer suporte a *templates* e classes, atributos não presentes em OpenCL na sua versão atual. Mais importante, para o caso das placas da NVIDIA, todo código escrito para OpenCL é executado em cima do *driver* de CUDA, apresentando um possível "*overhead*" em OpenCL, já que seria preciso uma geração de código

OpenCL → CUDA. Diante destes motivos, CUDA foi a arquitetura escolhida neste trabalho para toda a computação de alta performance em GPU. A subseção seguinte descreve brevemente os principais conceitos relacionados à CUDA.

2.2.4.1 CUDA - *Compute Unified Device Architecture*

CUDA[CUDA10] se refere à arquitetura de computação paralela paraGPGPU da NVIDIA, tendo sido lançada em 2006 com os processadores G80, contando atualmente com mais de 110 modelos de placas gráficas compatíveis. Entretanto, o *framework*foi disponibilizado apenas em meados de 2007. É composto por três componentes principais:

- CUDA *Toolkit* – contém o compilador e bibliotecas adicionais como o CUBLAS (*CUDA port of Basic Linear Algebra Subprograms*) e o CUFFT (*CUDA implementation of Fast Fourier Transform*). Além disso, o *Toolkit* contém o CUDA *Profiler*, ferramenta para análise de desempenho e uso de memória do código em GPU. Atualmente encontra-se na versão 3.2, sendo esta a versão utilizada no RT^2;
- CUDA SDK – fornece exemplos de código e bibliotecas necessárias para a compilação de código escrito para CUDA. Além disso, o SDK dispõe de toda a documentação necessária para a programação em CUDA;
- CUDA *Driver* – este componente vem instalado em conjunto com o próprio *driver* da placa de vídeo, necessário para o sistema operacional. Realiza então toda a comunicação em baixo nível entre o sistema operacional e a GPU.

O código é escrito em arquivos com a extensão ".cu". Nestes arquivos são permitidos códigos similares à linguagem C, com algumas extensões, que são necessárias para lidar com detalhes relacionados à forma como o modelo GPGPU opera. Adicionalmente oferece suporte a algumas partes de C++, como *templates* e classes simples.

Alguns conceitos importantes relacionados à CUDA são *threads*,*warps*, *blocks*, *grids*e *kernels*. *Threads* são as unidades de execução paralela em uma GPU. Cada instrução definida pelo programador é executada simultaneamente em um conjunto de 32 *threads*, chamado de *warp*. Centenas de *threads*podem ser agrupadas em *blocks*, onde *threads* pertencentes a um mesmo bloco podem sincronizar sua execução e compartilhar um mesmo espaço de memória (*shared memory*). Um conjunto de blocos representa um *grid*. Um*kernel* consiste no código que é executado por cada *thread*. Cada chamada a um *kernel* precisa especificar uma

configuração contendo o número de blocos em cada *grid*, o número de *threads* em cada bloco, além de parâmetros opcionais. A Figura 2-12 demonstra a organização de *threads*, *grids* e *blocks*.

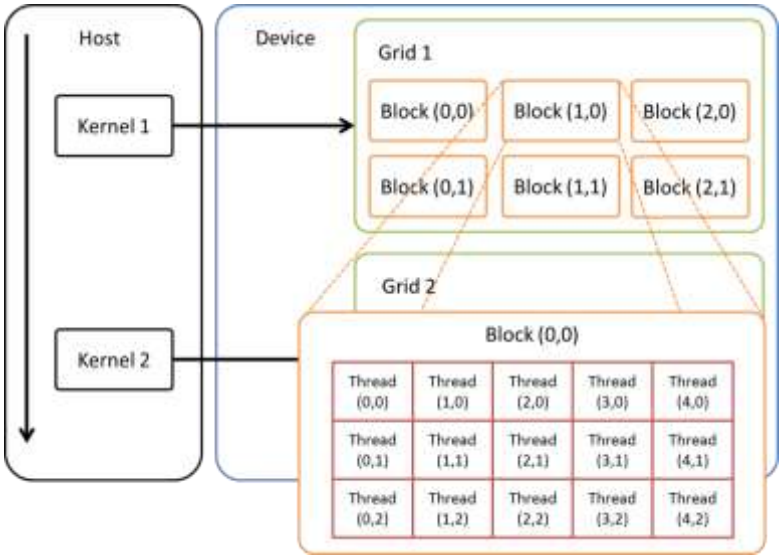


Figura 2-12. Organização de *threads*, *blocks* e *grids* em CUDA.

Em CUDA, algumas palavras-chaves são adicionadas à linguagem C. Como exemplo, modificadores que definem o escopo de uma função ou variável: *__host__*, *__device__* e *__global__*. Os modificadores *__host__* e *__device__* são aplicados tanto a funções como a variáveis, para especificar sua localização, e de que arquitetura de processador podem ser chamadas, *__host__* para CPU e *__device__* para GPU. A palavra-chave *__global__* é utilizada apenas para as declarações de *kernels*, porque o *kernel* é uma função que reside na GPU, mas é sempre chamada a partir da CPU. Outra modificação na linguagem é a invocação dos *kernels*; a chamada de um *kernel* é seguida pela sua configuração entre a sequência de caracteres "<<<" e ">>>", como em:*kernel <<<gridDim, blockDim>>>(params)*.

A hierarquia de memória de CUDA é demonstrada na Figura 2-13. Cada espaço de memória tem um escopo diferenciado. *Threads* podem individualmente ler e gravar em *registers*ou registradores, que assim como a maioria das arquiteturas, é o nível de memória de acesso mais rápido. *Threads* também podem gravar e ler num tipo especial de memória conhecido como *local memory* ou memória local. A memória local é individual para cada *thread*. É geralmente utilizada para armazenar pilhas temporárias e variáveis quando o número disponível de registradores é menor do que o necessário para o cálculo do *kernel*. A memória local reside na memória RAM do dispositivo. Por não ter *cache*, tem um tempo de

acesso alto, com exceção das recentes placas Fermi [CUDA10], que disponibilizam dois níveis de *cache* para a memória local.

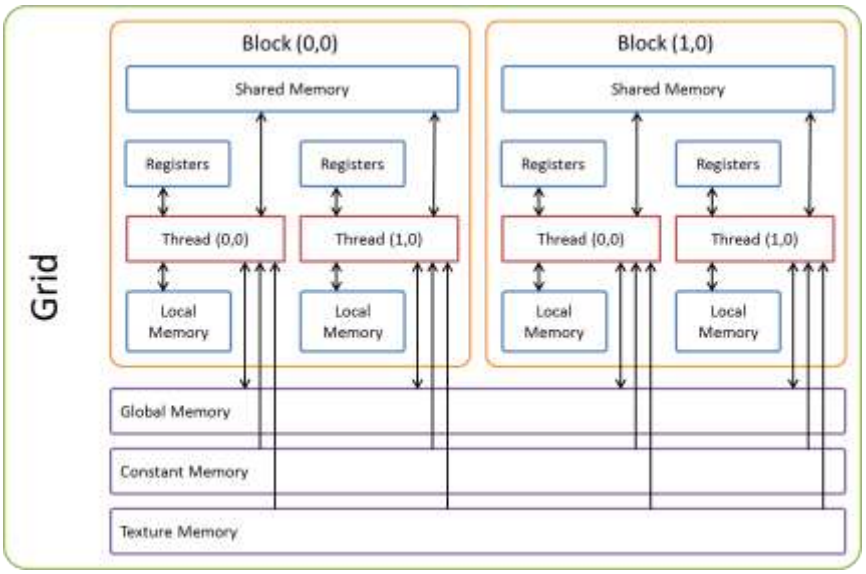


Figura 2-13. Hierarquia de memória de CUDA.

Outro tipo de memória residente na RAM do dispositivo é a *global memory* ou memória global. De fato, é a que representa o espaço da RAM, sendo o maior espaço de memória disponível da GPU, indo de 64 MB a 4 GB (placas Tesla [CUDA10]). Assim, é onde reside a maioria dos dados a serem processados. A memória global tem *cache* automática até dois níveis nas placas Fermi. Em gerações anteriores, é possível realizar *cache* manual através das memórias de texturas. Neste caso, o programador especifica quais *array*stirarão proveito da*cache*de textura, sendo uma *cache* por multiprocessador da GPU. Somente através de texturas é possível ter acesso às unidades de filtragem bilinear em *hardware* da placa, sendo uma vantagem particular das texturas.

Um tipo especial de memória de CUDA é a *shared memory* ou memória compartilhada. Tem esta designação já que *threads* de um mesmo bloco compartilham deste espaço de memória para se comunicar entre si, ou para extraírem informação de maneira eficiente. A palavra-chave __*shared*__ foi introduzida em CUDA para indicar que uma variável será alocada neste espaço de memória. A presença da memória compartilhada é capaz de trazer um ganho de performance elevado para alguns algoritmos que demandam alta taxa de transferências de memória, já que o tempo de acesso (escrita e leitura) da memória compartilhada é próximo [CUDA10] ao de registrador e utilizado em múltiplas *threads* simultaneamente. Entretanto, a quantidade de *shared memory* por bloco é bastante limitada (48 KB no Fermi vs. 16 KB nas placas mais antigas).

Existe um espaço de memória em CUDA para armazenamento de constantes puras ou valores que raramente variam. É o espaço de memória constante (*constant memory*). Todo literal do código é armazenado nesta memória. Num *kernel*, é possível realizar apenas leituras na memória constante. Assim, valores de memória constante podem ser gravados somente através de código de *host*(CPU), durante a transferência de dados da CPU para GPU.

Assim, o modelo de *software* proposto por CUDA define a GPU como um co-processador de dados paralelo, utilizando um diversificado sistema de memória. Para facilitar o desenvolvimento, a NVIDIA oferece três bibliotecas auxiliares: a *Common Runtime Library*, que provê alguns tipos de dados como vetores com duas, três ou quatro dimensões, e algumas funções matemáticas; o *Host Runtime Library*, que provê gerenciamento dos dispositivos e da memória, incluindo funções para alocar e liberar memória, e funções para invocar *kernels*; o *Device Runtime Library*, que pode apenas ser utilizado pelos dispositivos, e que provê funções específicas como sincronização de *threads*.

Este capítulo apresentou as principais técnicas de renderização em tempo real, seguindo uma ordem cronológica da evolução da arquitetura dos pipelines gráficos predominantes no mercado. Este capítulo também apresentou pesquisas relacionadas com técnicas de *ray tracing* em tempo real, implementadas em diferentes tipos de arquiteturas de processadores, relatandoa vantagens e desvantagens de cada arquitetura. Apesar da arquitetura moderna da GPU não oferecer a mesma flexibilidade de programação que a CPU, novas ferramentas e APIs vem surgindo de forma a facilitar a sua programação. Além disso, a eficiência de cálculo superior da GPU moderna se mostra mais adequada para implementações de *ray tracing* em tempo real do que as abordagens iniciais, puramente baseadas em CPU.

3 Estruturas de Aceleração

O alto custo computacional de um *ray tracer* se deve principalmente ao problema de, para cada raio **r** emitido na cena, encontrar a primitiva geométrica **p** intersectada por **r**, de forma que tal intersecção seja a mais próxima e também posterior à origem de **r**. Uma forma simples de resolver tal problema é a realização de testes de intersecção entre **r** e todas as primitivas da cena. Por ser uma busca pelo conjunto total de primitivas, esta abordagem é considerada como a de força-bruta para o traçado de raios. O pseudocódigo é demonstrado no Algoritmo 3-1.

Algoritmo 3-1: Busca de Força Bruta em *Ray Tracing*

```
1: Ray r = (org, dir); {Ponto de Origem + Vetor de direção}
2: Float t; {Distância paramétrica raio-primitiva}
3: Primitive intersected; {Possível primitiva intersectada}
4: for all Primitive p in Scene do
5:    {Testa r com todas as primitivas da cena}
6:    I = Intersection(r,p);
7:    if I ≠ null then
8:       {Encontrou uma intersecção}
9:       Update t and intersected, using best (smaller t) result;
10:   end if
11: end for
12: if intersected ≠ null then
13:    return HIT(intersected,t);
14: else
15:    return MISS;
16: end if
```

Algoritmo 3-1. Pseudocódigo de força-bruta para busca da intersecção raio-primitiva mais próxima.

Por percorrer todas as primitivas, o algoritmo de força-bruta tem a complexidade linear[HAVR01] para o número de primitivas. Como o custo da intersecção raio-primitiva é relativamente alto na maioria das arquiteturas e tipos primitivos, envolvendo dezenas de operações aritméticas em ponto flutuante, tal abordagem se torna proibitiva em *ray tracing* de tempo real para cenas com milhões de primitivas geométricas. Utilizando esta abordagem, mesmo com algumas centenas de primitivas, mais de 95%[WHI80] do tempo de execução do *ray tracing* se deve ao testes de intersecção raio-objeto. Este alto custo também é devido ao problema de busca estar presente em diversas etapas do *ray tracing*. Como exemplo, para simulação de sombras, é preciso gerar raios de sombra e realizar uma busca por primitivas de oclusão. Da mesma forma, os efeitos de reflexão e refração são obtidos através da emissão de

novos raios e, consequentemente, de novas buscas. Assim, qualquer melhoria realizada no algoritmo de busca representa diretamente um ganho de desempenho no próprio *ray tracer*.

Para reduzir consideravelmente o número de testes de intersecção raio-primitiva, tira-se proveito de estruturas de dados que organizam espacialmente ou agrupam de alguma maneira os objetos da cena. Deste modo, ao invés de seguir uma abordagem de busca "cega" compreendendo todos os objetos, apenas aqueles com alguma probabilidade de intersecção com o raio são testados. Com isto, são descartados da busca os objetos garantidos de não intersectar, como primitivas "distantes" ou anteriores à origem do raio, sendo geralmente tais objetos a maior parte do conjunto de busca original. A maioria destas estruturas são adaptações de árvores de busca [WALD09]. Assim, reduzem a complexidade média do problema para sub-linear[HAVR01] e, por aumentarem consideravelmente a performance de um *ray tracer*, são chamadas de Estruturas de Aceleração (EA).

Como as EA impactam diretamente no grau de eficiência de um sistema de *ray tracing*, foi realizado neste trabalho um levantamento das principais estruturas, utilizadas tanto em sistemas em tempo real como em aplicações off-line. Assim, este capítulo é integralmente dedicado à descrição conceitual de tais estruturas. As seções seguem uma ordem cronológica do surgimento de tais algoritmos na literatura pesquisada. Os algoritmos de busca ou travessia para cada estrutura são descritos em maiores detalhes no capítulo 5. Resultados de implementação de tais estruturas são analisados no capítulo6.

3.1 Octree

A *octree*(*Octaltree*) [REV00] é uma árvore de subdivisão espacial, na qual o nó raiz representa o volume da cena, que será subdividido recursivamente. Na *octree*, um nó interior tem exatamente oito filhos, que subdividem o pai no formato de células de dimensões iguais, como mostra a Figura 3-1. Referências às primitivas da cena são armazenadas nos nós folhas em forma de listas, podendo ser vazias. Assim, a função de um nó interior numa *octree* é o de subdividir o espaço enquanto que um nó folha fornece a lista de primitivas a serem intersectadas.

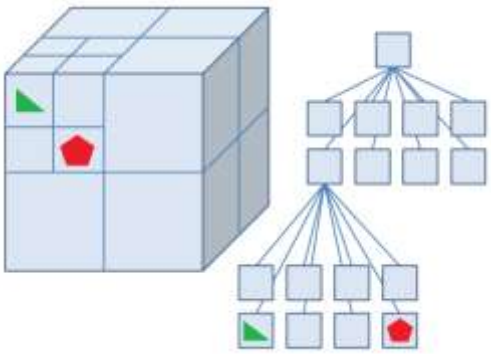


Figura 3-1. Formato de uma *octree*. Nós internos subdividem o espaço, enquanto que nós folhas guardam a lista das primitivas que intersectam.

Para cada subdivisão, os dois planos de corte são alinhados com os eixos espaciais, gerando células em formas de caixas, ou *Axis Aligned Bounding Boxes* (AABBs). Esta subdivisão acontece até o nó filho não conter primitivas, se tornando em um nó-folha, ou se o critério de parada é atingido. Normalmente o critério de parada é alcançar a profundidade máxima da árvore ou um limite de primitivas por nó-folha ser atingido. Apesar da simplicidade quando comparada com outras estruturas, sua construção é ineficiente (ver capítulo 6). Isto se deve ao considerável volume de operações realizadas em cada subdivisão, já que será necessário realizar 8 testes de intersecção AABB-Objeto para cada primitiva presente no nó, de forma a se determinar as listas de primitivas dos nós filhos.

O uso de *octrees* em *ray tracing* foi introduzido por Glassner [GLAS85], que definiu o primeiro método de travessia. Comparado com outros tipos de árvores de subdivisão, a travessia numa *octree* é complexa, por necessitar analisar diversos casos de intersecção entre um raio e um nó interior. Um raio pode atravessar até quatro nós filhos, sendo necessário computar a ordem com que essas intersecções ocorrem, para cada nó interior. Além disso, como os planos de corte são fixos, oito nós folhas serão sempre criados, mesmo quando apenas um desses filhos contenha primitivas. Como consequência, a *octree*frequentemente gera espaços vazios contíguos, que não são unificados.

No contexto de *ray tracing*de primitivas poligonais, a *octree*entrou em segundo plano com o surgimento de novas estruturas que oferecem melhor performance de construção e travessia. Entretanto, pesquisas recentes no campo de *ray tracing* com *voxels*[LAIN10] [LAIN2-10] tiram proveito de uma *octree* compacta, chamada de *Sparse Voxel Octree*(SVO), árvore que armazena apenas células ou nós com elementos ou *voxels* válidos. É importante observar que

apesar da boa performance de travessia com *voxels*, os custos de construção da SVO em termos de tempo e principalmente memória são elevados, limitando-se à criação de cenas estáticas.

3.2 *Grid*Uniforme

O *grid* uniforme (*uniform grid*), assim como a *octree*, é uma estrutura baseada em subdivisão espacial, sendo introduzido no contexto de *ray tracing* por Fujimoto *et al.*[JIMT85] como uma alternativa mais eficiente à*octree*, ao evitar o formato recursivo da busca e construção em árvore. Num *grid* uniforme, todo o espaço é subdivido em AABBs de volumes iguais, chamados células. Entretanto, tais células não são necessariamente cubos, podendo variar suas dimensões a cada eixo. De fato, as dimensões da célula dependem das dimensões de cada eixo do volume total da cena e da resolução de subdivisão do *grid*.

Cada célula armazena uma lista de primitivas que contém. Como geralmente os modelos contém informação apenas de superfície e não de volume, a maior parte das células do *grid* é vazia, sendo este o caso das células localizadas na parte interna do modelo.

No *grid*uniforme, testes de intersecção raio-primitiva são realizados apenas nas primitivas contidas nas células que o raio atravessa, como mostra a Figura 3-2.

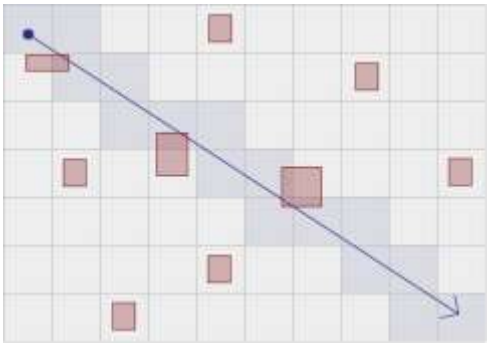


Figura 3-2. Travessia do *grid* uniforme (em 2D). Testes de intersecção ocorrem apenas com primitivas em *voxels* visitados pelo raio.

Por não ser uma estrutura recursiva, sua construção pode ser feita de maneira simples [JIMT85] localizando as células que cada primitiva pertence, sendo assim uma abordagem de complexidade linear em relação ao número de primitivas da cena. Cormen [CORM01] propôs uma implementação utilizando listas ligadas, evitando desperdício de memória. É comum se utilizar vetores ou *arrays* dinâmicos [CORM01] no lugar de listas ligadas, por oferecerem

melhor performance em relação aos acessos de memória. Visando reduzir o uso de memória, Lagae [LAGAE08] propôs duas novas abordagens de construção, o *grid* compacto e o *grid* uniforme com *hashing*. O *grid* compacto unifica todas as listas ou vetores de objetos em uma única, enquanto que o método com *hashing* compacta as células a partir de compressão por deslocamento de linhas, tirando melhor proveito dos espaços com células vazias. A implementação de *grid* uniforme deste trabalho foi baseada no *grid* compacto de Lagae, por ser mais eficiente [LAGAE08] que as outras abordagens.

3.3 *kD-Tree*

Apesar da grande diversidade de estruturas propostas na literatura pesquisada, um tipo comum se refere ao das árvores binárias de partição espacial, ou *Binary Space Partitioning Trees* (BSP-*Trees*) [GLAS89]. Tais árvores subdividem a cena a cada nó interior, através de um único plano de corte, como mostra a Figura 3-3. A subdivisão recursiva ocorre então a partir da escolha de um plano de corte do espaço em todas as ramificações da árvore.

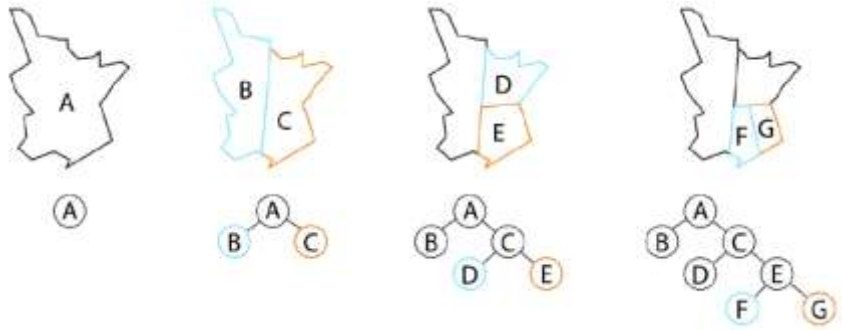


Figura 3-3. Exemplo de uma BSP-*Tree*. Os planos de corte não precisam ser alinhados aos eixos.

Entretanto, no contexto de *ray tracing*, geralmente utiliza-se uma BSP-*Tree* em que os planos de corte são alinhados com os eixos coordenados da cena, com o intuito de reduzir custos computacionais de construção, além de simplificar o algoritmo de travessia. Esta BSP-*Tree*é chamada de *kD-Tree* (***k–D****imensions****T****ree*), sendo introduzida por Bentley [BENT75] no contexto de busca de elementos. De acordo com a literatura pesquisada, a *kD-Tree* foi adaptada para *ray tracing* por Kaplan [KAP85], que definiu um algoritmo de travessia não recursivo. Numa *kD-Tree*, todo nó interno armazena a dimensão e posição do plano de corte que irá dividir a cena em dois sub-espaços, sendo os filhos do nó os representantes desses sub-

espaços. Assim como na *octree*, as folhas da *kD-Tree* representam os menores sub-espaços da árvore e armazenam uma lista dos objetos contidos.

Como os vetores normais dos planos de corte de uma *kD-Tree*são vetores canônicos das coordenadas de mundo, todo sub-espaço da cena é representado por AABBs. Deste modo, todo algoritmo de travessia da árvore se resume à busca de nós a partir de pontos de entrada e testes de intersecção raio-AABB, até encontrar uma intersecção com alguma primitiva ou o raio sair do volume da cena, sem encontrar intersecção. Para cada nó interno, como apenas um dos eixos é escolhido como suporte do plano de corte, o teste de intersecção raio-AABB é novamente simplificado apenas para a dimensão do plano de corte, reduzindo consideravelmente o número de operações se comparado com outras estruturas como a *octree* e BVH (explicada na próxima seção), que necessitam de múltiplos testes de intersecção raio-AABB a cada nó interior visitado.

AFigura 3-4 demonstra o *Stanford Dragon* [STAN10]subdividido em uma *kD-Tree*. As caixas verdes representam as AABBs das folhas. Neste caso, toda a região esverdeada representa a AABB do dragão, que por sua vez representa o nó raiz da árvore. Assim, quanto mais nósuma região tem, mais esverdeada a região é, de forma que a imagem serve como um mapa de densidade da *kD-Tree*.

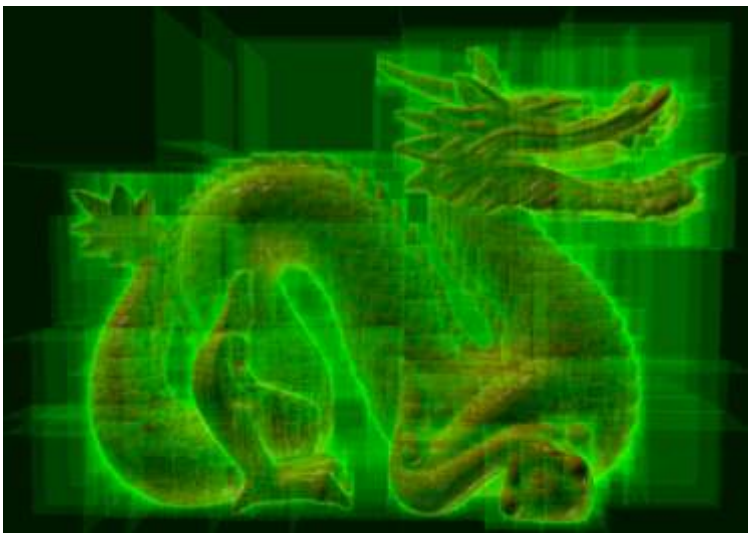


Figura 3-4. Representação da *kD-Tree* (regiões esverdeadas) no modelo do Stanford Dragon [STAN10].

3.4 BVH

A BVH, ou *Bounding Volume Hierarchy*, é uma estrutura de dados de particionamento de objetos que utiliza o conceito de AABBs para realizar a construção da estrutura, bem como a sua travessia. Assim como uma *kD-Tree*, a BVH é uma árvore binária, mas não se caracteriza como subdivisão espacial, já que é baseada em partição do conjunto de objetos. O uso de BVHs em *ray tracing* foi introduzido no trabalho de Kayjia [KAYK86].

Cada partição realizada em uma BVH separa os objetos em dois grupos. Cada nó contém os pontos máximo e mínimo do AABB que o envolve, bem como referências para cada nó filho. Assim, é possível garantir que as primitivas estarão localizadas dentro de caixas que possuem o volume mínimo necessário para contê-las (Figura 3-5). Pode acontecer do volume total ocupado pelas AABBs dos nós filhos ser menor que o da AABB do nó pai. Esta característica indica uma maximização dos espaços vazios. Outra particularidade da BVH se refere à primitiva pertencer a apenas um nó folha, sem gerar novas referências, como acontece em estruturas baseadas em subdivisão espacial. Esta característica beneficia tanto a construção, que não precisa lidar casos de cortes no objeto, como a travessia, já que um raio não repetirá testes de intersecção com uma primitiva, situação que ocorre frequentemente em estruturas de subdivisão espacial.

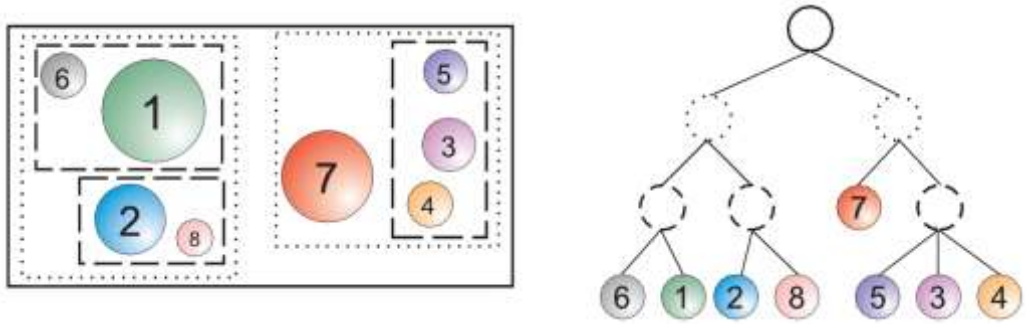


Figura 3-5. Esquerda: hierarquia dos AABBs que representam cada nó da BVH. Direita: *Layout*da árvore gerada [HAVR01].

Devido à característica das primitivas serem agrupadas na menor AABB que as contém, é possível obter uma boa performance na etapa de travessia para os casos de raios que atravessam espaços vazios, já que bastaria continuar a travessia a partir do ponto de saída. Entretanto, como desvantagem, por não ser uma estrutura de subdivisão espacial, a BVH permite que nós irmãos se sobreponham no espaço. Caso um raio atravesse essa sobreposição, o nó mais distante da origem do raio tem que ser empilhado para ser

processado posteriormente. O algoritmo de travessia não pode ser finalizado enquanto não forem desempilhados todos os nós à procura de uma interseção mais próxima, sendo esta uma desvantagem quando comparado com estruturas de subdivisão espacial.

3.5 BIH

A BIH, ou *Bounding Interval Hierarchy*, é uma estrutura híbrida da BVH e *kD-Tree*, que busca otimizar o processo de construção com impacto moderado na performance da travessia. Watcher *et al.*[WCHT06] atribuíram o termo BIH à estrutura. Entretanto, publicações anteriores divulgaram conceitos similares, como a s*kD-Tree*[OOIB], H-*Tree*[HAVR01]e b*kD-Tree*[WOOP06]. Um diferencial do trabalho de Watcher *et al.* é a proposta de um algoritmo eficiente de construção da estrutura. BIH é o termo utilizado neste trabalho, já que tal estrutura é popularmente conhecida por este nome.

Numa BIH, cada nó interno subdivide o espaço em um eixo, da mesma forma que a *kD-Tree*. Entretanto, a subdivisão ocorre em dois planos de corte ao invés de um, demonstrado na Figura 3-6. Além disso, assim como na BVH, a subdivisão da BIH também gera dois conjuntos disjuntos de primitivas, um para cada nó filho. É importante observar que os planos de corte não subdividem as primitivas, de forma que o algoritmo de construção não necessita gerar cópias de referências de primitivas. O centroide da primitiva determina se a mesma pertence ao nó esquerdo ou direito, sem subdividi-la em duas partes. Assim, a lista de primitivas é fixa, determinada antes da construção, o que não acontece com a *kD-Tree*, que necessita criar novas referências para as primitivas repartidas espacialmente.

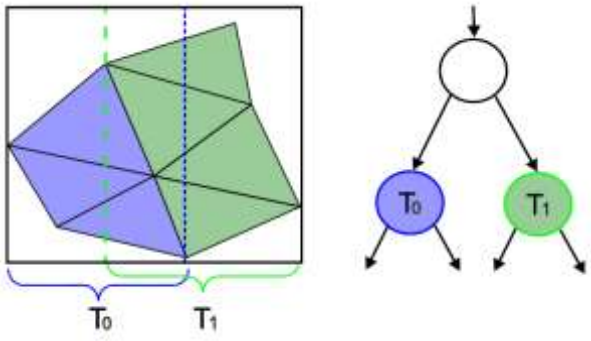


Figura 3-6. Subdivisão da BIH. Linhas tracejadas representam os planos de corte. Primitivas azuis e verdes pertecem ao nó esquerdo e direito, respectivamente.

A travessia da BIH é bastante similar à da *kD-Tree* com pilhas. A busca na árvore ocorre de maneira *top-down*, e caso o raio atravesse dois nós, um é empilhado para ser processado posteriormente. Entretanto, ao contrário da *kD-Tree* e da mesma forma que a BVH, é preciso processar todos os nós empilhados antes de terminar a busca, já que é possível que uma intersecção mais próxima seja encontrada num dos nós empilhados.

A existência de dois planos de corte possibilita o surgimento de espaços vazios entre os filhos de um mesmo nó pai, como mostra a Figura 3-7. Ao contrário da *kD-Tree*, estes espaços vazios não precisam ser instanciados, já que na travessia a BIH é capaz de detectar estas situações a partir de simples checagem se os planos de corte de cada nó filho se entrelaçam ou não e se o raio entra e sai do nó nessa região. Este caso é bastante semelhante ao dos espaços vazios da BVH, descrito na seção anterior.

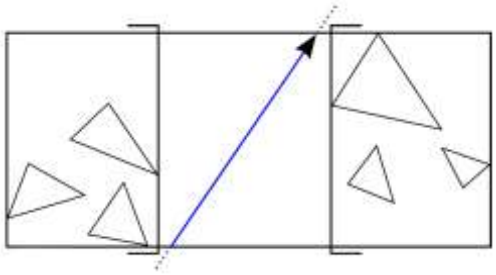


Figura 3-7. Espaço vazio entre dois nós filhos de uma BIH. Isto ocorre quando o ponto de corte referente ao começo do filho direito vem depois do ponto de corte do fim do filho esquerdo. Neste caso, o raio que atravessa este intervalo visita nenhum dos dois nós filhos.

Pela combinação de características da *kD-Tree* e da BVH, a BIH é considerada o meio termo entre partição espacial e de objetos, herdando tanto vantagens como desvantagens das duas abordagens. Por ser uma estrutura definida recentemente, há poucas fontes na literatura pesquisada, sendo uma estrutura ainda pouco estudada. Entretanto, sua pesquisa neste trabalho levou a resultados relevantes, descritos no capítulo6.

3.6 SBG

O SBG, ou*Sparse Box Grid*(*grid* de caixas esparsas) é uma nova estrutura de aceleração, definida neste trabalho. O SBG busca unir a simplicidade de um *grid* uniforme à eficiência da travessia em *kD-Tree*. Seu conceito inicial surgiu a partir de uma busca por melhorias no algoritmo de travessia de *ropes*[ALS09] [PPOV07] da *kD-Tree*, descrito em detalhes no capítulo 5.

O SBG é uma estrutura dupla de subdivisão espacial: um *grid* uniforme combinado com uma coleção de AABBs, como demonstrado na Figura 3-8. Cada célula do *grid* uniforme armazena apenas um índice, que representa uma referência ao AABB que pertence. Não há uma lista de objetos nas células deste*grid*. Os AABBs do conjunto são disjuntos, sendo equivalentes aos AABBs dos nós folhas de uma *kD-Tree*.

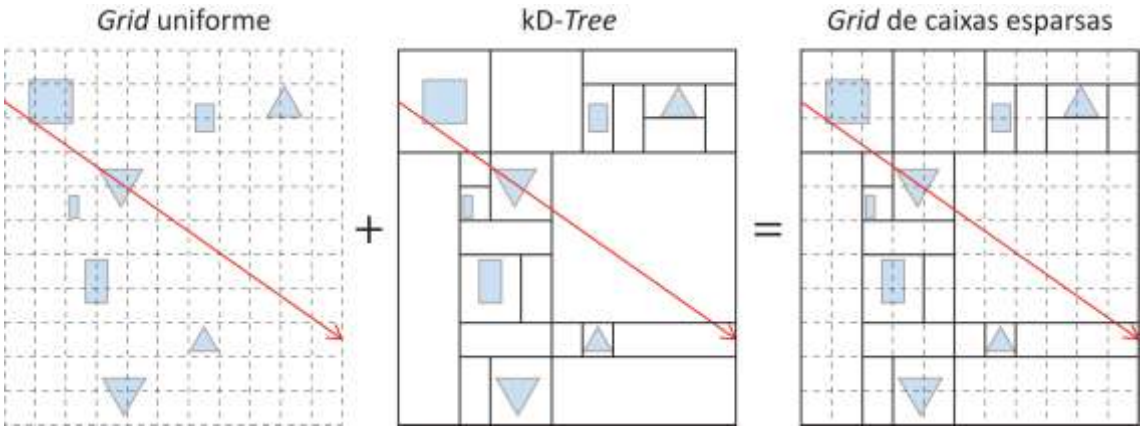


Figura 3-8. SBG resultante da combinação de um *grid* uniforme com uma *kD-Tree*.

É importante observar que, apesar de utilizar um *grid*uniforme de índices, o SBG se comporta como uma estrutura esparsa, já que os AABBs podem ter dimensões variadas nos três eixos, contendo assim múltiplas células do *grid* uniforme. Esta característica é crucial para a eficiência do SBG, já que espaços vazios contíguos podem ser agrupados em um mesmo AABB, sendo visitados de maneira eficiente, enquanto que no *grid* uniforme tradicional, múltiplas células vazias serão visitadas até encontrar uma célula ocupada.

Uma característica importante do SBG se refere à sua compatibilidade com algoritmos de construção de diferentes estruturas de subdivisão espacial por eixos alinhados. Como exemplo, a partir de qualquer algoritmo de construção de uma *kD-Tree* ou *octree*, é possível construir um SBG, já que os nós-folhas das árvores representam os AABBs do SBG. Também é possível construir um SBG a partir de um *grid* uniforme, apesar de demandar passos adicionais para unificar células.

O SBG se comporta como um super-conjunto das estruturas de subdivisão por eixos alinhados, já que é capaz de adicionar alguns tipos de subdivisões que estruturas como a *kD-Tree* não suportam. Como mostra a Figura 3-9. Estruturas em forma de árvore definem uma região de corte por todo eixo, de um extremo ao outro. Subdivisões como as das caixas esparsas da Figura 3-9 fogem desta regra, favorecendo o algoritmo de travessia para alguns casos, como o exemplo da imagem. Entretanto, não foi possível definir neste trabalho um

algoritmo de construção capaz de gerar tais situações para se realizar testes práticos. Assim, os algoritmos de construção utilizados no SBG foram baseados nos da *kD-Tree*. Faz parte de um trabalho futuro a definição de um algoritmo de construção específico para o SBG.

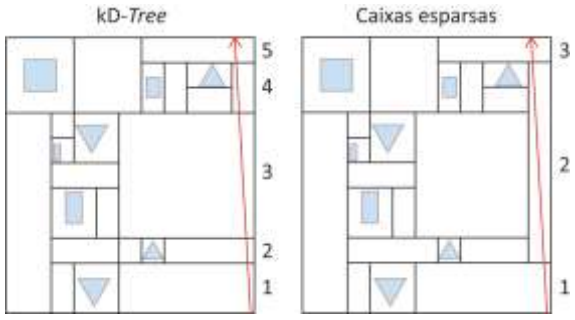


Figura 3-9. Exemplo comparativo de travessia em *kD-Treevs.*caixas esparsas, que não precisam obedecer as regras de subdivisão em árvore.

Este capítulo apresentou as principais estruturas de aceleração para *ray tracing* pesquisadas, utilizadas tanto em sistemas em tempo real como em aplicações off-line. Tais estruturas foram descritas seguindo uma ordem cronológica de surgimento de acordo com a literatura pesquisada. Observou-se (Tabela 3-1) que a maioria dessas estruturas se baseiam em árvores binárias, podendo ser subdivisão binária espacial (*kD-Tree, octree*) ou partição de primitivas, como a BVH. Outras são híbridas entre duas estruturas, sendo o caso da BIH (*kD-Tree* e *BVH*) e do SBG (*kD-Tree* e *grid* uniforme).

Tabela 3-1. Quadro comparativo dos tipos de Estruturas de Aceleração.

	Subdivisão Hierárquica	**Tipo de Partição**
Octree	Sim (Árvore Octal)	Espacial
Grid Uniforme	Não	Espacial
kD-Tree	Sim (Árvore Binária)	Espacial
BVH	Sim (Árvore Binária)	Primitivas
BIH	Sim (Árvore Binária)	Primitivas
SBG	Não	Espacial

4 RT² – *Real-Time Ray Tracer*

Este capítulo apresenta o RT² (*Real-Time Ray Tracer*), biblioteca que oferece uma API (*Application Programming Interface*) para *ray tracing* em tempo real, tirando proveito de processadores modernos. O RT² foi introduzido como Trabalho de Conclusão de Curso (TCC) de graduação do autor [ALSR09], sendo aperfeiçoado nesta pesquisa de mestrado. Muitas das capacidades providas pelo RT² foram contribuições diretas deste trabalho, como a implementação do sub-*pipeline* de *shading* e o suporte às múltiplas estruturas de aceleração, necessário para o estudo comparativo destas. Assim, toda a coleta de resultados (capítulo 6) foi realizada no RT². Este capítulo também apresenta todas as técnicas de *shading*e *anti-aliasing* estudadas e implementadas neste trabalho.

4.1 RT²– Uma Camada de Abstração para *Ray Tracing* em Tempo Real

Normalmente, um aplicativo 3D interativo depende de uma API gráfica para renderizar a cena. Quando se utiliza um *pipeline* gráfico em *hardware*, como hoje ocorre com o uso de OpenGL e Direct3D, é preciso de uma ponte de comunicação entre a camada na qual o programador define seu aplicativo e o algoritmo que a biblioteca gráfica executa. Esta divisão traz a vantagem de desacoplar a camada de aplicação do *pipeline* gráfico, tornando o aplicativo do usuário menos dependente da tecnologia utilizada para renderizar. Além disso, com o surgimento de futuras tecnologias de renderização, o programador terá menor dificuldade em atualizar seu aplicativo para funcionar corretamente em novos dispositivos, já que as alterações necessárias serão limitadas à ponte de comunicação entre o aplicativo e a API 3D.

O RT²foi estabelecido seguindo este conceito em camadas. Tal modelo foi fundamental para a simplicidade na representação do RT², principalmente por ser desenvolvido tanto em CPU como em GPU. Além disso, por ser um *pipeline* gráfico em tempo real 100% programável em *software*, o RT² é capaz de adicionar novas técnicas de CGcom maior grau de liberdade do que em APIs gráficas comuns, como OpenGL e Direct3D.Muitas funcionalidades de taisAPIs são dependentes deimplementações em *hardware*, sendo passíveis de melhorias apenas quando novas placas de vídeo atualizadas para as novas funcionalidades surgem, o que nos dias de

hoje ocorre em um ciclo de seis meses adois anos[CUDA10]. Uma biblioteca como o RT^2, por ser 100% programável, pode ser continuamente atualizada sem o pré-requisito de um novo dispositivo.

O RT^2 oferece suporte a reflexões e sombras seguindo o modelo descrito por Turner Whitted [WHI80], um paradigma simples de *ray tracing*, que utiliza o conceito de recursão de raios, como descrito no capítulo 2. Entretanto, por questões de otimização e limitações de *hardware*, CUDA não suporta chamadas recursivas de funções para a maioria das placas gráficas. Para resolver este problema, o núcleo do RT^2 implementa um algoritmo iterativo adaptado do recursivo padrão: utilizando apenas a instância do raio primário, cada vez que uma intersecção é encontrada, o raio é transladado para a posição de intersecção e tem sua direção modificada para um sentido dependente do tipo de raio a ser gerado, podendo ser um raio de sombra ou um de reflexão. A iteração é finalizadase encontrar um raio de sombra, se o raio refletido não atingir nenhum objeto ou se o raio atingir um nível de reflexão máximo pré-determinado.

A versão atual do RT^2 não suporta refração, pois demandaria uma pilha para tratar os casos de reflexão e refração, causando uma penalidade severa na performance oferecida em CUDA. Entretanto, há o suporte a elementos transparentes sem refração, ou seja, sem desvios na direção da luz, como ocorre em rasterização de elementos semi-transparentes. O suporte total a elementos refratários será adicionado em futuras versões e quando a arquitetura de CUDA evoluir a ponto de não sofrer tanto com o impacto de uma pilha de dados a mais no *kernel*.

É importante salientar que, apesar do RT^2 ser capaz de executar *ray tracing* em tempo real e em resoluções de alta definição(1920 x 1080), o mesmo não supera a performance de uma implementação de rasterização otimizada em *hardware*, principalmente por seu núcleo se basear em *ray tracing*. Entretanto, o RT^{2}disponibiliza de forma simples alguns efeitos gráficos de alta qualidade que são conseguidos em rasterização apenas através de *shaders*complexos ou utilizando um custoso *pipeline* de múltiplos estágios. Para exemplificar, seguem algumas vantagens do RT^2:

- Reflexões de luz obtidas de forma simples e com alta precisão. Em rasterização, apenas com o auxílio de *shaders* e técnicas baseadas em *environment mapping* [WONG06]se torna possível simular (com menor qualidade) reflexões no ambiente. Entretanto, abordagens como *environment*

mapping dificilmente são capazes de capturar inter-reflexões entre objetos, efeito muito comum entre objetos reflexivos, como espelhos e muitos metais. No RT^2 tais efeitos são facilmente obtidos através de emissão de raios secundários, como descrito no capítulo 2.

- Uso de diversas entidades algébricas como esferas, caixas e planos, além de triângulos. Em rasterização, as únicas primitivas permitidas no *pipeline* são vértices, linhas e polígonos, fazendo com que toda entidade algébrica seja transformada em formas facetadas, gerando o efeito "origami" e consequentemente reduzindo o realismo e a qualidade visual da cena. O volume de informação para representar e renderizar o modelo 3D no RT^2em alguns casos também é reduzido. Para o caso da esfera, por exemplo, sua representação utiliza ao invés de algumas centenas ou milhares de pontos 3D dos seus polígonos, apenas quatro valores de ponto flutuante representando a sua equação espacial, sendo três desses valores para o ponto central, mais um para o tamanho do raio da esfera. A texturização também é simplificada, ao não necessitar informar coordenadas de textura em esferas, sendo calculadas a baixo custo computacional durante o próprio *ray tracing*.
- Transformações de primitivas para coordenadas de câmera não são necessárias no RT^2. Em *ray tracing*, testes de visibilidade não se baseiam em projeções e sim em proximidade de cada raio com a cena, independente da câmera. No RT^2, o teste de profundidade por *z-buffer* não ocorre, tendo seu papel substituído pela própria travessia do raio na estrutura de aceleração, que ordena parcialmente os objetos da cena, sendo esta ordenação completamente independente da posição e orientação da câmera, e sim da cena.
- Funcionalidades de câmera, como *zoom* e foco são facilmente definidas no RT^2, por serem parâmetros apenas dos raios primários. Em rasterização, esta definição envolve todo o processo de visibilidade e cálculo de perspectiva, fazendo com que algumas dessas técnicas sejam possíveis apenas através de *shaders*.
- *Shading* eficiente. No RT^2, cálculos de iluminação e sombreamento ocorrem apenas em regiões que afetarão o resultado da imagem final, sendo o custo de computação da iluminação baixo se comparado com todo o *pipeline*. Já em rasterização, o processo de *shading* é geralmente realizado em muitas regiões

que poderão ser descartadas em etapas posteriores como, por exemplo, *shading* em regiões ocluídas por objetos ou sombras, algumas vezes detectadas apenas posteriormente no *pipeline*. O conceito de *deferred shading* [WATT00] reduz parcialmente este descarte, entretanto dificulta a implementação de outras etapas, como *anti-aliasing*.

- Ausência de empilhamento de matrizes de modelo de visão e de perspectiva. Como não há a necessidade de transformar cada primitiva em coordenadas de câmera, também não se faz necessário o empilhamento de matrizes do modelo de visão. Da mesma forma, matrizes de perspectiva também são desnecessárias, já que no RT^2 o modelo de perspectiva a ser utilizado depende apenas da origem e direção dos raios primários. A ausência destas pilhas de matrizes facilita o trabalho do programador ao isentá-lo da necessidade de controle manual de operações de *push* e *pop*[OGL05] das matrizes, como ocorre em rasterização.

4.2 Arquitetura

Um aplicativo interativo síncrono geralmente obedece a um ciclo bem definido de operações. No caso de aplicativos 3D, este ciclo costuma funcionar como mostrado na Figura 4-1. O RT^{2}é uma biblioteca que executa única e completamente a etapa C. Apesar de estar inserido em um ciclo síncrono, por utilizar múltiplos processadores (CPU + múltiplas GPUs), o RT^2 realiza a etapa C de forma paralela, alocando *threads* de controle para CPUs e GPUs.

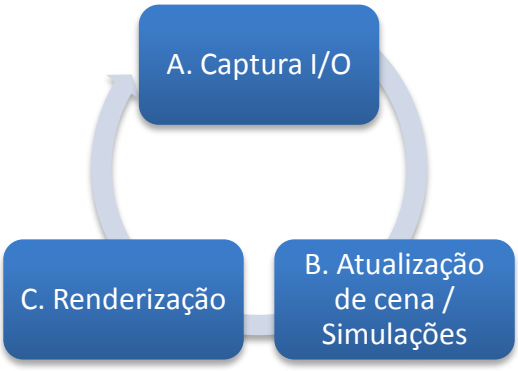


Figura 4-1. Ciclo padrão de uma aplicação 3D.

Com o intuito de ocultar as *threads* de controle, o RT^2 une a comunicação das duas arquiteturas de *hardware* utilizadas (GPU e CPU) em uma única interface de alto nível, a

RT²API, como mostra a Figura 4-2. Como descrito na seção anterior, utilizando a RT²API, o desenvolvedor não precisa ter conhecimento sobre a implementação da biblioteca em baixo nível de CPU ou GPU. Além de ser uma interface orientada a objetos, tal API oferece um sistema de gerenciamento de memória que reduz consideravelmente o trabalho de alocar/atribuir entidades e materiais.

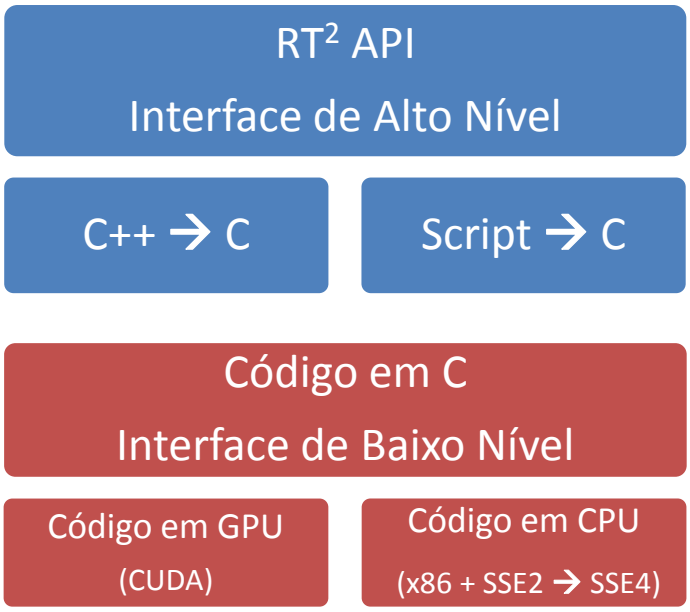


Figura 4-2. Camadas de *software* do RT².

A RT² API se divide em dois módulos, um de programação em C++ e outro através do uso de arquivos de *script*. O módulo de C++ oferece ao programador todas as funcionalidades da biblioteca. No módulo de *scripts*, cenas, materiais, imagens e modelos podem ser carregados em arquivos externos, sem necessidade de compilação. Na presente versão do RT² o módulo de *scripts* não oferece suporte à programação, sendo uma meta futura.

A primeira camada de baixo nível é escrita em C, sendo responsável por repassar todas as operações de alto nível para um nível de decisão sobre o uso em CPU e GPU. Como exemplo, a operação de reconstrução da cena é realizada em CPU e o *ray tracing* em si é realizado em GPU. Esta camada de controle escalona tais operações para serem realizadas em suas respectivas arquiteturas, podendo ocorrer simultaneamente duas operações, como é o caso da reconstrução em CPU e do *ray tracing* em GPU. Assim, toda a geração de múltiplas *threads* ou instâncias de cena é definida e controlada nessa camada.

Na camada inferior existem dois módulos, um para a GPU e outro para a CPU. O módulo para a GPU é escrito em C com as extensões de CUDA, enquanto que o em CPU é escrito em C

e também em *assembly* x86 com extensões SSE4. É nesta camada que a computação de alta performance (construção, travessia e *shading*) ocorre.

A Figura 4-3 mostra o pipeline do RT^2. Cada modelo 3D é encapsulado em um ator, que pode realizar transformações afins, incluindo rotação e escala. Caso a cena seja dinâmica de corpos rígidos, o RT^2 cria uma hierarquia de estruturas de aceleração, na qual a primeira estrutura é responsável pela cena enquanto que as estruturas do nível inferior são para os corpos rígidos.

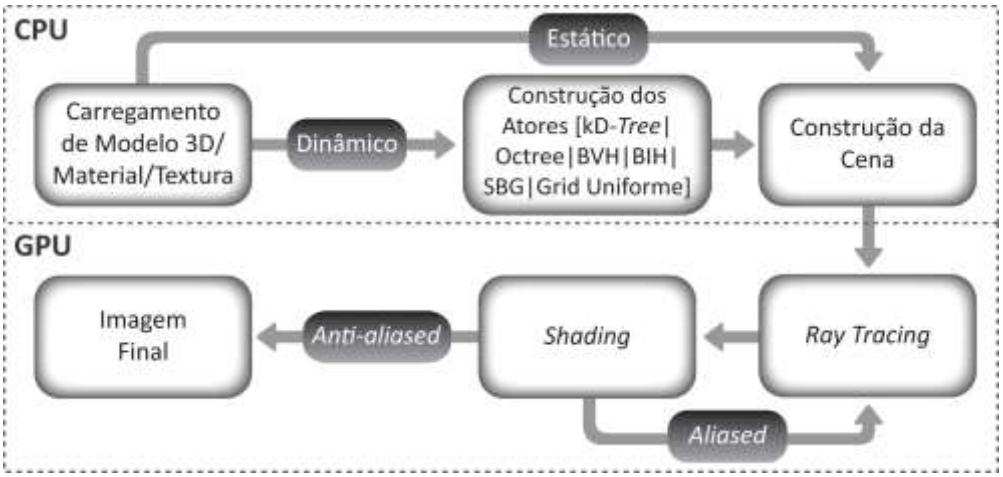


Figura 4-3. *Pipeline* do RT^2.

4.3 Implementação em CUDA

Apesar do alto custo computacional, a técnica de *ray tracing*é massivamente paralelizável. Cada *pixel* da imagem origina um raio primário, que por sua vez gera raios secundários computados de maneira independente dos *pixels* vizinhos. Assim, o processamento de um *ray tracer* pode ser igualmente sub-dividido entre os *pixels* da imagem. Seguindo esta ideia, o RT^{2}distribui o trabalho de cada *pixel* a diferentes *threads* do *kernel* em GPU.

Como CUDA é uma arquitetura SIMT (*Single Instruction*,*Multiple-Thread*)[CUDA10], para alcançar boa performance é importante que *threads*vizinhas (em um mesmo *warp* ou num mesmo bloco) sigam sempre que possível a mesma sequência de instruções, de forma que a instrução seja executada simultaneamente em todas as *threads*. Assim, no RT^2*pixels* próximos são executados em conjunto, favorecendo a execução paralela pelo fato de raios provenientes de *pixels* próximos terem origens iguais e direções parecidas, com alta probabilidade de

realizarem a mesma travessia na cena, intersectando as mesmas primitivas e, consequentemente, executando o mesmo conjunto de instruções, além de otimizarem a *cache* ao reduzir a carga de leitura de dados.

Raios de caminhos parecidos como os originados pela câmera são conhecidos como raios coerentes, ou *coherent rays*[WALD01]. Implementações em CPU tiram proveito de raios coerentes ao utilizar algoritmos de travessia em pacotes de raios, agrupando a computação da busca em conjuntos de instruções SIMD, como descrito no capítulo 2. Já a arquitetura SIMT de CUDA não exige que o algoritmo de busca seja alterado, e assim não tem um possível "*overhead*" quando comparado com o modelo SIMD. Em CUDA, para tirar algum proveito de raios em pacotes, basta garantir que *threads* vizinhas computem raios coerentes.

Ao distribuir o trabalho de um *pixel* por *thread*, é possível utilizar apenas uma chamada de *kernel* para todo o *ray tracing* em GPU. Neste modelo, cada*block* de CUDA processa uma sub-região da imagem por vez. Estas sub-regiões são comumente chamadas de *tiles*, que são ilustradas naFigura 4-4. Esta abordagem oferece menor sobrecarga (*overhead*) que implementações com várias chamadas de *kernel*, já que no último caso, para cada mudança de estágio no *pipeline*, é necessário armazenar na memória global todo o estado do processamento a ser repassado para a próxima etapa.

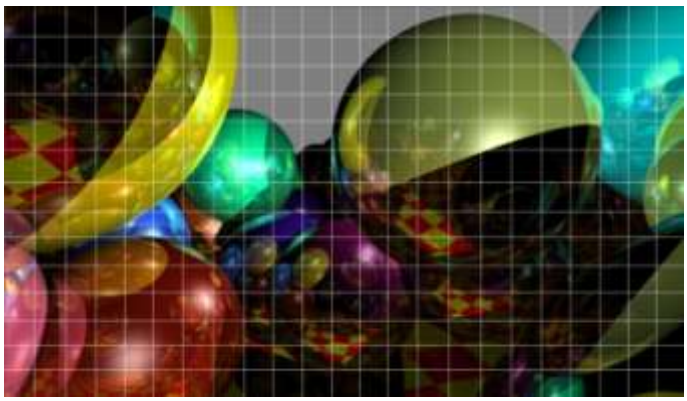


Figura 4-4. Distribuição de trabalho em *tiles*. Cada *tile*é processado em um mesmo bloco ou multiprocessador.

O modelo de um único *kernel* com múltiplos blocos foi implementado e inicialmente utilizado no RT^2. Entretanto, esta abordagem oferece um baixo aproveitamento [AILA09] da GPU para cenas com muitos objetos reflexivos, que geram muitos raios secundários com pouca coerência, chamados de raios divergentes. Este problema é comum em CPU e agravado na GPU, por reduzir o grau de paralelismo da execução devido às múltiplas instruções divergentes, com acessos arbitrários à memória. Além disso, o maior agravante observado se

refere a uma limitação no modelo de blocos da arquitetura. Em CUDA, um bloco termina sua execução apenas quando todas suas *threads* finalizarem suas atividades [CUDA10]. Isto se torna problemático nos casos de apenas uma pequena parte do bloco gerar raios secundários, já que as *threads* com raios primários que terminaram sua execução ficarão em modo de espera, liberando recursos para outras *threads* apenas quando todo o processamento do bloco for finalizado. Este problema não ocorre em CPU, já que normalmente um novo *pixel* é processado assim que o anterior for finalizado.

Diante desses problemas, Aila *et al.*[AILA09] propuseram o conceito de *threads* persistentes. Nesta técnica, ao invés do bloco, cada *warp* (conjunto de 32 *threads* executadas simultaneamente em um mesmo multiprocessador) é responsável por alocar as execuções de suas *threads*. Para isto, *warps* alocam uma memória intermediária ou *pool* de *tiles* a serem executados. Cada *pool*é produzido a partir de uma única variável alocada em memória global. Consumindo concorrentemente esta variável (através de operações atômicas na memória global), *warps*ficam continuamente ocupados, aumentando consideravelmente o aproveitamento em GPU [AILA09].

O RT^{2}em sua versão atual tira proveito da técnica de *threads* persistentes. Entretanto, foi utilizado um sistema de *pools*diferente. Ao invés de instanciar um *pool* para cada conjunto de 32 *threads* como ocorre em *threads* persistentes, o RT^{2}cria um *pool* em memória compartilhada para cada multiprocessador, sendo produzido pela mesma variável global original. Entretanto, w*arps* de um mesmo multiprocessador consomem esse novo *pool* na memória compartilhada. Como a execução de operações atômicas na memória compartilhada é mais eficiente que em memória global [CUDA10],o impacto da sobrecarga das *threads* persistentes é minimizado [ALS11].

4.3.1 Múltiplas GPUs

No RT^2, é possível utilizar múltiplas placas gráficas para atingir um maior nível de performance. O único requisito é que as placas utilizadas tenham a mesma versão de dispositivo CUDA ou *capability*[CUDA10].

É importante observar que cada dispositivo tem o seu próprio espaço de memória, sem intercomunicação direta, que ocorre apenas através da CPU. Então, dependendo do tipo de computação, pode ser necessário criar dados redundantes em cada dispositivo. Este caso se

mostra particularmente difícil quando precisa-se unir os resultados parciais de cada dispositivo em um único espaço de memória, sendo geralmente a memória primária utilizada pela CPU. Infelizmente, o algoritmo de *ray tracing* se encaixa neste perfil. Todo o dado da cena precisa ser replicado para todas as placas gráficas. Entretanto, apenas o resultado da computação (*array* de *pixels*) precisa ser transferido para a memória principal, de forma a ser unificado e repassado para a placa de vídeo principal, que exibe a imagem na tela.

O paralelismo no nível de multi-GPU ocorre com a subdivisão da tela em linha de *tiles* que seguem um padrão intercalado, como mostra a Figura 4-5. Apesar de uma subdivisão mais simples, por linhas puras, ter oferecido melhor balanceamento de carga entre as GPUs, resultou em pior performance quando comparada à subdivisão em *tiles*, caso explicado em mais detalhes no capítulo 6.

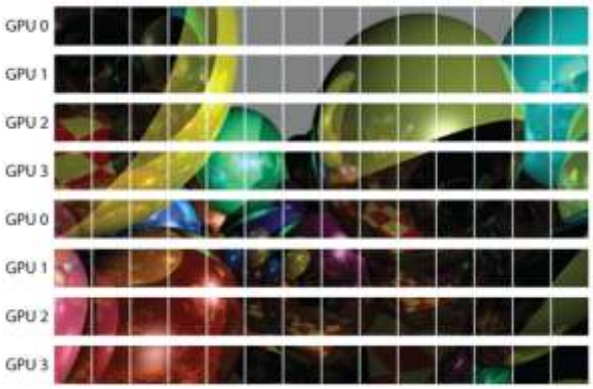


Figura 4-5. Distribuição do processamento entre múltiplas GPUs. Exemplo utilizando 4 GPUs.

4.3.2 Espaços de Memória

Apesar da técnica de *ray tracing* apresentar um alto grau de paralelismo, demanda um considerável volume de acessos à memória principal e como consequência, depende de largura de banda de memória. No caso do RT^2,as *threads*de CUDA realizam com frequência acessos à memória global ao buscar informações da cena. Se realizados aleatoriamente ou de maneira espalhada em relação ao endereçamento de memória, esses acessos podem gerar resultados aquém do esperado, principalmente no contexto de GPUs, que são mais sensíveis à leitura e escrita em endereços arbitrários quando comparados às CPUs, que normalmente dispõem de um sistema de *cache* mais sofisticado.

Com o intuito de reduzir o número de acessos à memória global, o RT^2 organiza a cena 3D na memória principal de forma que primitivas geométricas espacialmente próximas estejam alocadas em blocos de memória vizinhos. Desta maneira, ao realizar a travessia na cena, um raio provavelmente irá intersectar grupos de objetos vizinhos, que seguem o princípio da localidade[BRDL02]tanto em relação ao ambiente virtual 3D como à organização da memória global, aumentando as taxas de acerto (*hit*) na *cache* para tais objetos.

No RT^2, as estruturas de aceleração também são definidas de maneira a aumentar as chances de *hit* na *cache* e reduzir a demanda por banda de memória. Como exemplo, os nós da árvore de uma *kD-Tree*são ordenados seguindo o formato de busca em profundidade, demonstrado na Figura 4-6. O formato de van Emde Boas [BRDL02]também foi implementado;apesar de favorecer a travessia ao garantir acertos na *cache* ao descer para um nó de nível ímpar da árvore, o formato de van Emde Boas utiliza o dobro de memória quando comparado ao de busca em profundidade, por requisitar ponteiros adicionais para outros nós, enquanto que no formato de busca em profundidade o filho esquerdo se encontra exatamente na posição posterior do nó pai, descartando assim a necessidade de armazenar o seu endereçamento de memória.

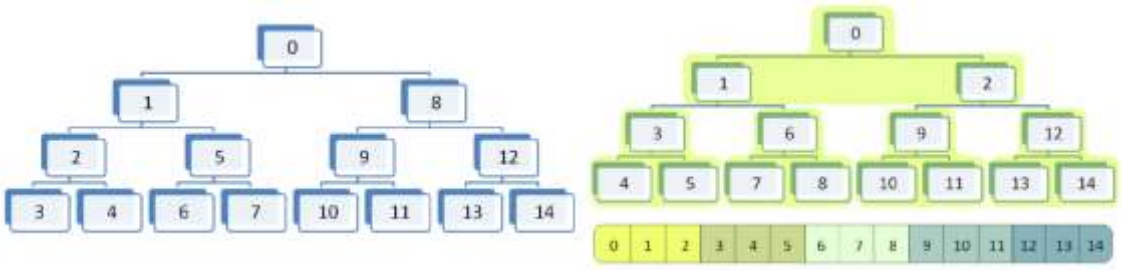


Figura 4-6. Esquerda: formato de busca em profundidade. Direita: formato de van Emde Boas.

Um problema encontrado no desenvolvimento da camada em CUDA para *ray tracing* se refere à demanda de registradores. Quando o compilador de CUDA não é capaz de reduzir o número de variáveis em registradores, o excesso dessas variáveis é alocado no espaço de memória local, situação conhecida como *register spilling*[CUDA10]. O custo de acesso deste tipo de memória é de cerca de 600 ciclos de *clock* a mais que o de registrador.

No RT^2, com o intuito de amenizar o efeito de *register spilling*, o espaço de memória compartilhada é utilizado para armazenar variáveis de cálculo do *ray tracing*, já que o custo de acesso à memória compartilhada é cerca de 4 ciclos de *clock* para acessos sem conflitos de banco[CUDA10]. Assim, o raio é armazenado na memória compartilhada, já que é utilizado em

todas as etapas do *ray tracing*, amenizando em todo código a necessidade por registradores. Outra vantagem de se armazenar a informação do raio na memória compartilhada se refere ao suporte de indexação implícita neste espaço de memória, uma característica importante em testes de travessia. Como em tempo de compilação não é possível saber qual eixo será utilizado em um teste de travessia, é preciso acessar o eixo de um vetor através de acesso por índices, indo do valor 0 (eixo x) ao 2 (eixo z). Em CUDA, variáveis acessadas por indexação implícita são armazenadas automaticamente na ineficiente memória local, já que registradores não suportam indexação. Ao utilizar memória compartilhada, é possível contornar este problema, mantendo a indexação ao mesmo tempo que realizando operações eficientes de acesso à memória. É importante observar que no caso das placas Fermi [CUDA10] o espaço de memória local dispõe de um nível de *cache*, reduzindo o impacto deste espaço de memória na performance, mas não alcançando a performance da memória compartilhada [CUDA10].

O espaço de memória constante de CUDA é usado no RT^2 para armazenar a maioria dos parâmetros fixos, como configurações da câmera virtual e limite máximo da profundidade do *ray tracing*. É importante observar que parâmetros de *kernel* são automaticamente armazenados no espaço de memória compartilhada. Como esta memória está sendo dedicada para dados relacionados à computação dos raios, tais parâmetros são manualmente colocados no espaço de memória constante. Como este tipo de memória é rápido por ter *cache*, esta alteração não adicionou penalidades na performance quando comparado com parâmetros em memória compartilhada.

4.3.3 Anti-Aliasing e Remoção de Serrilhado

Aliasing é o fenômeno que provoca o surgimento de artefatos na imagem quando a taxa de amostragem é muito baixa para capturar as frequências altas de um sinal. As frequências altas, ao invés de desaparecerem, surgem em locais indesejados na imagem [GORM08].

Exemplos do efeito de *aliasing* podem ser vistos na Figura 4-7. A imagem original (esquerda) é formada por padrões de linha que possuem alta frequência. A imagem amostrada (direita) apresenta um padrão de “batimento”, por ter sido amostrada a uma taxa inferior à de Nyquist (que corresponde ao dobro da maior frequência presente no espectro do sinal).

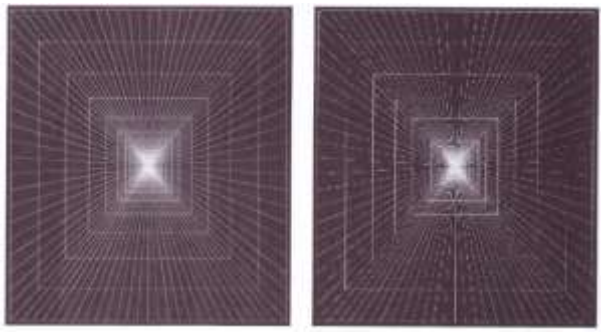


Figura 4-7. Efeito de *aliasing* na imagem da direita, ocasionado por sub-amostragem em taxa inferior à de Nyquist.

O efeito de serrilhado é popularmente designado como outro tipo de *aliasing*, ocorrendo com frequência em imagens sintetizadas por computador, quando linhas inclinadas aparecem em forma da silhueta de uma lâmina de serrote, como mostra a Figura 4-8. Em cenas 3D poligonais é muito comum a presença de serrilhado, já que as arestas dos polígonos têm a forma de linhas.

Figura 4-8. Efeito serrilhado.

No RT^2, esses dois tipos de *aliasing* são tratados de maneira diferente. O caso de *aliasing* por sub-amostragem ocorre no mapeamento de texturas nas superfícies dos objetos. Um problema comum neste tipo de mapeamento é o processo de obtenção do *pixel* para cada coordenada 3D do objeto virtual. A forma mais básica de se obter essa informação é utilizar a técnica de *nearest neighbour*[PBRT07], na qual o *pixel* vizinho mais próximo da coordenada (u,v) da textura é utilizada. Entretanto, por ser geralmente uma sub-amostragem de frequências altas, gera um efeito "pixelado" incômodo ao olho humano. Para amenizar este efeito, o RT^{2}realiza a filtragem bilinear [PBRT07], que suaviza a imagem através de uma interpolação bilinear entre 4 *pixels* vizinhos da imagem. Esta filtragem, apesar de simples, reduz significativamente o efeito "pixelado", como mostra a Figura 4-9.

Figura 4-9. Filtragem de textura no RT². Esquerda: textura mapeada sem filtragem (*nearest neighbour*). Direita: textura mapeada com filtragem bilinear.

Entretanto, o filtro bilinear é eficiente apenas com imagens até 50% reduzidas ou 50% aumentadas em relação ao espaço de tela, mantendo a percepção de *aliasing*nos outros casos que ocorrem com frequência em simulações 3D.Geralmente utiliza-se o método de *mip-mapping* como uma solução mais abrangente e com baixo custo de performance. Infelizmente, *mip-mapping* é uma técnica que o RT²atualmente não suporta, sendo um provável trabalho futuro.

Para reduzir os casos de serrilhado, o RT²implementa duas técnicas: *supersampling*uniforme [GLAS89] e *supersampling* adaptativo baseado em bordas (SSABB)[JMMX10]. A última é contribuição direta deste trabalho, descrito em mais detalhes na subseção seguinte.

No contexto de *ray tracing*, a técnica de *supersampling* uniforme (Figura 4-10) se baseia em aumentar o número de raios gerados por *pixel*, para toda a tela, com tais raios uniformemente distribuídos na tela. Quando se utiliza uma amostra ou raio, apenas um material de uma primitiva intersectada é considerado no processo de *shading*. Como no *supersampling*, raios amostrados podem intersectar objetos diferentes, e a técnica possibilita que um único *pixel* tenha sua cor resultante a partir da contribuição de múltiplos objetos ou materiais, criando uma suavização da coloração em regiões de silhueta, onde ocorre a transição da visualização de um objeto para outro.

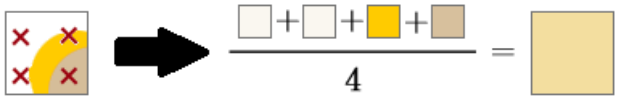


Figura 4-10. Exemplo da técnica de *supersampling* uniforme, utilizando 4 amostras por *pixel*.

Apesar de o *supersampling* uniforme reduzir consideravelmente o efeito serrilhado, o impacto na performance do *ray tracing* é linear para o número de amostras por *pixel*, já que o *supersampling* é realizado em toda tela.

4.3.3.1 Supersampling Adaptativo Baseado em Bordas

O efeito serrilhado se apresenta principalmente nas regiões com alta variação de intensidade, como nas texturas sem filtragem, além de silhuetas e bordas, já que nestes pontos ocorre uma passagem brusca, sem transição, de um objeto para outro. A partir desta observação, desenvolveu-se a técnica de *SuperSampling*Adaptativo Baseado em Bordas (SSABB), que gera novos raios ou amostras apenas em *pixels* de borda, com maior probabilidade de conter o efeito serrilhado.

O SSABB consiste em um processo formado por três etapas: traçado dos raios iniciais, com geração da imagem base; seleção das áreas da imagem que apresentam os artefatos indesejáveis; e, em seguida, o tratamento das regiões problemáticas.

Para implementar as três etapas do SSABB, foi preciso adicionar no RT^2 novas funções de *kernel*. Entretanto, o *kernel* original se manteve, contendo a primeira etapa. Assim, o novo *loop* de execução em CUDA é formado pelos seguintes *kernels*:

- `traceKernel`;
- `toGrayScale`;
- `traceKernelSecondarySamples`.

O primeiro *kernel*, `traceKernel`, gera uma imagem base (realiza o primeiro *ray tracing*) sem considerar algoritmos de *anti-aliasing*. Ou seja, artefatos estão presentes nessa imagem intermediária.

O segundo *kernel*, `toGrayScale`, é responsável por transformar a informação de cor gerada pelo primeiro *kernel* em um valor em escala de cinza que será posteriormente utilizado para avaliar se o *pixel* em questão necessita de mais processamento, de forma a evitar o efeito de *aliasing*.

O terceiro *kernel*, `traceKernelSecondarySamples`, baseia-se inteiramente na versão original do *kernel* de *ray tracing*. A diferença encontra-se na checagem prévia da necessidade de *supersampling* do *pixel*. Antes de alguma técnica de *anti-aliasing* ser aplicada ao *pixel*, é realizada uma chamada à função que executa o filtro de Sobel [GONZ08] sobre um

único *pixel* da imagem, sendo um filtro espacial para aguçamento de imagem. É um filtro diferencial discreto cujo objetivo é destacar transições em intensidade. A Equação 1 apresenta os operadores de Sobel. Como a soma dos coeficientes é zero, há uma resposta nula em áreas de intensidade constante.

$$\partial_x i(x,y) = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix} [1 \quad 2 \quad 1]$$

$$\partial_y i(x,y) = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} [-1 \quad 0 \quad 1] \tag{1}$$

Através da utilização do filtro de Sobel, é possível detectar regiões na imagem com alta variação de intensidade (valor de gradiente elevado). Conforme mostrado naFigura 4-11, as partes em branco da imagem da direita representam as regiões de maiores frequências da imagem original (esquerda).

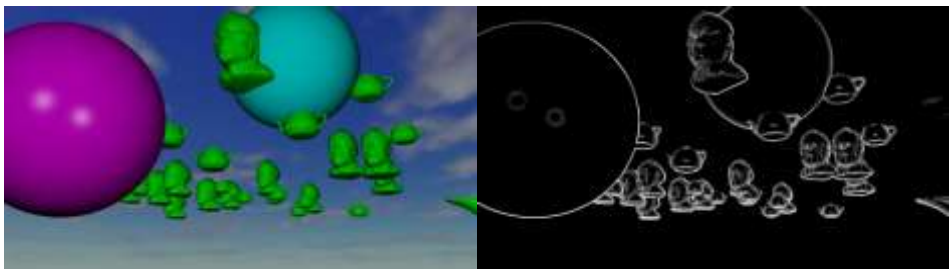


Figura 4-11. Resultado da aplicação do filtro de Sobel.

O cálculo realizado leva em consideração os valores dos gradientes em x (*gx*) e em y (*gy*) de cada *pixel* da imagem, obtidos a partir da aplicação dos operadores de Sobel. De posse dos gradientes, o valor final que representa a frequência do *pixel* é calculado a partir da Equação 2, de soma quadrada,limitando-se o valor de $pixel_{freq}$ ao intervalo normalizado [0,1].

$$pixel_{freq} = \sqrt{gx^2 + gy^2} \tag{2}$$

A definição do operador de Sobel permite implementação relativamente simples, uma vez que apenas oito pontos ao redor do ponto central são necessários para computar o resultado. É importante ressaltar que é também possível aplicar o filtro de Sobel em cada um dos canais de cor gerados pelo primeiro *kernel*, e só a partir daí extrair a informação de necessidade de mais processamento. Neste trabalho, optou-se por trabalhar usando escala de cinza como forma de reduzir o processamento necessário.

Com a filtragem realizada, é possível avaliar se o *pixel* apresenta real necessidade de ser pós-processado. Em caso positivo, dá-se continuidade à execução do código do *kernel*, e a operação de *supersampling* será executada de acordo com o limiar definido na aplicação. Em caso negativo, a execução da *thread* referente ao *pixel*é interrompida e a cor que foi calculada na execução do primeiro *kernel* não é alterada, já que pode ser considerada a cor final.

Por oferecer melhor performance (mais detalhes no capítulo 6) que o *supersampling* uniforme, o SSABB se tornou a técnica de *supersampling* padrão na versão atual do RT^2.

Este capítulo apresentou o RT^2 (*Real-Time Ray Tracer*), biblioteca que oferece uma API (*Application Programming Interface*) para *ray tracing* em tempo real, tirando proveito de processadores modernos. Muitas das capacidades providas pelo RT^2 foram contribuições diretas deste trabalho, como a implementação do sub-*pipeline* de *shading* e o suporte às múltiplas estruturas de aceleração, necessário para o estudo comparativo destas, descrito nos capítulos 5 e 6.

5 Algoritmos de travessiaem GPU

Este capítuloapresenta os algoritmos de travessia das estruturas de aceleração introduzidas nocapítulo3. Cada estrutura dispõe de um ou mais algoritmos de travessia.Astécnicas descritas neste capítulo foram implementadas em GPU como parte do RT^2 (ver capítulo 4). Estes algoritmos divergem em termos técnicos, como por exemplo, no número de acessos à memória, uso de registradores ou tamanho de pilha. Como estas variações impactam diretamente nodesempenho de cada travessia, foi preciso adaptarcada algoritmo para a arquitetura de CUDA, de forma a se obter um melhor aproveitamento do *hardware* utilizado para *ray tracing* no RT^2. Assim, este capítulo também apresenta alguns detalhes de implementação em GPU destes algoritmos. O Apêndice A contém o pseudocódigo de cada algoritmo implementado. Resultados destas implementações e análise comparativa são apresentados no capítulo 6.

5.1 Octree

O algoritmo de travessia em *octree* implementado neste trabalho é baseado na técnica de travessia *top-down* descrita notrabalho de Revelles[REV00].Uma travessia do tipo *top-down* se inicia sempre no nó raiz da árvore, descendo para os nós interiores até encontrar nós folhas. Revelles propôs também o uso de cálculos de indexação dos nós a serem visitados a partir da representação paramétrica do raio. Técnicas semelhantes foram utilizadas em outras estruturas, como travessias em *kD-Tree*[JANS86]. Entretanto, Revelles propôs uma alteração específica para o algoritmo de *octree*, ao calcular os valores paramétricos nos quais o raio intersecta os três planos que dividem cada nó ou *voxel* da *octree* somente a partir de somas e divisões por dois, já que, do contrário da *kD-Tree*, os nós da *octree* se subdividem em planos de cortes pré-determinados, no meio do volume. Além disso, o algoritmo de Revelles reaproveita computações em nós anteriores para evitar repetição de cálculo.

Por simplificação, o algoritmo de Revelles é explicado nesta seção utilizando o plano 2D (através de uma *quadtree*), sendo diretamente aplicável para uma versão 3D (*octree*). Assim, um raio é definido como um par **(O,D)**, no qual **O = (Ox,Oy)** é o ponto de origem e **D = (Dx,Dy)** é o vetor de direção. Cada ponto no raio pode ser definido como: **P = O + tD**. Um ponto de

intersecção pode então ser definido a partir do valor paramétrico **t**. Uma intersecção entre um raio e um nó de *quadtree* gera diferentes casos de travessia dos sub voxels, como demonstra a Figura 5-1.T_{xm} e t_{ym} representam os valores paramétricos quando o raio cruza o meio do *quad* no eixo correspondente. A partir desses valores comparados com o da própria intersecção com o nó pai, é possível determinar a ordemdos *sub-quads* a serem visitados.

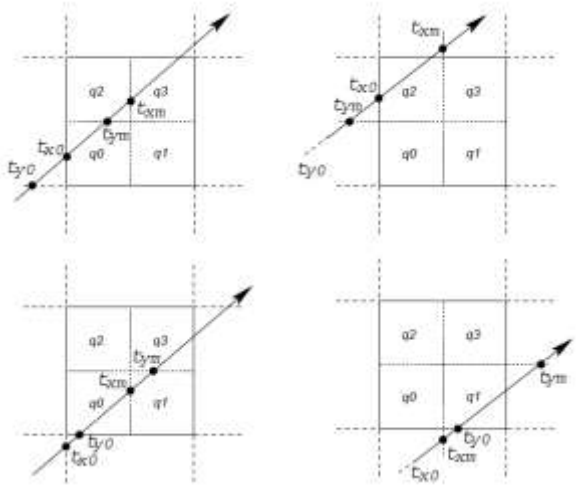
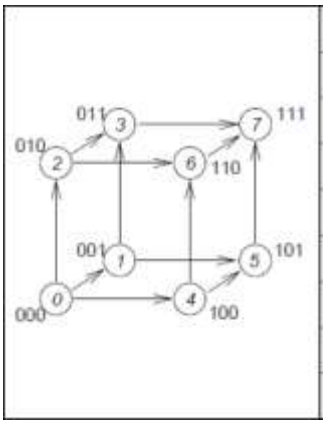


Figura 5-1. Casos de intersecção entre raio e nó de *quadtree (sub-quads q0,q1,q2,q3)*. T_{xm} e t_{ym} representam os valores paramétricos quando o raio cruza a linha horizontal do eixo correpondente.

A partir destes casos de intersecção e dos planos de corte, se define uma tabela de saída que determina o próximo *quad* ou *voxel* a ser visitado. Esta tabela também pode ser interpretada como uma máquina de estados (Figura 5-2), que descreve as possíveis transições de um nó para outro da árvore.



Nó / Estado Atual	Plano de corte YZ	Plano de corte XZ	Plano de corte XY
0	4	2	1
1	5	3	Fim
2	6	Fim	3
3	7	Fim	Fim
4	Fim	6	5
5	Fim	7	Fim
6	Fim	Fim	7
7	Fim	Fim	Fim

Figura 5-2. Máquina de estados do algoritmo de Revelles.

O algoritmo de Revelles é normalmente implementado recursivamente, visitando um novo nó a partir da tabela da Figura 5-2. Entretanto, como a maioria das placas gráficas não

suportam recursão, este algoritmo foi adaptado neste trabalho para ser implementado de maneira iterativa. Foi utilizada a técnica de pseudo-recursão através de *templates*, na qual o código é definido como uma função recursiva, entretanto em tempo de compilação é gerado o código com todas as possíveis chamadas iterativas. Isto só é possível quando o nível da recursão é pré-determinada e de baixa profundidade, sendo este o caso da *octree*, já que sua profundidade é determinada pela complexidade geométrica da cena e raramente ultrapassa uma profundidade de nível 10. O pseudocódigo desta travessia é demonstrado noAlgoritmo A-1.

5.2 *Grid* uniforme

Fujimoto et al. [JIMT85] propuseram o primeiro algoritmo de travessia de raio em *grid* uniforme, chamado 3D Digital Differential Analyzer (3D-DDA), uma extensão do 2D-DDAmodificado de Bresenham [BRE65], comumente utilizado em rasterização. No algoritmo de Bresenham (Figura 5-3), se define um dos eixos como principal, chamado de *drivingaxis*, sendo geralmente o de maior dimensão, e o outro como eixo passivo. O eixo principal irá definir a largura do passo de travessia, e assim quais células serão visitadas, entretanto sem garantir que visite todas as células que interceptam a linha, como mostra a Figura 5-3. No caso de *ray tracing*, é preciso detectar todas as células possíveis.

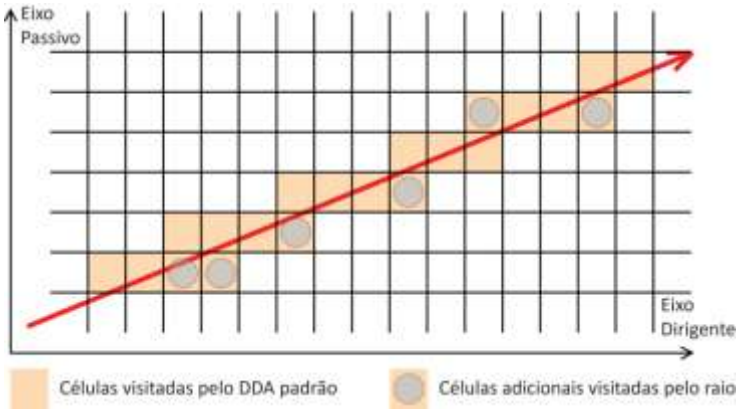


Figura 5-3. Algoritmo de DDA, no qual algumas células que o raio atravessa não são detectadas.

A extensão proposta por Fujimoto et al. no 3D-DDA lida com o problema do 2D-DDA de Bresenham ao analisar os outros eixos além do principal. Um algoritmo similar também baseado no 2D-DDA foi proposto por Amanatides e Woo [AMAN87], no qual não há discriminação entre eixos passivos e principal, além de utilizar propriedades de raios coerentes

para prevenir casos repetidos de intersecção raio-primitiva. A técnica de Amanatides e Woo é a utilizada no RT^2, já que é a mais recente e eficiente[AMAN87]. O pseudocódigo é demonstrado no Algoritmo A-2.

Um problema da travessia em *grid* se refere à visita de muitas células vazias em cenas esparsas, além da repetição de testes de raio-objeto com primitivas, já que é frequente o caso de uma primitiva estar presente em mais de uma célula.

5.3 *kD-Tree*

O primeiro algoritmo de travessia para *kD-Tree* foi introduzido por Kaplan [KAP85]. Esta travessia (nomeada como *Sequential Traversal*por Havran [HAVR01]) é baseada numa busca não recursiva de localização de pontos espaciais do raio entre os nós internos da *kD-Tree*. Após alcançar um nó folha, a busca sequencial retorna ao nó raiz da árvore. Desta forma, nós internos (principalmente o nó raiz) são revisitados constantemente. O formato da travessia sequencial é ilustrado na Figura 5-4. Seu pseudocódigo é demonstrado no Algoritmo A-3.

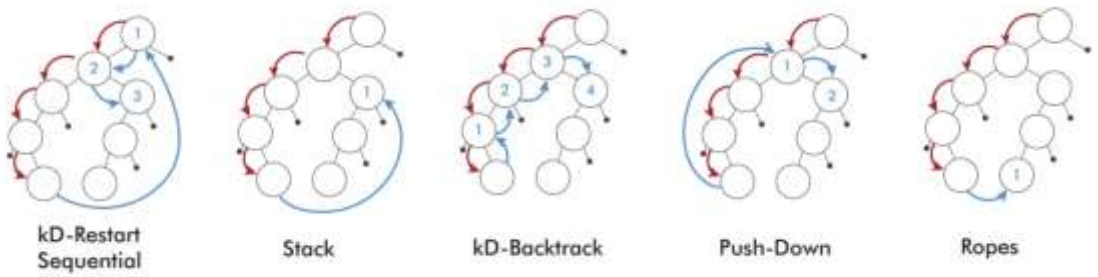


Figura 5-4. Diferentes tipos de travessias em *kD-Tree*. Setas vermelhas representam a busca seguindo para níveis inferiores da árvore, enquanto que as setas azuis representam os caminhos para a busca do próximo nó folha não visitado.

O algoritmo conhecido [HAVR01] como *Standard Traversal*(Algoritmo A-4) ou de travessia padrão reduz o número de visitas à nós internos que ocorrem na travessia sequencial. Ao interceptar um nó, um raio pode atravessar um ou ambos os filhos, como mostra a Figura 5-5. Para o último caso, na travessia padrão, uma pilha é utilizada para armazenar os nós filhos mais distantes. Esta informação armazenada pode ser utilizada posteriormente, já que a travessia segue primeiramente pela sub-árvore do filho mais próximo. Assim, a pilha garante que a travesia ocorra de acordo com a ordem de intersecção

do raio com os AABBs dos nós folhas. Além disso, diferentemente da travessia sequencial, nenhum nó da árvore é revisitado.

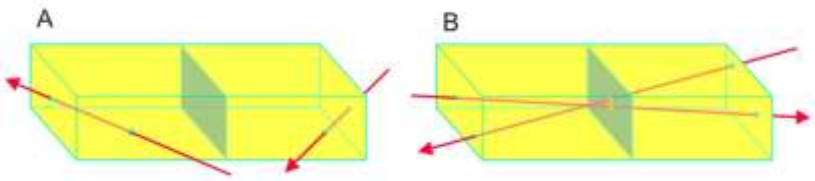


Figura 5-5. Casos de intersecção raio-nó: a) um filho; b) ambos os filhos.

Através de análises estatísticas de casos de intersecção raio-nó, Havran [HAVR01] modificou o algoritmo padrão para uma travessia mais robusta numericamente. Além disso, de acordo com o trabalho de Havran, a implementação deste algoritmo em uma arquitetura MIPS ofereceu um ganho de performance de até 65% quando comparado com a travessia padrão. Nesta dissertação este algoritmo é referenciado como travessia de Havran (Algoritmo A-5).

Foley et al. [FOL05]adaptaram a travessia padrão para linguagem de *shading* em GPU, criando dois novos algoritmos, *kD-Restart* e *kD-Backtrack*,que não necessitam de uma pilha para a busca. Esta modificação tornou possível implementações de *ray tracing* em placas gráficas com um limitado espaço de memória e linguagem simples, sem suporte de pilhas.

O algoritmo *kD-Restart* (Algoritmo A-6) é codificado de maneira semelhante ao algoritmo padrão. Entretanto, o *kD-Restart* retorna a busca para o nó raiz da árvore após visitar um nó-folha, como ilustrado na Figura 5-4. Esta operação de retorno ou reinício ocorre continuamente até que uma intersecção com primitiva seja encontrada ou o raio sair do AABB da cena. Não é mais necessário armazenar um possível filho distante, já que a busca sempre retorna para a raiz da árvore. Assim, no *kD-Restart* é removido o requisito de uma pilha. Entretanto o custo médio da busca cresce, já que muitos nós serão revisitados, da mesma forma do algoritmo sequencial.

Com o intuito de reduzir o número de nós revisitados e continuar sem a necessidade de uma pilha, o algoritmo *kD-Backtrack* altera a estrutura do nó interior, que passa a armazenar seu AABB e um ponteiro para o nó pai. A partir do ponteiro para o nó pai, não é preciso retornar ao nó raiz a cada nó folha encontrado. O algoritmo simplesmente retorna subindo na árvore pelos nós pais, até encontrar o nó interno que possibilita visitar outros nós não visitados, como mostra a Figura 5-4. Apesar de reduzir o número de nós visitados, o *kD-Backtrack*, comparado com a travessia padrão, demanda (ver capítulo 6) aproximadamente de

uma ordem de magnitude a mais de memóriapara armazenar toda a árvore, devido aos ponteiros aos pais e os AABBs dos nós. O pseudocódigo é demonstrado no Algoritmo A-7.

Horn et al. [HORN07] propuseram algumas modificações ao algoritmo *kD-Restart* com o objetivo de alcançar uma performance em GPU mais próxima ao da travessia padrão em CPU. Tais modificações essencialmente reduzem o número de nós revisitados. A técnica *Push-Down*(Algoritmo A-8) mantém o último nó visitado que pode ser considerado como o nó raiz. Neste nó, o raio intercepta apenas um dos dois filhos. Desta forma, toda vez que um evento de reinício ocorre, ao invés de retornar ao nó raiz da *kD-Tree*, a busca segue para o nó armazenado do *Push-Down*, de forma que o total de nós revisitados é reduzido, como ilustrado na Figura 5-4.

Além da técnica de *Push-Down*, Horn et al. [HORN07] definiram a travessia *Short-Stack*(Algoritmo A-9)para GPUs com espaço limitado de memória. O *Short-Stack* utiliza uma pequena pilha circular ao invés da pilha com tamanho equivalente à profundidade da *kD-Tree*. Caso a pilha se esvazie e o raio não intercepte uma primitiva, um evento de reinício é processado, de maneira similar ao *kD-Restart*, redirecionando a busca para o nó raiz da árvore.

O algoritmo de *Push-Down* e *Short-Stack* otimizam partes diferentes do *kD-Restart*, podendo então ser utilizados juntos. Neste caso, se um evento de reinício é disparado, ao invés de retornar para o nó raiz, a busca retorna para o nó previamente armazenado pelo algoritmo de Push-Down. Neste trabalho, tal algoritmo híbrido é chamado de PD & SS. Seu pseudocódigo é demonstrado no Algoritmo A-10.

Popov et al. [PPOV07] demonstraram uma implementação em CUDA da técnica de ropes, uma travessia sem pilha que utiliza o conceito de *Ropes*[HAVR01] ou cordas, no qual cada nó folha armazena seu AABB, além de 6 ponteiros (*ropes*) para o nó vizinho das faces do AABB. Desta forma, quando um nó folha é alcançado, ao invés de retornar ao nó armazenado numa pilha, um dos 6 ponteiros é utilizado. A Figura 5-4 ilustra este conceito. Consequentemente, a travessia utilizando ropes visita menos nós que qualquer algoritmo de *kD-Tree* descrito anteriormente. Entretanto, há um custo adicional de acessos de memória para a leitura dos AABBs dos nós folhas, além da leitura dos ponteiros de *ropes*. Seu pseudocódigo é demonstrado no Algoritmo A-11.

No contexto de travessias em *kD-Tree*, este trabalho tem duas contribuições: o algoritmo *Ropes++* e uma modificação na estrutura da pilha, de algoritmos que a

necessitam(*Standard, Havran* e *Short-Stack*). Estas contribuições estão descritas nas próximas seções.

5.3.1 Ropes++

O *Ropes++*é uma melhoria do algoritmo de *Ropes*[HAVR01], sendo publicado em 2009[ALS09]. Esta técnica reduz o número de leituras de memória global e de operações aritméticas por nó folha quando comparado com o *Ropes*. Para tal, o algoritmo de intersecção foi simplificado de forma a localizar o ponto de saída do AABB do nó folha a partir de apenas metade das leituras da informação do AABB. A face de saída do raio no AABB depende da direção do raio, relacionado com os sinais de suas coordenadas 3D. A partir destas coordenadas é possível localizar a face de saída com 3 leituras, ao inves de 6 como ocorre no algoritmo de intersecção tradicional. Assim, a redução do número de leituras é alcançado pela substituição do acesso à 6 valores de ponto flutuante para 3 acessos. Como a redução de leituras de informação de AABB cai pela metade, o número de registradores necessários para o kernel também é moderadamente reduzido, possibilitando a execução em paralelo de um maior número de threads de CUDA.

O *Ropes++* também simplifica o primeiro teste de intersecção raio com o AABB da cena, normalmente necessário pra obter o ponto e entrada do raio, que representam o ponto de início e término da busca. Quando um raio não intersecta um AABB, este raio não intersecta nenhum sub-volume do AABB. Assim, o teste inicial raio-AABB da cena é adiado para quando chegar no primeiro nó-folha, mesmo quando este não é intersectado. Num nó folha, esta computação é sempre necessária. Apesar de aumentar o custo computacional quando o raio não intersecta a cena, este custo é compensado nos casos de intersecção, caso com maior probabilidade de acontecer em simulações 3D convencionais, já que a câmera geralmente faz parte da cena. O pseudocódigo do *Ropes++* é demonstrado no Algoritmo A-12.

5.3.2 8-bytes Standard traversal

O requisito de uma pilha na travessia padrão afeta consideravelmente (ver capítulo 6) a performance de implementações em GPU, principalmente quando comparado com as em CPU, já que a arquitetura em CPU geralmente apresenta um sistema de cache sofisticado, reduzindo significativamente o impacto de acessos à pilha da travessia. Como uma *kD-Tree* de alta qualidade geralmentenão é balanceada (ver capítulo 3), apresenta uma alta profundidade,

podendo chegar a mais de uma centena para cenas de alta complexidade geométrica, com muitas primitivas. Existem apenas dois espaços de memória de CUDA capazes de alocar para cada thread uma pilha deste tamanho: memória global e local. Ambos espaços de memória têm o mesmo custo de acesso. Entretanto, o espaço de memória local oferece automaticamente acessos coalescentes. Por isto, neste trabalho, as pilhas utilizadas nos algoritmos de travessia padrão foram alocados na memória local.

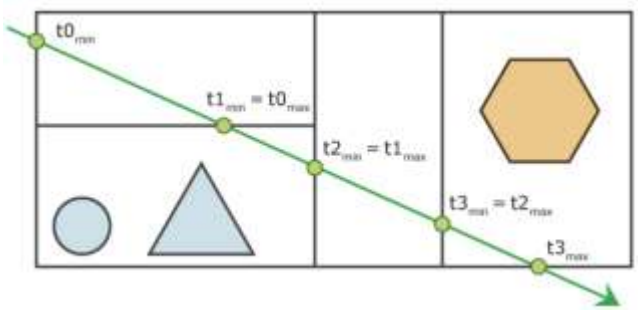


Figura 5-6. Equivalência entre *tMin* e *tMax* de nós folhas vizinhos.

Trabalhos anteriores [HAVR01] [HORN07] [KAP85] relacionados à travessia padrão utilizam um modelo de pilha com três variáveis, que armazenam a informação necessária para manter a travessia executando cada vez que se alcança um nó-folha. Estes três valores são compostos pelo o índice do nó a ser visitado posteriormente e pelo intervalo paramétrico [*tMin, tMax*] do raio da busca. Estes dois valores paramétricos representam os pontos de intersecção entre o o raio e o AABB do próximo nó a ser visitado. Como cada uma dessas informações são de variáveis de 32 bits, são geralmente armazenados num formato de struct de 12 *bytes*. Após implementar todos os algoritmos de travessia de *kD-Tree* pesquisados, ficou visível que com pequenas modificações no algoritmo original, seria possível remover uma das três informações, a variável *tMin*, de forma a se criar um novo struct de 8 bytes. O *tMin* se mostrou desnecessário, já que está sempre presente durante a travessia, armazenado no último *tMax* encontrado após alcançar um nó folha. Interpretando o raio em um caminho contínuo, *tMax*, representa o último ponto do raio que foi visitado, significando que este ponto também será o primeiro ponto a ser visitado quando o algoritmo remover da pílha o próximo nó a ser visitado. Então, o *tMin* armazenado na pilha tem o mesmo valor que o *tMax* encontrado no último nó folha. Este conceito é ilustrado na Figura 5-6. Como a pilha nova tem 8 *bytes* por elemento, o seu algoritmo de travessia é chamado neste trabalho como 8-*byteStandardtraversal*. É importante notar que este novo formato de pilha também pode ser utilizado com o *Short-Stack*. O *8-byte Standard traversal* é demonstrado como pseudocódigo noAlgoritmo A-13.

5.4 BVH

A forma mais comum de travessia em BVH é a *top-down*. Para nós internos, o raio é testado com AABB associado. Se uma intersecção é encontrada, o raio é testado recursivamente com os nós filhos. Diferentemente, da *kD-Tree*, é preciso visitar os dois filhos, já que estes não são ordenados e podem se sobrepor. Se o raio não intersecta um nó de BVH, este não é recursivamente visitado. Para nós folhas, o raio é testado com os triângulos contidos em sua lista. A travessia neste caso continua se tiver algum nó armazenado na pilha. Kay *et al.*[KAYK86] propõe um método no qual o filho mais próximo da origem do raio é o primeiro a ser visitado. Para tal, é utilizada uma fila de prioridade de forma a extrair rapidamente o próximo nó a ser visitado. Entretanto, não foi relatado o ganho de performance com a adição desta fila. Shirley et al. [SHIR03]advertiram sobre o impacto negativo de uma fila de prioridade ao mostrar que não houve um ganho significativo de performance com esta abordagem.

A travessia implementada neste trabalho segue o trabalho de Shirley et al. [SHIR03], com contribuições adicionais de Lindoso [JRG10]. Por ser uma abordagem top-down, automaticamente leva a uma estruturação de busca em árvore binária. O algoritmo de travessia envolve múltiplos cálculos de intersecção do raio com AABBs dos nós. No cálculo de intersecção há o retorno de dois valores, um que indica a distância da origem do raio ao ponto de entrada da caixa, chamado de *tMin*, e outro que indica a distância do raio ao ponto de saída da caixa, chamado da *tMax*. Primeiramente ocorre a intersecção do raio com o AABB da cena toda ou o nó raiz e posteriormente se realiza a travessia em caso de intersecção. O nó atual é constantemente atualizado na medida em que vão ocorrendo intersecções do raio com os AABBs dos nós. Quando ocorre a intersecção em um nó e este é um nó interno da estrutura, ocorre a verificação da travessia nos dois nós filhos e surgem casos a serem tratados.

Quando o raio intercepta somente um dos filhos, o nó atual é atualizado e passa a ser o nó filho interceptado. Entretanto, existe o caso em que o raio pode interceptar ambos e cabe ao algoritmo tratar esta ocorrência. Para este caso, o algoritmo escolhe o nó com menor *tMin*de intersecção, surgindo o problema de a travessia neste nó escolhido falhar, não encontrando intersecção e o de outro nó com *tMin* maior ser desprezado. Para isto, o algoritmo utiliza uma estrutura de pilha de maneira similar à *kD-Tree* para armazenar os nós

com *tMin* maior para uma posterior verificação da travessia. Se o nó atual for folha, ocorre a verificação da travessia nas primitivas contidas neste nó a fim de encontrar a primitiva com menor distância de intersecção. OAlgoritmo A-14demonstra esse processo de travessia.

5.5 BIH

Atravessia da *BIH*[WCHT06] [RA10]é bastante similar à da travessia padrão da *kD-Tree*, com pilha. A busca na árvore ocorre de maneira *top-down*, e caso o raio atravesse dois nós, um é empilhado para ser processado posteriormente. Entretanto, do contrário da *kD-Tree* e da mesma forma que a *BVH*, é preciso processar todos os nós empilhados antes de terminar a busca, já que é possível que uma intersecção mais próxima seja encontrada num dos nós empilhados. Esta é uma das principais diferenças do algoritmo de travessia padrão da *kD-Tree* para o da BIH, com a adição de um *loop* interno no fim do algoritmo.

Devido a sua abordagem de utilização de planos que definem intervalos na estrutura, é possível que um raio atravesse a estrutura sem visitar nenhum nó filho. Este caso ocorre quando o raio passa pelo intervalo de valores entre o plano de corte da esquerda e da direita, que representa o espaço que não contem nenhuma primitiva. Como não há a possibilidade de intersectar alguma primitiva, o raio sai imediatamente da estrutura sem ter que visitar mais nós. É então uma vantagem em termos de performance e uso de memória, pois o algoritmo de travessia implicitamente ignora espaços vazios. Sendo assim, os nós vazios não precisam ser armazenados nem acessados na estrutura, reduzindo o uso de memória.

Como início do processo de travessia, o raio é testado contra oAABB que envolve toda a *BIH*. Como resultado deste teste, tem-se os dois valores que determinam o intervalo válido para o parâmetro de saída*t*: o *tMin* e o *tMax*. Estes valores representam a distância paramétrica dos pontos de interseção do raio com o AABB da cena.

Caso o raio seja originado dentro da cena, ou seja, quando a câmera se encontra dentro da estrutura, o valor paramétrico *tMin* recebe um valor negativo. Após o teste de interseção com oAABB da cena, é feita uma checagem sobre o valor de *tMin* e, caso seja negativo, o valor é zerado.

Em seguida, calcula-se a distância paramétrica de interseção do raio com os planos de corte dos dois filhos. Caso estes valores estejam fora do intervalo válido estabelecido por

*tMin*e *tMax*, significa que o raio está atravessando o espaço vazio contido entre os dois nós, que não possui nenhuma primitiva, logo a travessia é finalizada sem interseção válida.

Caso o raio intersecte apenas um dos nós, e o nó em questão não for um nó folha, a travessia prontamente pula para o dito nó, sem haver a necessidade de testar o outro nó e sem empilhar o nó que foi intersectado. Esta característica se deve ao fato de haver apenas um nó intersectado. Caso os dois nós sejam intersectados, empilha-se o nó que está mais longe da origem do raio e inicia-se a travessia do nó mais próximo.

As duas últimas etapas, que dizem respeito à travessia de nós, são repetidas até que se encontre algum nó folha. Uma vez que esta condição é satisfeita e uma folha encontrada, é realizado o processamento deste nó folha. Para cada primitiva presente no nó folha intersectado são computados os valores de *t*, *u* e *v* de interseção com o triângulo. Os valores finais para estes parâmetros são os da primitiva que resultar no menor valor de *t*, significando que é a primitiva mais próxima da origem do raio.

Por fim, o nó que está no topo da pilha é desempilhado e processado do mesmo modo. No fim do processo de travessia, também é retornado o índice da primitiva intersectada no *array* de primitivas. Este índice é utilizado para buscar as normais dos vértices das primitivas e interpolar em cima dos parâmetros *u* e *v* e realizar o cálculo de iluminação e sombreamento corretamente. O pseudocódigo da travessia da BIH é demonstrado no Algoritmo A-15.

5.6 SBG

Da mesma forma que na *kD-Tree*, a travessia do SBG se define por visitar em ordem os AABBs que o raio intersecta. Entretanto, na *kD-Tree*, para localizar o próximo AABB é preciso visitar múltiplos nós internos até chegar num nó-folha. Isto também ocorre nos algoritmoscom*ropes*, mas em menor frequência. No caso do SBG, por ser uma estrutura dupla, a localização do próximo AABB pode ser feita em tempo constante por cálculo de indexação do ponto de entrada do raio no AABB. Este ponto indexado representa uma célula no *grid* de índices que armazena o índice do AABB que o contém. A Figura 5-7 mostra as células de pontos de entrada em cada AABB intersectado. Por ser um simples cálculo de índice, O SBG reduz a complexidade assintótica do subproblema de busca pelo próximo AABB para O(1), enquanto que na *kD-Tree* este problema é resolvido em O(log n) [HAVR01] para o número de

nós, já que neste último caso a busca pelo próximo AABB ocorre a partir da própria busca em árvore.

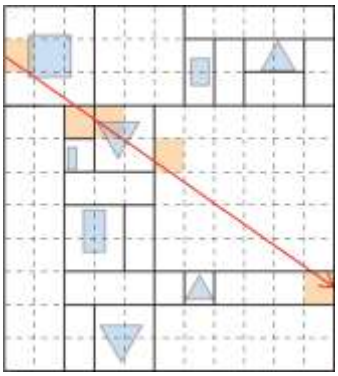


Figura 5-7. Travessia do SBG. Apenas os AABBs intersectados e suas células de entrada são visitados.

O pseudocódigo da travessia do SBG é demonstrado noAlgoritmo A-16. Apesar de não ser um fator comprobatório de simplicidade, é importante observar que o pseudocódigo da travessia do SBGse mostra mais compacto, com menor número de linhas que todos os algoritmos pesquisados neste trabalho.

Este capítulo descreveu os algoritmos de travessia das estruturas de aceleração introduzidas no capítulo 3. As técnicas descritas neste capítulo foram implementadas em GPU como parte do RT^2 (ver capítulo 4). Assim, foi preciso adaptar cada algoritmo para a arquitetura de CUDA, de forma a se obter um melhor aproveitamento do *hardware* utilizado para *ray tracing* no RT^2. Estas adaptações foram descritas no capítulo, exibindo detalhes de implementação em GPU destes algoritmos. Resultados numéricos e qualitativo destas implementações e análise comparativa são apresentados no capítulo 6.

6 Análise Comparativa e Resultados

Este capítulo apresenta uma análise comparativa entre todos os algoritmos de travessia e de técnicas de *ray tracing* pesquisadas neste trabalho. Esta análise foi obtida a partir de uma coleta de resultados, relatados em diveros aspectos quantitativos, como tempo de execução para cenas com diferentes complexidades; consumo de memória; número de instruções; grau de divergência de *branches*eperformance em multi-GPU. Os resultados foram coletados em uma máquina Intel Core i7 3.2 GHz com 8 GB de RAM e duas placas gráficas NVIDIA GeForce GTX 480, no sistema operacional Windows 7 Professional de 64 bits. O Toolkit de CUDA utilizado foi o de versão mais recente no momento dos testes: 3.2. A versão do Driver de GPU foi a 260.99. As placas GTX 480 utilizadas nos testes contêm 15 multiprocessadores cada, com *clock* de 1401 MHz e 1.5 GB de RAM. Tais GPUs fazem parte da arquitetura Fermi [CUDA10], com *cache* na memória local e global.

A maioria dos modelos 3D utilizados nos testes sãoprovenientes do *Stanford 3D Scanning Repository*[STAN10]: *Bunny* (69k triângulos), *Dragon* (847 mil triângulos), *Happy Buddha* (1.08 milhão de triângulos) e *Asian Dragon* (3.6 milhões de triângulos). Tais modelos são demonstrados na Figura 6-1. O *Asian Dragon* original tem mais de 7 milhões de triângulos. Em nossos testes este modelo teve que ser simplificado em um editor 3D, já queem algumas estruturas de aceleração esta quantidade de primitivas excedia a capacidade máxima da memória da GPU. A malha do *Asian Dragon* foi então reduzida para metade do número original de triângulos.

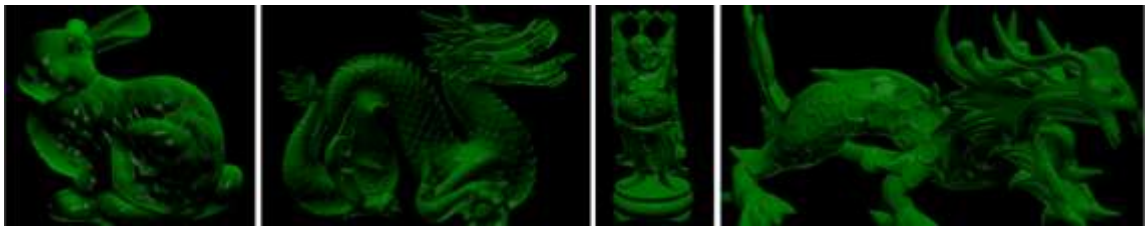


Figura 6-1. Modelos 3D do *Stanford Scanning Repository*[STAN10] utilizado nos testes. Da esquerda para direita: *Bunny*, *Dragon*, *Happy Buddha*, *Asian Dragon*.

Todos os testes foram realizados gerando imagens com 1408x768 de resolução (>1*MPixels*). Os valores dos gráficos deste capítulo foram obtidos a partir da média de 20 simulações idênticas de cada cena de teste. A ferramenta CUDA *profiler*[CUDA10] foi utilizada

para obter detalhes como número de acessos à memória global e desvios em estruturas de controle.

6.1 Análise Comparativa dos Algoritmos de Travessia

Apesar de cada algoritmo de travessia ter um conceito de busca diferente, é possível notar certa similaridade em sua estrutura. Inicialmente, antes da própria busca ser realizada, todos os algoritmos executam um teste de intersecção entre o raio e o AABB da cena. Esta operação ocorre para detectar os casos de um raio não atravessar a cena, de forma que tais raios possam finalizar sua travessia antes mesmo de realizar a busca. Outro motivo para este teste inicial é o de obtenção dos pontos de entrada e saída do raio na cena, dados utilizados com frequência na travessia e também como um dos critérios de parada do algoritmo, normalmente definido a partir do ponto de saída.

Nos testes realizados, o algoritmo em GPU de travessia da *octree* obteve o pior resultado em termos de *performance*, como mostra oGráfico 6-1. O principal motivo se refere à inflexibilidade dos planos de corte desta árvore, que são sempre definidos no meio do nó a ser subdivido. Outras estruturas como a *kD-Tree* podem subdividir o espaço em qualquer plano de corte, de forma que uma distribuição das primitivas pode ser otimizada para reduzir o número de nós que cada primitiva estará contida. Na *octree*, isto não é possível, tendo como consequência um uso de memória elevado (Gráfico 6-2), também agravado pelo fato do nó armazenar oito ponteiros dos nós filhos, além de seu AABB. A*octree* também sofre perda de performance com seu requisito de empilhamento das variáveis necessárias para a pseudo-recursão (ver seção 5.1) em GPU, já que tais variáveis vão para a memória local. Infelizmente, não foi possível obter informações numéricas em relação a esse tipo de uso, já que a ferramenta CUDA *profiler*, utilizada para a coleta dos resultados, não suporta em sua versão atual estatísticas de leitura em memória local. Em média, a *octree* obteve um resultado *2.72x*mais lento que a travessia mais eficiente (*Ropes++,* descrito na seção 6.1.1).

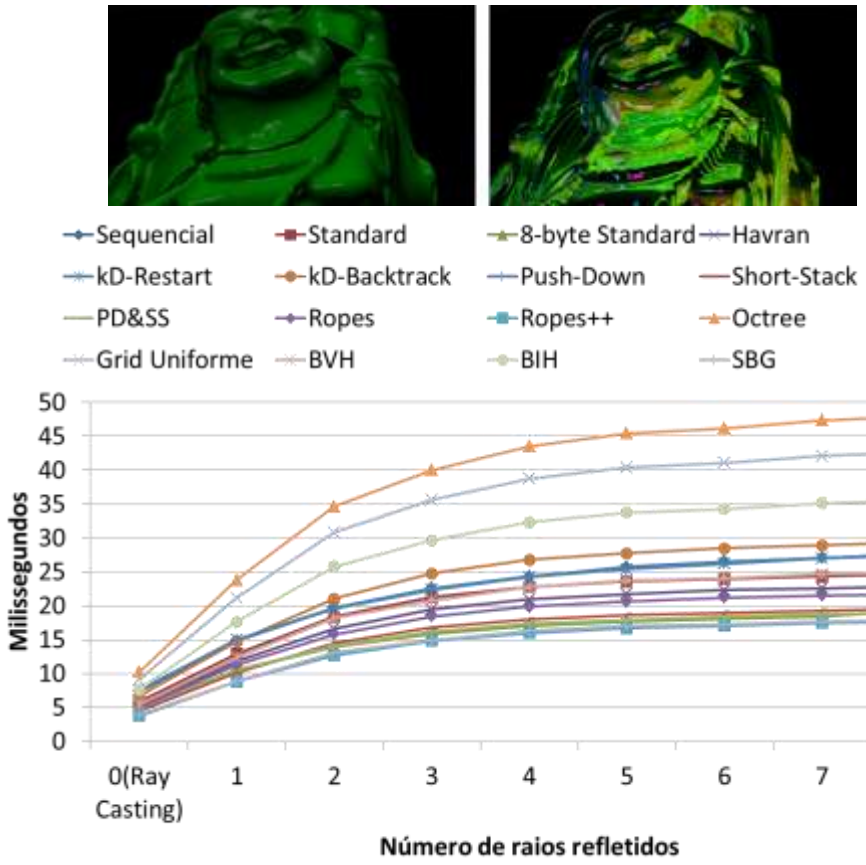


Gráfico 6-1. Performance em tempo de execução (ms) dos algoritmos de travessia, com até oito níveis de reflexão. A cena contém o modelo Stanford Happy Buddha, quatro esferas e um plano.

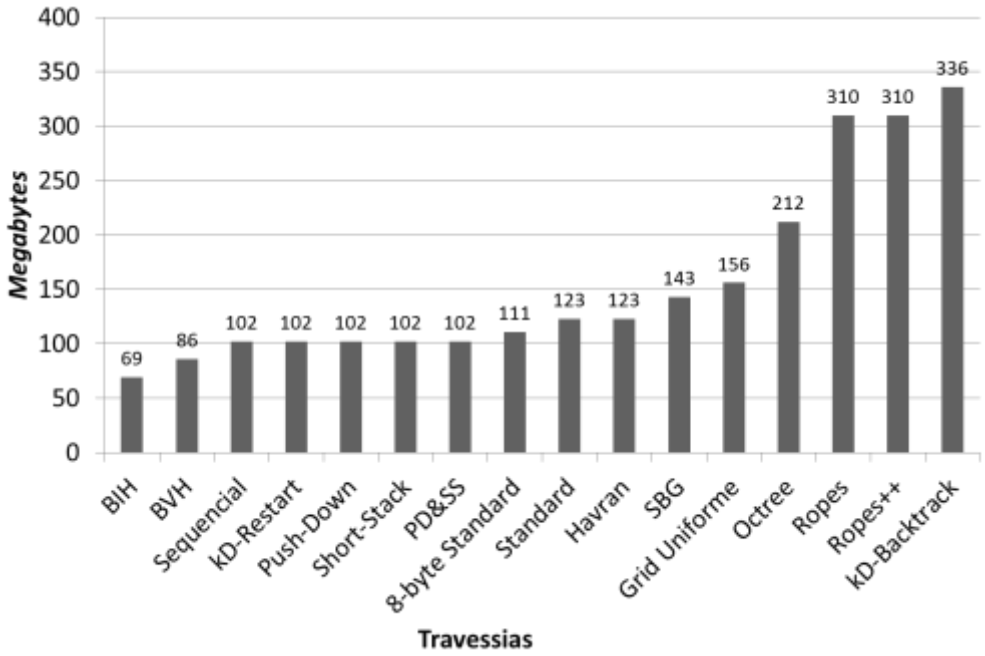


Gráfico 6-2. Uso de Memória das estruturas de dados, utilizando o modelo *Stanford Asian Dragon*, versão de 3.6 milhões de triângulos.

O *grid* uniforme se mostrou com uma performance superior à *octree*, como demonstra oGráfico 6-1. Sua simplicidade de codificação, sem recursão ou pilhas favoreceu a uma implementação mais eficiente.O uso de memória (Gráfico 6-2) se mostrou inferior ao da *octree* e superior ao da maioria dos algoritmos. Um problema de performance observado no algoritmo do *grid* uniforme se refere aos raios coerentes que visitam células diferentes, como mostra a Figura 6-2. Outras estruturas como a *kD-Tree* e SBG sofrem menos com esse problema, já que tendem a agrupar espaços vazios. Em média, o *grid* uniforme obteve um resultado *2.4x* mais lento que a travessia mais eficiente (*Ropes++*).

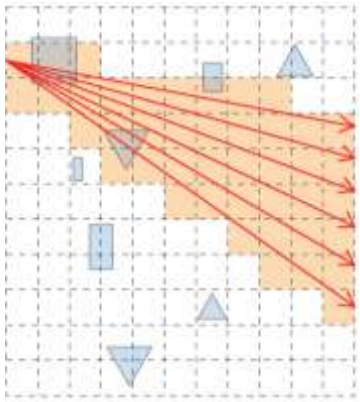


Figura 6-2.Divergência de células visitadas em raios coerentes. Apesar dos raios terem direções parecidas, visitam células diferentes.

Na maioria dos testes, a BVH obteve uma desempenho melhor(Gráfico 6-1)que o *grid*uniforme, a *octree* e a BIH, apenas sem superar o SBG ealguns algoritmos da *kD-Tree*. Apesar de a BVH apresentar a vantagem de cada nó da estrutura apresentar o seu AABB compacto, os algoritmos da*kD-Tree*ofereceram em sua maioria melhor tempo de travessia devido ao custo consideravelmente menor de verificação de intersecção com os nós da cena. Essa verificação, sabendo que o raio interceptou o AABB do nó, consiste em apenas checar qual lado do plano de corte do nó interno o ponto se encontra. Já a verificação da BVH consiste em checar a intersecção do raio com a AABB em cada nó, sendo consideravelmente mais custoso. Entretanto, a BVH apresenta um dos melhores resultados de construção (em CPU), como mostra oGráfico 6-3. Seu uso de memória é inferior(Gráfico 6-2) ao de todasas estruturas, com exceção da BIH.

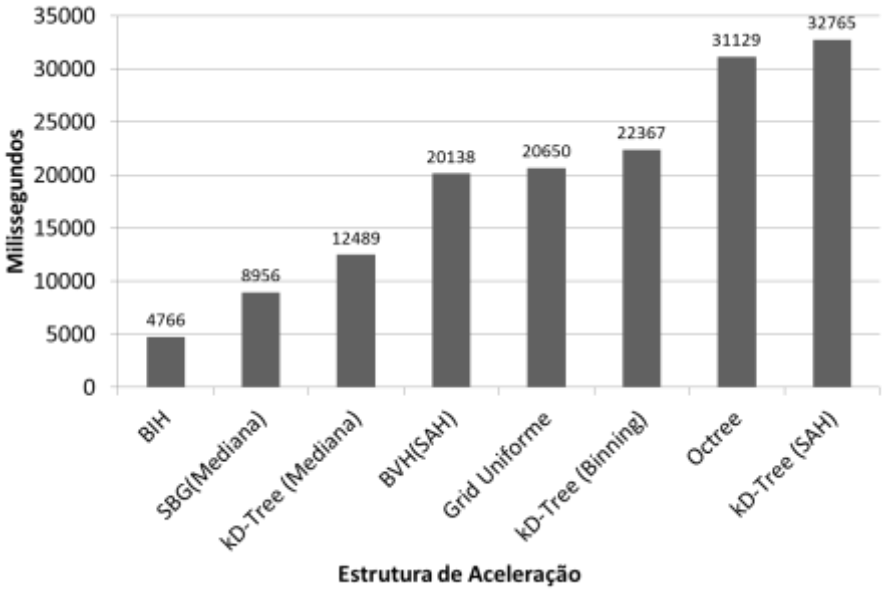


Gráfico 6-3. Tempo de construção em CPU, utilizando o modelo *Stanford Asian Dragon*, versão de 3.6 milhões de triângulos. Todos os algoritmos de construção foram implementados sem paralelização. A técnica SAH (*Surface Area Heuristisc*) [HAVR06] se baseia em probabilidade de intersecção do raio e distribuição das primitivas no espaço para determinar qual o melhor local de corte. A técnica de *Binning*[WALD09] é uma aproximação da SAH completa, de forma a alcançar melhor performance.

A BIH apresentou uma performance inferior (Gráfico 6-1)à BVH e a alguns algoritmos de *kD-Tree*. Em média, A BIH obteve um resultado aproximadamente*2x* mais lento que a travessia mais eficiente (*Ropes++*). Entretanto, se mostrou como a estrutura de menor consumo de memória (Gráfico 6-2) e tempo de construção (Gráfico 6-3), sendo assim a estrutura adotada como padrão no RT[2]paraa árvore de cenas dinâmicas de atores de corpos rígidos (ver capítulo 4).

O SBG apresentou um desempenho levemente inferior (aproximadamente 4% mais lento) que o algoritmo de travessia mais eficiente, o *Ropes++*. Entretanto, seu uso de memória é aproximadamente 46% do utilizado pelo *Ropes++*. Além disso, seu tempo de construção é menor (Gráfico 6-3) que os da *kD-Tree*e *grid* uniforme, sendo inferior apenas ao tempo de construção da BIH. Por atingir boa performance sem comprometer espaço de memória, o SBG é a estrutura adotada como padrão para modelos estáticos e partes rígidas dos atoresdo RT[2].

6.1.1 Travessias em kD-Tree

Dos 16 algoritmos de travessia implementados, 11 são para a*kD-Tree*, sendo assim a estrutura com maior número de travessias pesquisadas. Esta seção realiza uma análise em

separado para esta estrutura. Este estudo isolado é importante para comparar os detalhes específicos de *kD-Tree* que cada algoritmo apresenta.

Os algoritmos de travessia sequencial e *kD-Restart* não necessitam de uma pilha, de forma que seus códigos são simples e com um menor número de varíaveis (ver capítulo 5). Entretanto, quando a busca alcança um nó folha e retorna ao nó raiz, muitos nós são revisitados, o que consequentemente resulta em um maior número de leituras na memória global. Este procedimento reduz a performance em tempo de execução, como ilustrado noGráfico 6-1.

Comparado com *o kD-Restart*, o algoritmo *kD-Backtrack* obteve um menor número de visita a nós, como mostra oGráfico 6-4. Entretanto, como o algoritmo necessita armazenar os AABBs de todos os nós, seu consumo de memória (Gráfico 6-2) é aproximadamente 3 vezes maior que o da maioria dos algoritmos de *kD-Tree*. Nos testes realizados, o algoritmo *kD-Backtrack* é o que demandou maior espaço de memória.

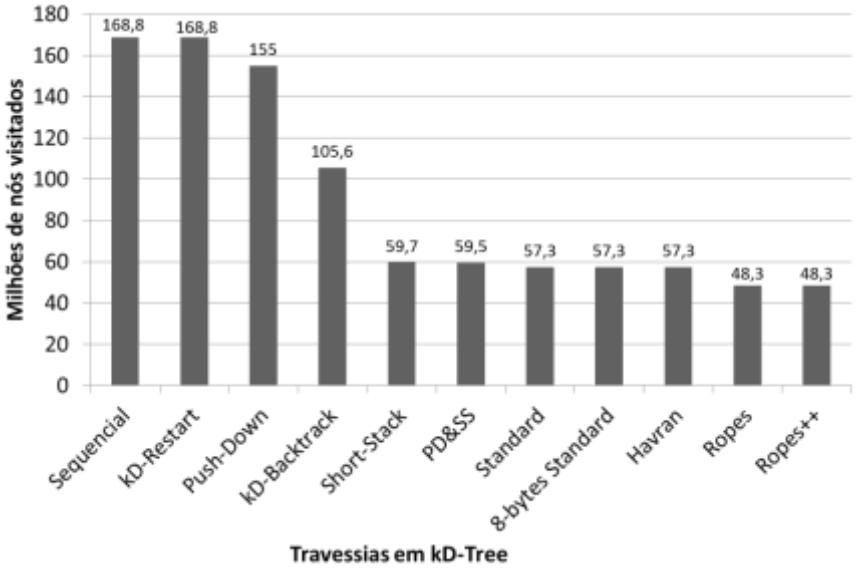


Gráfico 6-4. Número de nós visitados nas travessias em *kD-Tree*, utilizando o modelo *Stanford Asian Dragon*, versão de 3.6 milhões de triângulos.

O *Push-Down* reduz o número de nós visitados quando comparado ao *kD-Restart*. Para tal, o algoritmo adiciona estruturas de controle ao código de forma a evitar o retorno da busca para o nó raiz da árvore. Entretanto, estas estruturas de controle adicionaram casos de divergênciasGráfico 6-5, causando *stalls* em instruções de desvio, sendo uma possível causa para a performance levemente inferior (Gráfico 6-1) do *Push-Down* em relação ao *kD-Restart*.

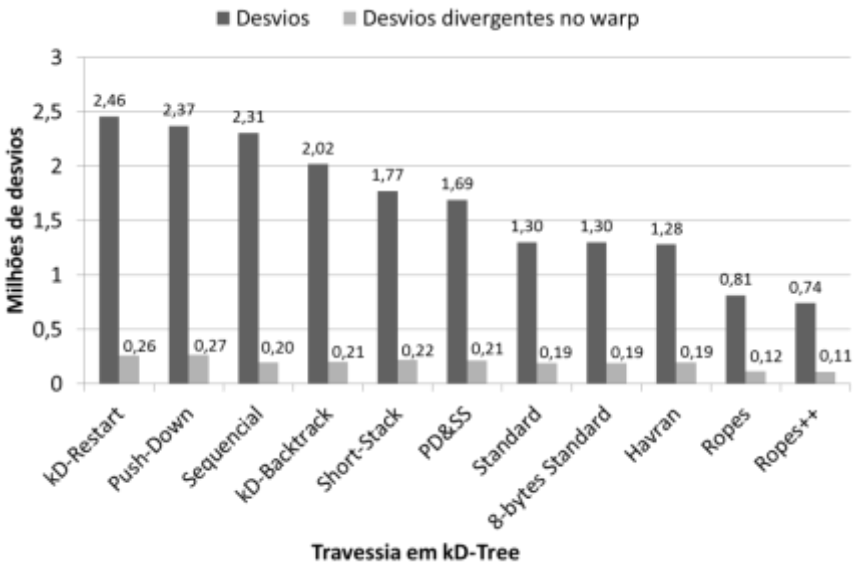


Gráfico 6-5. Número desvios e desvios divergentes por *warp* nas travessias em *kD-Tree*, utilizando o modelo *Stanford Asian Dragon*, versão de 3.6 milhões de triângulos.

Em relação ao Short-Stack, sua implementação em CUDA aloca a pilha circular na memória compartilhada, que por ser limitada, armazena um máximo de cinco nós por raio, mas sem a possibilidade de exceção de *stack overflow*, devido à natureza circular da pilha. Acessos à esta pilha é mais rápido do que o algoritmo de travessia padrão, já que o último armazena sua pilha na memória local. Entretanto, de maneira similar ao *kD-Restart*, o *Short-Stack* obriga um retorno ao nó raiz em casos de esvaziamento da pilha. Como consequência, nós são repetidamente revisitados, como mostra oGráfico 6-4. Assim, apesar do *Short-Stack* superar a performance do *kD-Restart*, não obteve um melhor desempenho que o da travessia Padrão.

O algoritmo híbrido de *Push-Down* com *Short-Stack* (PD \& SS) oferece um ganho de performance quando comparado com osalgoritmos isolados, mesmo com a adição das estruturas de desvio divergentes adicionado pelo Push-Down e operações de pilha do Short-Stack. Neste caso, a abordagem do Push-Down auxilia em limitar o uso da pilha ao "transformar" o nó raiz em nós internos da árvore, reduzindo o número de nós de pilha a serem armazenados, o que é conveniente, devido ao tamanho restrito da pilha do *Short-Stack*.

O algoritmo com *Ropes* em CUDA proposto por Popov *et al.*[PPOV07]resultou em um menor número de nós visitados (Gráfico 6-4)quando comparado com todos os algoritmos citados anteriormente. É importante observar que a travessia com *Ropes* não realiza custosas operações de divisão para cada nó visitado, apenas três operações de soma com multiplicação (FMAD [CUDA10]), que na arquitetura *Fermi*[CUDA10]é realizado em uma única instrução de

hardware. Entretanto, uma operação adicional de intersecção raio-AABB é necessária para cada nó folha, de forma a definir o ponteiro de *rope* a ser utilizado. Como é preciso armazenar os ropes e AABBs dos nós folhas, um algoritmo com Ropes utiliza aproximadamente três vezes mais memória que os outros algoritmos de travessia em *kD-Tree*, como é ilustrado no (Gráfico 6-2).

O trabalho de Günther [GUNT07] mostrou performance de travessia da BVH semelhante com a *kD-Tree* com *Ropes* de Popov *et al.*[PPOV07], sendo divergente com o resultado obtido nesta pesquisa. Entretanto, é importante observar que a arquitetura da GPU evoluiu significativamente desde o ano da publicação (2007) do trabalho de [GUNT07], com a adição de *cache*[CUDA10] em alguns espaços de memória, além de novas instruções [CUDA10], de forma que é possível que o aproveitamento da travessia em *kD-Tree* nas GPUs da época fosse inferior que o de outras estruturas.

O algoritmo com segunda melhor performance da *kD-Tree* foi o de travessia padrão com 8 bytes. O layout de 8 *bytes* ofereceu melhor desempenho que o formato de 12 *bytes*, como mostra oGráfico 6-1. Esta melhoria era previsível, pois em CUDA instruções de leitura de 64 *bits* é mais rápido que acessos de 128 *bits*, necessário para leituras de *structs* com mais de 8 *bytes*. O algoritmo de Havran obteve uma performance intermediária entre a travessia padrão de 8 e 12 *bytes*, explicado pelo crescimento do custo de travessia por nó se comparado com a travessia padrão.

O algoritmo que obteve a melhor performance para a maioria das cenas foi o *Ropes++*. O principal motivo está diretamente relacionado ao baixo número de nós visitados e das otimizações de intersecção raio-AABB do *Ropes++*, o qual reduz significativamente o total de acessos à memória global quando comparado com o algoritmo *Ropes*. Outro motivo para a boa performance é o menor número de estruturas de controle e desvios divergentes (Gráfico 6-5). Infelizmente, como é um algoritmo com *ropes*, o Ropes++ demanda mais memória quando comparado com a maioria das outras travessias, como mostra o (Gráfico 6-2).

O algoritmo de travessia mais eficiente, o *Ropes++*, foi comparado com o engine de *ray tracing* interativo da NVIDIA, o OPTIX [OPT10]. OGráfico 6-6 mostra que o *Ropes++* alcançou um desempenho quase 3x vezes superior ao da OPTIX. O modelo Asian Dragon causou um erro de estouro de memória no OPTIX, de forma que não foi possível comparar esta cena com a implementação do Ropes++.

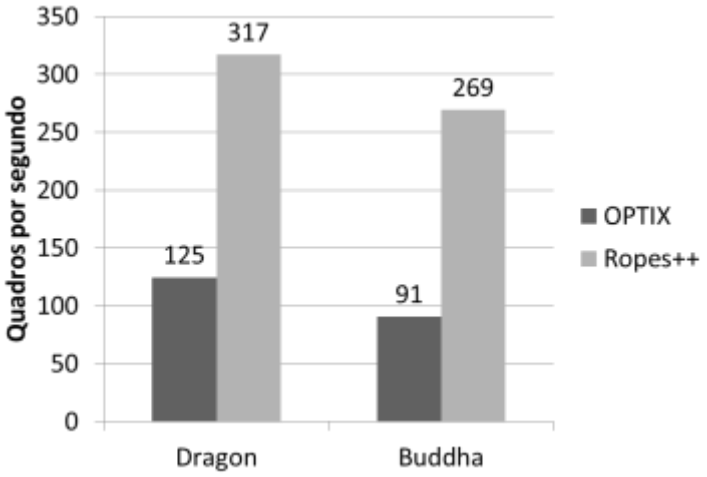


Gráfico 6-6. Comparação entre o algoritmo de travessia mais eficiente pesquisado neste trabalho (Ropes++) e o engine de ray tracing interativo OPTIX[OPT10], da NVIDIA.

Por fim, o Gráfico 6-7 mostra a redução de performance das travessias com o aumento do número de primitivas do modelo Stanford Asian Dragon. É notável a degradação do algoritmo de força bruta, que deixa de ser tempo real com apenas algumas centenas de primitivas. Já as travessias se mostraram com degradação sub-linear, mantendo interatividade mesmo com modelos de milhões de primitivas.

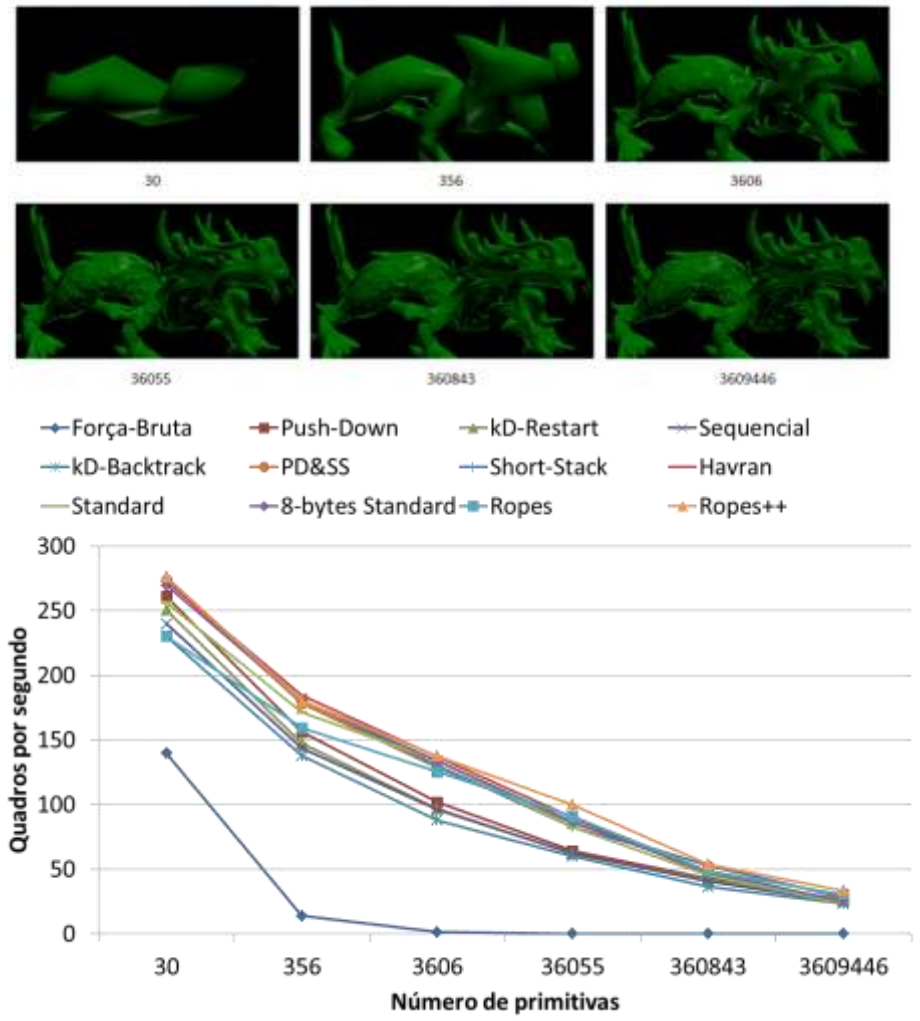


Gráfico 6-7 . Redução de performance com o aumento do número de primitivas. Observeo comportamento sub-linear das travessias.

6.2 Multi-GPU

Como descrito na seção 4.3.1, no RT^2, o *ray tracing* em multi-GPU ocorre com a subdivisão da tela em linha de *tiles* que seguem um padrão intercalado. Como demonstrado noGráfico 6-8, uma subdivisão mais simples, por linhas, resultou em pior performance quando comparada à subdivisão em *tiles*. Isto pode ser explicado pela redução de coerência dos raios em um mesmo *warp* de CUDA, já que no caso de subdivisão por linhas os raios vizinhos não são originados a partir de *pixels* vizinhos. O Gráfico 6-8também mostra resultados de *ray tracing* em resoluções diferentes para a mesma cena. Pode-se observar que a variação da performance para o número de pixels não é linear, como previsto, já que, apesar do loop primário do *ray tracing* ser sobre o número de pixels, a arquitetura da GPU tira melhor

proveito com raios coerentes, que tendem a ser cada vez mais próximos com o aumento da resolução da imagem. Além disso, a síntese de uma baixa resolução pode ocupar menos os processadores da GPU do que a de uma em alta resolução.

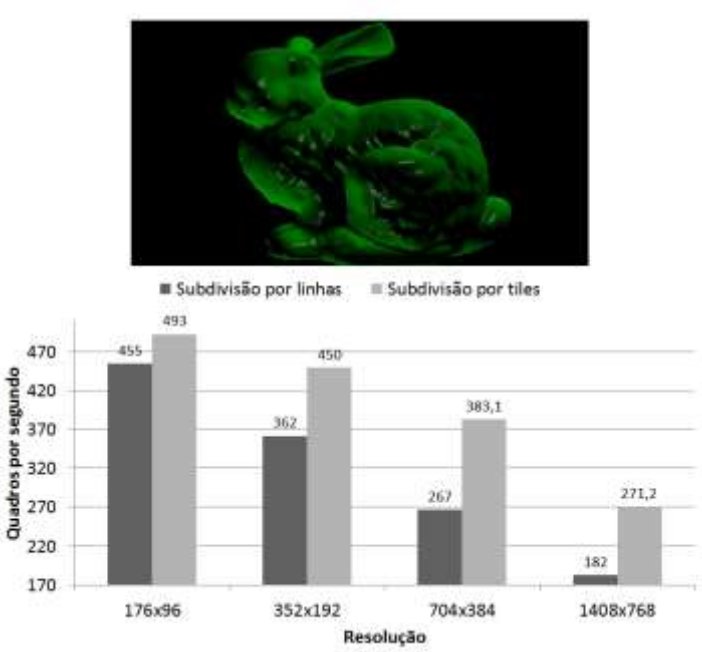


Gráfico 6-8. Subdivisão de trabalho em Multi-GPU. Por linhas vs por *tiles*. Modelo utilizado: Bunny (69 mil triângulos).

6.3 Supersampling

Com o objetivo de analisar o ganho na utilização da técnica SSABB, introduzida na seção 4.3.3.1, três estudos de caso foram avaliados. A Figura 6-3 ilustra as cenas de teste de cada estudo de caso.

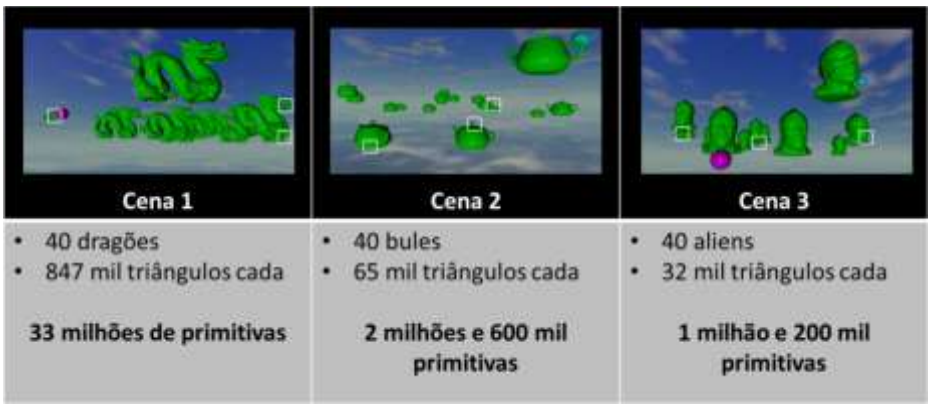


Figura 6-3. Cenas de teste de *supersampling*.

O ponto de vista das três cenas foi escolhido com o objetivo de maximizar a ocorrência de regiões com artefatos provenientes do *aliasing*. A esparsidade dos objetos na cena faz com que a malha 3D seja amostrada em poucos pixels, deixando mais visíveis os pontos de alta frequência. É possível visualizar casos de *aliasing* presentes em cada uma das cenas na Figura 6-4.

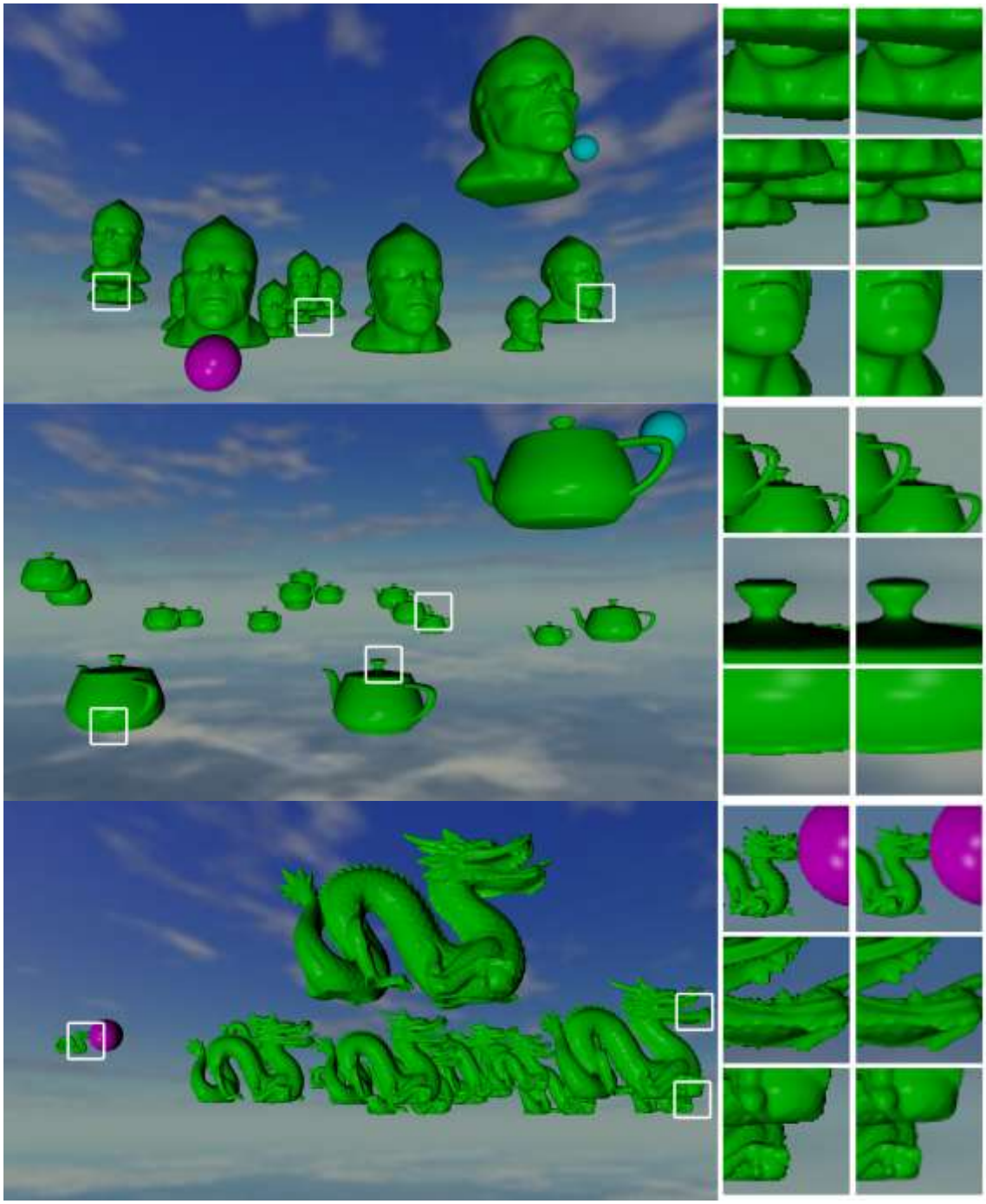


Figura 6-4. Cenas utilizadas nos estudos de caso para *supersampling*. As imagens menores mostram detalhes da cena original sem *supersampling* vs com *supersampling*.

A Tabela 6-1apresenta o tempo de execução em cada uma das cenas, quando não se utiliza a abordagem SSABB. Ao adotar a técnica proposta, é possível verificar um ganho relevante no desempenho do *ray tracing*, conforme mostrado no Gráfico 6-9.

Tabela 6-1.Tempo de execução (em milissegundos) das cenas sem a utilização do SSABB.

	Cena 1	Cena 2	Cena 3
Sem *anti-aliasing*	4,73	1,71	2,70
Supersampling 4x	17,88	5,95	9,87
Supersampling 9x	40,29	12,74	21,74
Supersampling 16x	70,91	22,77	38,97
Supersampling 25x	110,4	35,35	60,78

Apesar destes resultados dependerem fortemente da cena, foi possível obter ganho de desempenho variando de 26 a 40% para a cena 1, de 49 a 117% para a cena 2 e de 37 a 111% para a cena 3. Quanto menor a porcentagem de pixels de alta frequência, maior a redução no tempo de execução, de acordo com o observado nas cenas 1, 2 e 3. Conforme esperado, quanto menor a quantidade de *pixel*s a serem pós-processados, maior o ganho em desempenho com a técnica SSABB.

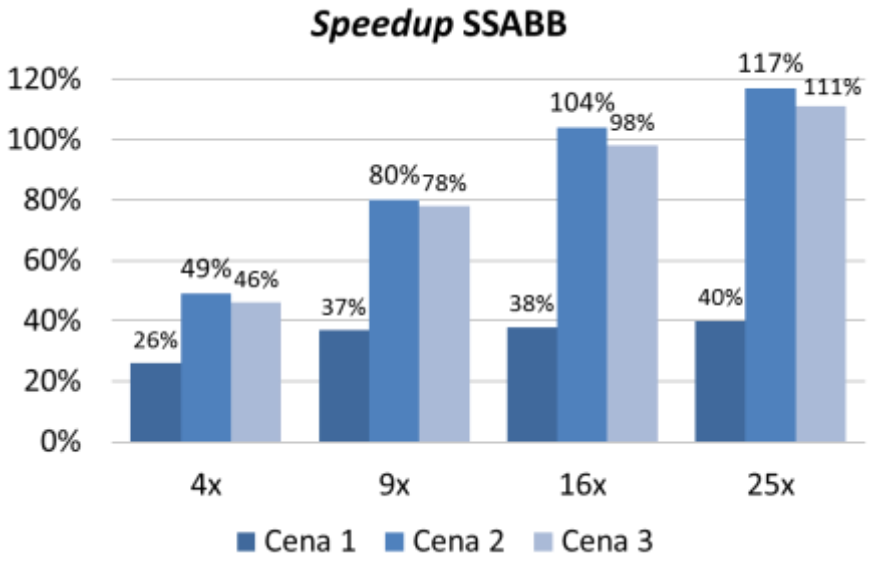


Gráfico 6-9. Speedup *alcançado* ao utilizar a técnica de *supersampling* adaptativo baseado em bordas (SSABB). Todas as três cenas foram analisadas levando em consideração diferentes níveis de *supersampling*(4x,9x,16x e 25x).

Este capítulo apresentou uma análise comparativa de diferentes estruturas de aceleração implementadas em GPU. Um considerável número de algoritmos (16) de travessia de estruturas de aceleração foi comparado em termos númericos, de forma a, principalmente, definir-se um algoritmo capaz de oferecer o menor tempo de execução. O melhor resultado foi alcançado pela *kD-Tree* utilizando um método de travessia proposto pelo autor deste trabalho, o *Ropes++*, que chega a ser 2.7x mais eficiente que o pior algoritmo de travessia em GPU, o da *octree*. Entretanto, outro algoritmo além do *Ropes++* se destaca com boa performance: a travessia do SBG, estrutura de aceleração proposta neste trabalho. Com performance próxima ao *Ropes++* (inferior em 4%), o SBG consome menos memória (aproximadamente 46% da *kD-Tree* com *Ropes*), por não ser uma estrutura em árvore que necessita armazenar os nós e *ropes* presentes no *Ropes++*.

Outra estrutura de aceleração que obteve bons resultados é a BIH, que apesar de necessitar de maior tempo de execução que a *kD-Tree*, tem o menor uso de memória e menor tempo de construção em CPU, sendo utilizada no RT^2como a estrutura de aceleração para geração da hierarquia de atores em cenas dinâmicas, enquanto que o algoritmo de travessia do SBG é utilizado para a geração das partes rígidas dos atores, como modelos e terrenos estáticos. No RT^2, os outros algoritmos estão disponíveis para uso, mas não são utilizados por padrão, já que oferecem pior performance ou maior uso de memória ou maior tempo de construção. Entretanto, é importante notar que todos os algoritmos de travessia estudados foram capazes de executar *ray tracing* em tempo real em imagens de alta definição. Além disso, foi observado que a arquitetura de CUDA favorece implementações de algoritmos de travessia de codificação simples, que apresentam poucas estruturas de controle, ausentes de pilha ou recursão.

Também foram apresentados resultados relacionados a *ray tracing* em multi-GPU, resultadosestes não encontrados na literatura pesquisada. A abordagem de paralelismo por subdivisão em *tiles* se mostrou mais eficiente que o algoritmo simples de subdivisão por linhas, apesar de ter desfavorecido em 15% o balanceamento de carga.Por fim, este capítulo também apresentou o SSABB, técnica de *supersampling* que utiliza processamento de imagens para aprimorar o desempenho da etapa de *anti-aliasing* em *ray tracing*. Esta técnica atingiu *speedups* de até 117% comparada com a técnica de *supersampling* pura. Uma vez que a abordagem independe da implementação do *ray tracer* adotado, a técnica pode ser incorporada facilmente em projetos que se beneficiam do paradigma de GPGPU.

7 Conclusão

Este trabalho apresentou uma compilação de diferentes técnicas aplicadas para o desenvolvimento de *ray tracing* em tempo real. Um considerável número de algoritmos (16) de travessia de estruturas de aceleração foi comparado em relação à performance, de forma a, principalmente, obter um algoritmo capaz de oferecer o menor tempo de execução. O melhor resultado foi alcançado pela *kD-Tree* utilizando um método de travessia proposto pelo autor deste trabalho, o *Ropes++*, que chega a ser 2.7x mais eficiente que o pior algoritmo de travessia em GPU, o da *octree*. Entretanto, outro algoritmo além do *Ropes++* se destaca com boa performance: a travessia do SBG, estrutura de aceleração proposta neste trabalho. Com performance próxima ao *Ropes++* (inferior em 4%), o SBG consome menos memória (aproximadamente 46% da *kD-Tree* com *Ropes*), por não ser uma estrutura em árvore que necessita armazenar os nós e *ropes* presentes no *Ropes++*. Já a BIH, apesar de necessitar de maior tempo de execução que a *kD-Tree*e o SBG, tem o menor uso de memória e melhor tempo de construção (em CPU). Assim, no RT^2, a BIH é utilizada para geração da hierarquia de modelos 3D em cenas dinâmicas, enquanto que o algoritmo de travessia do SBG é utilizado para a geração das partes rígidas, como modelos e terrenos estáticos.

É importante notar que todos os algoritmos de travessia estudados foram capazes de executar *ray tracing* em tempo real em imagens de alta definição. Além disso, foi observado que a arquitetura de CUDA favorece implementações de algoritmos de travessia de codificação simples, que apresentam poucas estruturas de controle, ausentes de pilha ou recursão.

Também foram apresentados resultados relacionados a processamento em multi-GPU, resultadosestes não encontrados na literatura pesquisada. A abordagem de paralelismo por subdivisão em *tiles* se mostrou mais eficiente que o algoritmo simples de subdivisão por linhas, apesar de ter desfavorecido em 15% o balanceamento de carga.

Este trabalho também apresentou o SSABB, técnica de *supersampling* que utiliza processamento de imagens para aprimorar o desempenho da etapa de *anti-aliasing* em *ray tracing*. Esta técnica atingiu *speedups* de até 117% comparada com a técnica de *supersampling* pura. Uma vez que a abordagem independe da implementação do *ray tracer* adotado, a técnica pode ser incorporada facilmente em projetos que se beneficiam do paradigma de GPGPU.

7.1 Contribuições

A principal contribuição deste trabalho se refere ao estudo comparativo das estruturas de aceleração (capítulos 3, 5 e 6) e seus algoritmos em GPU para *ray tracing* em tempo real, a partir de extensa revisão bibliográfica dos principais algoritmos de travessia. Este estudo foi fundamental para definir quais algoritmos de travessia teriam, em uma GPU atual, os melhores resultados para *ray tracing* de modelos de corpos rígidos, assim como o de melhor performance para a construção de cenas dinâmicas com corpos deformáveis.

Outra contribuição importante se refere à definição de uma nova estrutura de aceleração, o SBG, uma estrutura híbrida entre *kD-Tree* e *grid* uniforme, com performance equivalente ao *Ropes++* e vantagem de menor consumo de memória.

Algumas contribuições pontuais merecem ser mencionadas, como:

- Uma técnica de *anti-aliasing* para *ray tracing*que utiliza processamento de imagens, o SSABB (capítulos 4 e 6);
- Melhoria de um *pipeline* de *ray tracing*atravésdo RT^2para aplicações gráficas em tempo real. Este mesmo *pipeline* oferece performance superior ao do OPTIX, biblioteca da NVIDIA considerada no estado-da-arte de *ray tracing* interativo;
- Pesquisa e experimentação com programação em CUDA. O estudo aprofundado desta arquitetura foi fundamental para obter melhor aproveitamento do *hardware* utilizado.

Por fim, foram publicados os seguintes artigos:

- um artigo em periódico internacional [ALS11] foi aceito para publicação em 2011;
- um artigo completo em conferência nacional [JMMX10]foi publicado em 2010;
- um artigo completo em conferência internacional [ALS09]foi publicado em 2009.

7.2 Trabalhos Futuros

Alguns trabalhos futuros planejados são:

- Estudo comparativo de todos os algoritmos de construção em GPU das estruturas de aceleração pesquisadas. Esta pesquisa será importante para alcançar melhor performance em cenas com muitos modelos deformáveis;
- Definição de um algoritmo de construção específico para o SBG. Como descrito na seção 3.6, os algoritmos de construção utilizados no SBG foram baseados nos da *kD-Tree*, que por sua vez se baseiam em construção de árvore binária, com recursão, o que dificulta sua implementação de forma paralela. Como o SBG não é uma árvore binária, é possível utilizar diferentes abordagens para sua construção, como as baseadas em *grid* uniforme;
- Suporte à programação no módulo de *scripts* do RT^2. Assim, o desenvolvedor poderá definir funções de *shading* em *ray tracing* de maneira similar ao das linguagens de *shading* em *pipeline* programável de rasterização;
- Suporte à refração, ainda não presente no RT^2, que atualmente permite apenas transparência.

Referências

[3DS10] 3DS Max [Online]. Disponível em http://usa.autodesk.com/. Último acesso: Junho de 2010.

[ACAD10] AutoDesk AutoCAD [Online].Disponível em http://usa.autodesk.com/. Último acesso: Junho de 2010.

[AILA09] T. Aila e S. Laine, “Understanding the efficiency of ray traversal on gpus,” em HPG’09: Proceedings of the Conference on High Performance Graphics 2009. New York,NY, USA: ACM, 2009, pp. 145–149.

[ALS09] A. L. dos Santos, J. M. X. N. Teixeira, T. S. M. C. de Farias, V. Teichrieb, e J. Kelner, “kd-tree traversal implementations for ray tracing on massive multiprocessors: A comparative study”, SBAC-PAD 2009, 2009. IEEE Computer Society.

[ALS11] A. L. dos Santos, J. M. Teixeira, Thiago F. S. M. C., Veronica Teichrieb, J. Kelner, “Understanding the Efficiency of kD-tree Ray-Traversal “, 2011, International Journal of Parallel Programming.

[ALSR09] A. L. dos Santos, “RT^2 – Real Time Ray Tracer”, Trabalho de Conclusão de Curso, 2009. UFPE, Recife, Pernambuco.

[AMAN87] J. Amanatides and A. Woo. “A fast voxel traversal algorithm for ray tracing”.EUROGRAPHICS’87, Conference Proceedings, pages 3–10, 1987.

[AO07] M. Knecht, “State of the art report on ambient occlusion”. Techreport, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, 2007.

[APPEL68] A. Appel, “Some techniques for shading machine renderings of solids”, AFIPS ’68 (Spring Joint Computer Conference), 1968. New York, Estados Unidos.

[ARCH10] ArchiCAD. Disponível em http://www.archiCAD.com/. Último acesso: Dezembro de 2010.

[BENT75] J. Bentley. “Multidimensional binary search trees used for associative searching”. Communications of the ACM, 18:509–517, 1975.

[BRDL02] G. S. Brodal, R. Fagerberg, and R. Jacob, “Cache oblivious search trees via binary trees of small height,” in SODA ’02: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2002, pp. 39–48.1

[BRE65] J. Bresenham, “Algorithm for computer control of a digital plotter”. IBM Systems Journal, 1965, 25–30.

[CNET09] “The end of Larrabee”, http://news.cnet.com/8301-13924_3-10409715-64.html. Último acesso: Dezembro de 2010.

[CORM01] T. H. Cormen, C. E. Leieserson, L., RIVEST, C. Stein, “Introduction to algorithms”, 2001.

[CUDA10] NVIDIA CUDA Programming Guide 3.2. Disponível em: http://developer.download.nvidia.com/compute/cuda, Último acesso: Dezembro de 2010.

[EBER03] D.H. Eberly, “Game Physics”, 2003. Elsevier Science. New York, Estados Unidos.

[FOL05] T. Foley, J. Sugerman, “Kd-tree acceleration structures for a GPU Raytracer”, Proceedings of the ACM SIGGRAPH/Eurographics Conference on Graphics Hardware, ACM, New York, USA, 2005, pp. 15-22.

[GLAS85] A. Glasnner. Space Subdivision for Fast Ray Tracing, 1985.Spring.

[GLAS89] A. Glassner, An Introduction to Ray Tracing, Academic Press, London, 1989.

[GONZ08] R. C. Gonzalez and R. E. Woods, Digital image processing, 3rd ed. Upper Saddle River, NJ, USA: Pearson Prentice Hall, 2008.

[GORM08] L. O'Gorman, et al., Practical algorithms for image analysus, 2nd ed. New York: Cambridge University Press, 2008.

[GUNT07] J. Günther, S. Popov, H.-P. Seidel, P. Slusallek, “Realtime Ray Tracing on GPU with BVH-based Packet Traversal”. IEEE/Eurographics Symposium on Interactive Ray Tracing, 2007.

[HAVR01] V. Havran, “Heuristic Ray Shooting Algorithms”, Phd thesis, Faculty of Electrical Engineering, Czech Technical University, Prague, 2001.

[HAVR06] V. Havran, R. Herzog, SEIDEL H.-P.: On Fast Construction of Spatial Hierarchies for Ray Tracing. In Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing (2006), Wald I., Parker S. G., (Eds.).

[HEPAT00] J. L. Hennessy, D. A. Patterson. “Organização e projeto de computadores: A interface hardware/software”. 2º ed., LTC,2000.

[HOFF90] G. R.Hofmann, “Who invented ray tracing?”, 1998.

[HORN07] D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan, “Interactive k-d tree GPU raytracing”, Proceedings of the Symposium on I3D Graphics and Games, ACM, New York, USA, 2007, pp 167-174.

[ID09] “Enemy Territory: Quake Wars”, http://www.idsoftware.com/games/enemyterritory/etqw/.Último acesso: Dezembro de 2010.

[INT09] “Intel Quake Wars Ray Traced”, http://software.intel.com/en-us/articles/quake-wars-gets-ray-traced/. Último acesso: Dezembro de 2010.

[INT10] Intel. “Intel Corporation homepage”, 2010.[Online]. Disponível em: http://www.intel.com. Último acesso: Dezembro de 2010.

[JANS86] F. Jansen, “Data structures for ray tracing,” in Data Structures for Raster Graphics, L. R. A. Kessener, F. J. Peters, and M. L. P. van Lierop, Eds. Springer-Verlag, 1986, pp. 57–73.

[JEN96] H. Jensen, “Global Illumination Using Photon Maps”, Eurographics Workshop on Rendering Techniques, Dezembro de 1996.

[JIMT85] A. Fujimoto, e Iwata , K., 1985. "Accelerated Ray Tracing." Em Computer Graphics: Visual Technology and Art (Proceedings of Computer Graphics, Tokyo '85), pp. 41–65.

[JMMX10] J. M. Teixeira, E. Albuquerque, A. L. dos Santos, V. Teichrieb, J. Kelner, “Improving ray tracing anti-aliasing performance through image gradient analysis”, Simpósio de Sistemas Computacionais, 2010. Petrópolis

[JRG10] J. E. F. Lindoso, “Uma Bounding Volume Hierarchy Para Um Ray Tracer de Tempo Real”, Trabalho de Conclusão de Curso, 2010. UFPE, Recife, Pernambuco.

[KAP85] M. Kaplan. “Space-Tracing: A Constant Time Ray-Tracer”, 1985. P. 149–158.

[KAYK86] T.L. Kay and J.T. Kajiya.Ray tracing complex scenes. Computer Graphics, 20(4):269–278,November 1986. 1, 3.2, 5.1, 6.2, 20, 6.3, 6.3, 21, 6.3, IV

[KJIY86] J. T. Kajiya, “The rendering equation”, 13th annual conference on Computer graphics and interactive techniques, ACM, 1986.

[LAGAE08] A. Lagae e P. Dutré. “Compact, Fast and Robust Grids for Ray Tracing”, 2008.Eurographics Symposium on Rendering.

[LAIN10] S. Laine e T. Karras, “Efficient Sparse Voxel Octrees”, ACM SIGGRAPH, 2010, Symposium on Interactive 3D Graphics and Games 2010.

[LAIN2-10] S. Laine e T. Karras,”Efficient Sparse Voxel Octrees – Analysis, Extensions, and Implementation”,NVIDIA Technical Report, 2010.

[NVCG10] NVIDIA. Cg Toolkit, Disponível em http://developer.nvidia.com/object/cg_toolkit.html. Último acesso: Junho de 2010.

[NVID10] NVIDIA. “Nvidia Corporation homepage”. Disponível em: http://www.nvidia.com. Último acesso: Dezembro de 2010.

[OGL05] OpenGL, “OpenGL Architecture Review Board”, OpenGL Programming Guide, 2005. Addison-Wesley Professional, ed. 5.

[OOIB] B. C. OOI, SACKS-DAVIS R., MCDONNELL K. J.: Spatial k-d-tree: An indexing mechanism for spatial databases. In IEEE International Computer Software and Applications Conference (COMPSAC) (1987).

[OPCL10] OpenCL.OpenCL – The Industry Standard for Parallel Programming of Heterogeneous Systems. http://www.khronos.org/opencl/. Último acesso: Dezembro de 2010.

[OPT10] “NVIDIA optix ray tracing engine,” 2010.[Online]. Disponível em: http://developer.nvidia.com/object/optix-home.html. Último acesso: Dezembro de 2010.

[PBRT07] M. Pharr, G. Humphreys. “Physically Based Rendering: From Theory To Implementation”. 2007.

[PPOV07] S. Popov, J. Günther, H.-P. Seidel, e P. Slusallek, "Stackless kd-tree traversal for high performance gpu ray tracing", Computer Graphics Forum, v. 26, no. 3, Blackwell Publishing, 2007, pp. 415-424.

[RA10] R. M. Arôxa, “A Bounding Interval Hierarchy e a Renderização Em Tempo Real de Cenas Dinâmicas Com Ray Tracing e PBA”, Trabalho de Conclusão de Curso, 2010. UFPE, Recife, Pernambuco.

[REV00] U. N. Revelles, M. Lastra, An Efficient Parametric Algorithm for Octree Traversal, Sistemas Informaticos , 2000.

[SHIR03] P. Shirley, R. Keith Morley: “Realistic Ray Tracing”, AK Peters, 2003, ISBN 1-56881- 198-5.

[STAN10] “The Stanford 3D Scanning Repository”.Disponível em http://graphics.stanford.edu/data/3Dscanrep/. Acessado em 1 de Junho de 2009.

[UNI310] “Unit 3D Engine,” 2010. [Online]. Disponível em: http://unity3d.com/. Último acesso: Dezembro de 2010.

[WALD01] I. Wald, P. Slusallek, C. Benthin, M. Wagner, “Interactive Rendering with Coherent Ray Tracing”, Computer Graphics Forum, v. 20, no. 3, 2001, pp. 153-164.

[WALD09] I. Wald, W. R. Mark, J. Gunther, et al. “State of the Art in Ray Tracing Animated Scenes”. Computer Graphics forum, 28 (6): 1691-1722, 2009.

[WATT00] A. Watt, 3D Computer Graphics, Pearson – Addison Wesley, New York, 2000.

[WCHT06] C. Wachter e A. Keller, “Instant ray tracing: The bounding interval hierarchy”, Eurographics Workshop/ Symposium on Rendering, Eurographics Assoc., 2006, pp. 139–149.

[WHI80] Whitted, “An improved illumination model for shaded display”, Communications of ACM, v. 23, no. 6, ACM, New York, USA, 1980, pp. 343-349.

[WIDO06] W. Donnelly, Andrew Lauritzen. "Variance Shadow Maps", 2006.

[WONG06] T. T. Wong, L. Wan, C. S. Leung, and P. M. Lam. “Real-time Environment Mapping with Equal Solid-Angle Spherical Quad-Map”, Shader X4: Lighting & Rendering, Charles River Media, 2006

[WOOP06] S. Woop, G. Marmit, P. Slusallek, “BKD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes”, Graphics Hardware (2006).

Apêndice A –Algoritmos de Travessia

Algoritmo A-1: Travessia da Octree

```
1: procedure OctreeTraversal
2: Ray r = (org, dir); {Ponto de Origem + Vetor de direção}
3: Float tMin,tMax = Entry/Exit distance of r in the tree; {Pontos de entrada e saída de
   r na árvore}
4: OctreeNode node = root; {Nó inicializado como raiz da árvore}
5: {Chamada à função recursiva da travessia}
6: Intersection result = traverseNodes(node, r, tMin, tMax, 0, result);
7: end procedure

8: function traverseNodes (OctreeNode* node, Ray r, Float tMin, Float tMax, Int& cur-
   rentOctant, Intersection& result)
9: if node.isLeaf() then
10:    {Alcançou um nó folha}
11:    Primitive intersected; {Possível primitiva intersectada}
12:    Float t; {Distância paramétrica raio-primitiva}
13:    for all Primitive p in node do
14:       {Testa r com todas as primitivas do nó folha}
15:       I = Intersection(r,p,tMin,tMax);
16:       if I ≠ null then
17:          Update t and intersected, using best (smaller t) result;
18:       end if
19:    end for
20:    if intersected ≠ null then
21:       {Encontrou uma intersecção}
22:       result = HIT(intersected,t);
23:    end if
24:    return
25: end if
26: {Testa com os planos do nó, para definir próximo voxel ou octante a ser visitado}
27: Float3 tPlanes = getPlaneIntersections(node,r)
28: currentOctant = getOctant(tPlanes, tMin,tMax,currentOctant)
29: while currentOctant ≤ 7 do
30:    (tMin,tMax) = getIntersection(node.children[currentOctant],r);
31:    {Realiza a chamada recursiva da função}
32:    traverseNodes(node.children[currentOctant],r, tMin, tMax, currentOctant, result);
33:    if result then
34:       {Algum resultado foi encontrado}
35:       return
36:    end if
37: end while
38: end function
```

Algoritmo A-1. Travessia da Octree. A recursão foi removida na versão em GPU, utilizando pseudo-recursão com *templates*.

Algoritmo A-2: Travessia do *Grid* Uniforme

```
1: Ray r = (org, dir); {Ponto de Origem + Vetor de direção}
2: Float tMin,tMax = Entry/Exit distance of r in the tree; {Pontos de entrada e saída de
   r no Grid}
3: Float t; {Distância paramétrica raio-primitiva}
4: Primitive intersected; {Possível primitiva intersectada}
5:
6: {Localiza célula inicial e configura as variáveis do 3D-DDA}
7: Int3 indices,steps,stop;
8: Float3 delta, tNext;
9: indices = max3i(0,(r.org - Grid.AABB.min()) / Grid.cellDimensions());
10: indices = min3i(indices,Grid.dimensions() - 1);
11: for all Axis axis do
12:    if r.dir[axis] < 0.f then
13:       steps[axis] = 0;
14:       stop[axis] = indices[axis];
15:       deltas[axis] = 0;
16:       tNext[axis] = INFINITY;
17:    else if r.dir[axis] > 0.f then
18:       steps[axis] = 1;
19:       stop[axis] = Grid.dimensions()[axis] / r.dir[axis];
20:       deltas[axis] = Grid.cellDimensions()[axis];
21:       tNext[axis] = tMin + ((indices[axis] + 1) * Grid.cellDimensions()[axis] +
          Grid.AABB.min()[axis] - r.org[axis]) / r.dir[axis];
22:    else
23:       steps[axis] = -1;
24:       stop[axis] = -1;
25:       deltas[axis] = -Grid.cellDimensions()[axis] / r.dir[axis];
26:       tNext[axis] = tMin + (indices[axis] * Grid.cellDimensions()[axis] +
          Grid.AABB.min()[axis] - r.org[axis]) / r.dir[axis];
27:    end if
28: end for
29: Int3 cellStep = (steps[0],steps[1]*Grid.dimensions()[0],steps[2]*Grid.dimensions[0]*Grid.dimensions[1]);
30: Int cell = indices.x + indices.y * Grid.dimensions().x + indices.z * Grid.dimensions().x
    * Grid.dimensions().y;
31: {Travessia pelo Grid}
32: while TRUE do
33:    for all Primitive p in Grid.cells[cell].primitives do
34:       {Testa r com todas as primitivas da célula}
35:       I = Intersection(r,p,tMin,tMax);
36:       if I ≠ null then
37:          {Encontrou uma intersecção}
38:          Update t and intersected, using best (smaller t) result;
39:       end if
40:    end for
41:    if intersected ≠ null then
42:       return HIT(intersected,t);
43:    end if
44:    Int axis;
45:    {Define eixo a ser utilizado para calcular deslocamento para a próxima célula}
46:    if tNext.x < tNext.y ∧ tNext.x < tNext.z then
47:       axis = 0;
48:    else if tNext.y < tNext.z then
49:       axis = 1;
50:    else
51:       axis = 2;
52:    end if
53:    tMin = tNext[axis];
54:    if tMin ≥ tMax then
55:       return MISS;
56:    end if
57:    {Atualiza variáveis do 3D-DDA e calcula índice da próxima célula}
58:    indices[axis] += steps[axis];
59:    if indices[axis] == stop[axis] then
60:       return MISS;
61:    end if
62:    tNext[axis] += deltas[axis];
63:    cell += cellStep[axis];
64: end while
65: return MISS;
```

Algoritmo A-2. Travessia do *Grid* Uniforme.

Algoritmo A-3: Travessia Sequencial da *kD-Tree*

```
1: Ray r = (org, dir); {Ponto de Origem + Vetor de direção}
2: Float tMin,tMax = Entry/Exit distance of r in the tree; {Pontos de entrada e saída de
   r na árvore}
3: Float globalTMax = tMax; {Guarda ponto global de saída}
4: Float t; {Distância paramétrica raio-primitiva}
5: Primitive intersected; {Possível primitiva intersectada}
6: repeat
7:    Node node = root; {Nó inicializado como raiz da árvore}
8:    Point pEntry = r.org + tMin * r.dir;
9:    while ¬node.isLeaf() do
10:      if pEntry[node.axis] ≤ node.splitPosition then
11:         node = node.left;
12:      else
13:         node = node.right;
14:      end if
15:   end while
16:   {AABB representado por 6 Floats (pontos mínimos e máximos)}
17:   {É preciso testar com 6 valores parametrizados}
18:   localFarX = (node.AABB[(r.dir.x ≥ 0)*3] − r.org.x)/r.dir.x;
19:   localFarY = (node.AABB[(r.dir.y ≥ 0)*3 + 1] − r.org.y)/r.dir.y;
20:   localFarZ = (node.AABB[(r.dir.z ≥ 0)*3 + 2] − r.org.z)/r.dir.z;
21:   tMax = min(localFarX, localFarY, localFarZ);
22:   localNearX = (node.AABB[(r.dir.x < 0)*3] − r.org.x)/r.dir.x;
23:   localNearY = (node.AABB[(r.dir.y < 0)*3 + 1] − r.org.y)/r.dir.y;
24:   localNearZ = (node.AABB[(r.dir.z < 0)*3 + 2] − r.org.z)/r.dir.z;
25:   tMin = max(localNearX, localNearY, localNearZ);
26:   if tMax ≤ tMin then
27:      return MISS;
28:   end if
29:   for all Primitive p in node do
30:      {Testa r com todas as primitivas do nó folha}
31:      I = Intersection(r,p,tMin,tMax);
32:      if I ≠ null then
33:         {Encontrou uma intersecção}
34:         Update t and intersected, using best (smaller t) result;
35:      end if
36:   end for
37:   if intersected ≠ null then
38:      return HIT(intersected,t);
39:   end if
40:   if tMax ≥ globalTMax then
41:      {Fim da Busca}
42:      node = null;
43:   end if
44: until node == null
45: return MISS;
```

Algoritmo A-3. Travessia sequencial em *kD-Tree*.

Algoritmo A-4: Travessia Padrão (*Standard*) da *kD-Tree*

```
1: Ray r = (org, dir); {Ponto de Origem + Vetor de direção}
2: Primitive intersected; {Possível primitiva intersectada}
3: Float t; {Distância paramétrica raio-primitiva}
4: Stack < Node, Float , Float > stack; {Pilha da travessia (12 bytes por elemento)}
5: Float tMin,tMax = Entry/Exit distance of r in the tree; {Pontos de entrada e saída de
   r na árvore}
6: Node node = root; {Nó inicializado como raiz da árvore}
7: while TRUE do
8:    while ¬node.isLeaf() do
9:       {Enquanto não encontrar um nó folha}
10:      axis = node.axis();
11:      diff = node.splitPosition - r.org[axis];
12:      tDist = diff / r.dir[axis];
13:      (nearNode,farNode) = getOrderedNodes(r,node);
14:      if tDist < 0 ∨ tDist ≥ tMax then
15:         {Visita apenas filho mais próximo}
16:         node = nearNode;
17:      else
18:         if tDist ≤ tMin then
19:            {Visita apenas o mais distante}
20:            node = farNode;
21:         else
22:            {Visita ambos os filhos. O mais distante é empilhado}
23:            stack.push(farNode,tDist,tMax);
24:            node = nearNode;
25:            tMax = tDist;
26:         end if
27:      end if
28:   end while
29:   {Alcançou um nó folha}
30:   for all Primitive p in node do
31:      {Testa r com todas as primitivas do nó folha}
32:      I = Intersection(r,p,tMin,tMax);
33:      if I ≠ null then
34:         Update t and intersected, using best (smaller t) result;
35:      end if
36:   end for
37:   if intersected ≠ null then
38:      {Encontrou uma intersecção}
39:      return HIT(intersected,t);
40:   end if
41:   if stack.isEmpty() then
42:      return MISS;
43:   end if
44:   {remove da pilha o próximo nó a ser visitado}
45:   (node,tMin,tMax) = stack.pop();
46: end while
47: return MISS;
```

Algoritmo A-4. Travessia padrão (*Standard*) da *kD-Tree*.

Algoritmo A-5: Travessia de Havran para *kD-Tree*

```
1: Ray r = (org, dir); {Ponto de Origem + Vetor de direção}
2: Primitive intersected; {Possível primitiva intersectada}
3: Float t; {Distância paramétrica raio-primitiva}
4: Stack < Node, Float, Float > stack; {Pilha da travessia (12 bytes por elemento)}
5: Float tMin,tMax = Entry/Exit distance of r in the tree; {Pontos de entrada e saída de
   r na árvore}
6: Node node = root; {Nó inicializado como raiz da árvore}
7: while TRUE do
8:    while ¬node.isLeaf() do
9:       {Enquanto não encontrar um nó folha}
10:      axis = node.axis();
11:      splitPosition = node.splitPosition;
12:      valueEntry = r.org[axis] + tMin *r.dir[axis];
13:      valueExit = r.org[axis] + tMax *r.dir[axis];
14:      Node farNode;
15:      if valueEntry ≤ splitPosition then
16:         {Visita filho esquerdo}
17:         node = node.leftChild();
18:         if valueExit ≤ splitPosition then
19:            {Visita apenas filho esquerdo}
20:            continue;
21:         else
22:            {Visita ambos os filhos. O mais distante é empilhado}
23:            farNode = node.rightChild();
24:         end if
25:      else
26:         if valueExit ≥ splitPosition then
27:            {Visita apenas filho direito}
28:            continue;
29:         else
30:            {Visita ambos os filhos. O mais distante é empilhado}
31:            farNode = node.leftChild();
32:         end if
33:      end if
34:      diff = node.splitPosition - r.org[axis];
35:      tDist = diff / r.dir[axis];
36:      stack.push(farNode,tDist,tMax);
37:   end while
38:   {Alcançou um nó folha}
39:   for all Primitive p in node do
40:      {Testa r com todas as primitivas do nó folha}
41:      I = Intersection(r,p,tMin,tMax);
42:      if I ≠ null then
43:         Update t and intersected, using best (smaller t) result;
44:      end if
45:   end for
46:   if intersected ≠ null then
47:      {Encontrou uma intersecção}
48:      return HIT(intersected,t);
49:   end if
50:   if stack.isEmpty() then
51:      return MISS;
52:   end if
53:   {remove da pilha o próximo nó a ser visitado}
54:   (node,tMin,tMax) = stack.pop();
55: end while
56: return MISS;
```

Algoritmo A-5. Travessia de Havran para *kD-Tree*.

Algoritmo A-6: Travessia *kD-Restart* para *kD-Tree*

```
1: Ray r = (org, dir); {Ponto de Origem + Vetor de direção}
2: Primitive intersected; {Possível primitiva intersectada}
3: Float t; {Distância paramétrica raio-primitiva}
4: Float tMin,tMax = Entry/Exit distance of r in the tree; {Pontos de entrada e saída de
   r na árvore}
5: Float globalTMax = tMax; {Guarda ponto de saída, usado nos eventos de restart}
6: Node node;
7: while TRUE do
8:    node = root; {Nó atual passa a ser o nó raiz da árvore}
9:    while ¬node.isLeaf() do
10:      {Enquanto não encontrar um nó folha}
11:      axis = node.axis();
12:      diff = node.splitPosition - r.org[axis];
13:      tDist = diff / r.dir[axis];
14:      (nearNode,farNode) = getOrderedNodes(r,node);
15:      if tDist < 0 ∨ tDist ≥ tMax then
16:         {Visita apenas filho mais próximo}
17:         node = nearNode;
18:      else
19:         if tDist ≤ tMin then
20:            {Visita apenas o mais distante}
21:            node = farNode;
22:         else
23:            {Visita ambos os filhos}
24:            {O filho mais distante será visitado quando o evento de restart ocorrer}
25:            node = nearNode;
26:            tMax = tDist;
27:         end if
28:      end if
29:   end while
30:   {Alcançou um nó folha}
31:   for all Primitive p in node do
32:      {Testa r com todas as primitivas do nó folha}
33:      I = Intersection(r,p,tMin,tMax);
34:      if I ≠ null then
35:         Update t and intersected, using best (smaller t) result;
36:      end if
37:   end for
38:   if intersected ≠ null then
39:      {Encontrou uma intersecção}
40:      return HIT(intersected,t);
41:   end if
42:   if tMax ≥ globalTMax then
43:      {Fim da Busca}
44:      return MISS;
45:   end if
46:   {Evento de restart}
47:   tMin = tMax;
48:   tMax = globalTMax;
49: end while
50: return MISS;
```

Algoritmo A-6. Travessia *kD-Restart* para *kD-Tree*.

Algoritmo A-7: Travessia *kD-Backtrack* para *kD-Tree*

```
1: Ray r = (org, dir); {Ponto de Origem + Vetor de direção}
2: Primitive intersected; {Possível primitiva intersectada}
3: Float t; {Distância paramétrica raio-primitiva}
4: Float tMin,tMax = Entry/Exit distance of r in the tree; {Pontos de entrada e saída de
   r na árvore}
5: Float globalTMax = tMax; {Guarda ponto de saída global}
6: Node node = root; {Nó inicializado como nó raiz da árvore}
7: while TRUE do
8:    while ¬node.isLeaf() do
9:       {Enquanto não encontrar um nó folha}
10:      axis = node.axis();
11:      diff = node.splitPosition - r.org[axis];
12:      tDist = diff / r.dir[axis];
13:      (nearNode,farNode) = getOrderedNodes(r,node);
14:      if tDist < 0 ∨ tDist ≥ tMax then
15:         {Visita apenas filho mais próximo}
16:         node = nearNode;
17:      else
18:         if tDist ≤ tMin then
19:            {Visita apenas o mais distante}
20:            node = farNode;
21:         else
22:            {Visita ambos os filhos}
23:            {O filho mais distante será visitado quando o evento de backtrack ocorrer}
24:            node = nearNode;
25:            tMax = tDist;
26:         end if
27:      end if
28:   end while
29:   {Alcançou um nó folha}
30:   for all Primitive p in node do
31:      {Testa r com todas as primitivas do nó folha}
32:      I = Intersection(r,p,tMin,tMax);
33:      if I ≠ null then
34:         Update t and intersected, using best (smaller t) result;
35:      end if
36:   end for
37:   if intersected ≠ null then
38:      {Encontrou uma intersecção}
39:      return HIT(intersected,t);
40:   end if
41:   if tMax ≥ globalTMax then
42:      {Fim da Busca}
43:      return MISS;
44:   end if
45:   tMin = tMax;
46:   {Evento de backtrack}
47:   node = node.parent();
48:   while TRUE do
49:      Float tempTMax;
50:      localFarX = (node.AABB[(r.dir.x ≥ 0)*3] − r.org.x)/r.dir.x;
51:      localFarY = (node.AABB[(r.dir.y ≥ 0)*3 + 1] − r.org.y)/r.dir.y;
52:      localFarZ = (node.AABB[(r.dir.z ≥ 0)*3 + 2] − r.org.z)/r.dir.z;
53:      tempTMax = min(localFarX, localFarY, localFarZ);
54:      if (tempTMax - tMin) ≥ 0.001f then
55:         tMax = tempTMax;
56:         break;
57:      end if
58:      node = node.parent();
59:   end while
60: end while
61: return MISS;
```

Algoritmo A-7. Travessia *kD-Backtrack* para *kD-Tree*.

Algoritmo A-8: Travessia *Push-Down* para *kD-Tree*

```
1: Ray r = (org, dir); {Ponto de Origem + Vetor de direção}
2: Primitive intersected; {Possível primitiva intersectada}
3: Float t; {Distância paramétrica raio-primitiva}
4: Float tMin,tMax = Entry/Exit distance of r in the tree; {Pontos de entrada e saída de
   r na árvore}
5: Float globalTMax = tMax; {Guarda ponto de saída, usado nos eventos de restart}
6: Node node;
7: Node* currentRootNode = root; {Armazena o nó mais profundo que pode ser consider-
   ado raiz da busca}
8: while TRUE do
9:    Bool pushDown = TRUE;
10:   node = currentRootNode; {Retorna ao nó mais profundo que pode ser considerado
      raiz da busca}
11:   while ¬node.isLeaf() do
12:      {Enquanto não encontrar um nó folha}
13:      axis = node.axis();
14:      diff = node.splitPosition - r.org[axis];
15:      tDist = diff / r.dir[axis];
16:      (nearNode,farNode) = getOrderedNodes(r,node);
17:      if tDist < 0 ∨ tDist ≥ tMax then
18:         {Visita apenas filho mais próximo}
19:         node = nearNode;
20:         if pushDown then
21:            {Atualiza o nó raiz}
22:            currentRootNode = node;
23:         end if
24:      else
25:         if tDist ≤ tMin then
26:            {Visita apenas o mais distante}
27:            node = farNode;
28:            if pushDown then
29:               {Atualiza o nó raiz}
30:               currentRootNode = node;
31:            end if
32:         else
33:            {Visita ambos os filhos}
34:            {O filho mais distante será visitado quando o evento de restart ocorrer}
35:            node = nearNode;
36:            tMax = tDist;
37:            pushDown = FALSE;
38:         end if
39:      end if
40:   end while
41:   {Alcançou um nó folha}
42:   for all Primitive p in node do
43:      {Testa r com todas as primitivas do nó folha}
44:      I = Intersection(r,p,tMin,tMax);
45:      if I ≠ null then
46:         Update t and intersected, using best (smaller t) result;
47:      end if
48:   end for
49:   if intersected ≠ null then
50:      {Encontrou uma intersecção}
51:      return HIT(intersected,t);
52:   end if
53:   if tMax ≥ globalTMax then
54:      {Fim da Busca}
55:      return MISS;
56:   end if
57:   {Evento de restart}
58:   tMin = tMax;
59:   tMax = globalTMax;
60: end while
61: return MISS;
```

Algoritmo A-8. Travessia *Push-Down* para *kD-Tree*.

Algoritmo A-9: Travessia *Short-Stack* para *kD-Tree*

```
1: Ray r = (org, dir); {Ponto de Origem + Vetor de direção}
2: Primitive intersected; {Possível primitiva intersectada}
3: Float t; {Distância paramétrica raio-primitiva}
4: Float tMin,tMax = Entry/Exit distance of r in the tree; {Pontos de entrada e saída de
   r na árvore}
5: Float globalTMax = tMax; {Guarda ponto de saída, usado nos eventos de restart}
6: CircularStack < Node, Float , Float > shortStack; {Pilha circular da travessia}
7: Node node = root;
8: while TRUE do
9:    while ¬node.isLeaf() do
10:      {Enquanto não encontrar um nó folha}
11:      axis = node.axis();
12:      diff = node.splitPosition - r.org[axis];
13:      tDist = diff / r.dir[axis];
14:      (nearNode,farNode) = getOrderedNodes(r,node);
15:      if tDist < 0 ∨ tDist ≥ tMax then
16:         {Visita apenas filho mais próximo}
17:         node = nearNode;
18:      else
19:         if tDist ≤ tMin then
20:            {Visita apenas o mais distante}
21:            node = farNode;
22:         else
23:            {Visita ambos os filhos. O mais distante é empilhado}
24:            shortStack.push(farNode,tDist,tMax);
25:            node = nearNode;
26:            tMax = tDist;
27:         end if
28:      end if
29:   end while
30:   {Alcançou um nó folha}
31:   for all Primitive p in node do
32:      {Testa r com todas as primitivas do nó folha}
33:      I = Intersection(r,p,tMin,tMax);
34:      if I ≠ null then
35:         Update t and intersected, using best (smaller t) result;
36:      end if
37:   end for
38:   if intersected ≠ null then
39:      {Encontrou uma intersecção}
40:      return HIT(intersected,t);
41:   end if
42:   if tMax ≥ globalTMax then
43:      {Fim da Busca}
44:      return MISS;
45:   end if
46:   if stack.isEmpty() then
47:      node = root;
48:      tMin = tMax;
49:      tMax = globalTMax;
50:   else
51:      {remove da pilha o próximo nó a ser visitado}
52:      (node,tMin,tMax) = shortStack.pop();
53:   end if
54: end while
55: return MISS;
```

Algoritmo A-9. Travessia *Short-Stack* para *kD-Tree*.

Algoritmo A-10: Travessia *PD & SS* para *kD-Tree*

```
1: Ray r = (org, dir); {Ponto de Origem + Vetor de direção}
2: Primitive intersected; {Possível primitiva intersectada}
3: Float t; {Distância paramétrica raio-primitiva}
4: Float tMin,tMax = Entry/Exit distance of r in the tree; {Pontos de entrada e saída de
   r na árvore}
5: Float globalTMax = tMax; {Guarda ponto de saída, usado nos eventos de restart}
6: CircularStack < Node, Float , Float > shortStack; {Pilha circular da travessia}
7: Node node = root;
8: Node* currentRootNode = root; {Armazena o nó mais profundo que pode ser consider-
   ado raiz da busca}
9: Bool pushDown = TRUE;
10: while TRUE do
11:    while ¬node.isLeaf() do
12:       {Enquanto não encontrar um nó folha}
13:       axis = node.axis();
14:       diff = node.splitPosition - r.org[axis];
15:       tDist = diff / r.dir[axis];
16:       (nearNode,farNode) = getOrderedNodes(r,node);
17:       if tDist < 0 ∨ tDist ≥ tMax then
18:          {Visita apenas filho mais próximo}
19:          node = nearNode;
20:          if pushDown then
21:             {Atualiza o nó raiz}
22:             currentRootNode = node;
23:          end if
24:       else
25:          if tDist ≤ tMin then
26:             {Visita apenas o mais distante}
27:             node = farNode;
28:             if pushDown then
29:                {Atualiza o nó raiz}
30:                currentRootNode = node;
31:             end if
32:          else
33:             {Visita ambos os filhos. O mais distante é empilhado}
34:             shortStack.push(farNode,tDist,tMax);
35:             node = nearNode;
36:             tMax = tDist;
37:             pushDown = FALSE;
38:          end if
39:       end if
40:    end while
41:    {Alcançou um nó folha}
42:    for all Primitive p in node do
43:       {Testa r com todas as primitivas do nó folha}
44:       I = Intersection(r,p,tMin,tMax);
45:       if I ≠ null then
46:          Update t and intersected, using best (smaller t) result;
47:       end if
48:    end for
49:    if intersected ≠ null then
50:       {Encontrou uma intersecção}
51:       return HIT(intersected,t);
52:    end if
53:    if tMax ≥ globalTMax then
54:       {Fim da Busca}
55:       return MISS;
56:    end if
57:    if stack.isEmpty() then
58:       {Se pilha vazia, voltar para o nó raiz armazenado}
59:       node = currentRootNode;
60:       tMin = tMax;
61:       tMax = globalTMax;
62:       pushDown = TRUE;
63:    else
64:       {remove da pilha o próximo nó a ser visitado}
65:       (node,tMin,tMax) = shortStack.pop();
66:       pushDown = 0;
67:    end if
68: end while
69: return MISS;
```

Algoritmo A-10. Travessia *PD & SS* para *kD-Tree*.

Algoritmo A-11: Travessia *Ropes* para *kD-Tree*

```
1: Ray r = (org, dir); {Ponto de Origem + Vetor de direção}
2: Node node = root; {Nó inicializado como raiz da árvore}
3: Float tMin,tMax = Entry/Exit distance of r in the tree; {Pontos de entrada e saída de
   r na árvore}
4: Float t; {Distância paramétrica raio-primitiva}
5: Primitive intersected; {Possível primitiva intersectada}
6: repeat
7:    Point pEntry = r.org + tMin * r.dir;
8:    while ¬node.isLeaf() do
9:       if pEntry[node.axis] ≤ node.splitPosition then
10:         node = node.left;
11:      else
12:         node = node.right;
13:      end if
14:   end while
15:   {AABB representado por 6 Floats (pontos mínimos e máximos)}
16:   {É preciso testar com 6 valores parametrizados}
17:   localFarX = (node.AABB[(r.dir.x ≥ 0)*3] − r.org.x)/r.dir.x;
18:   localFarY = (node.AABB[(r.dir.y ≥ 0)*3 + 1] − r.org.y)/r.dir.y;
19:   localFarZ = (node.AABB[(r.dir.z ≥ 0)*3 + 2] − r.org.z)/r.dir.z;
20:   tMax = min(localFarX, localFarY, localFarZ);
21:   localNearX = (node.AABB[(r.dir.x < 0)*3] − r.org.x)/r.dir.x;
22:   localNearY = (node.AABB[(r.dir.y < 0)*3 + 1] − r.org.y)/r.dir.y;
23:   localNearZ = (node.AABB[(r.dir.z < 0)*3 + 2] − r.org.z)/r.dir.z;
24:   tMin = max(localNearX, localNearY, localNearZ);
25:   if tMax ≤ tMin then
26:      return MISS;
27:   end if
28:   for all Primitive p in node do
29:      {Testa r com todas as primitivas do nó folha}
30:      I = Intersection(r,p,tMin,tMax);
31:      if I ≠ null then
32:         {Encontrou uma intersecção}
33:         Update t and intersected, using best (smaller t) result;
34:      end if
35:   end for
36:   if intersected ≠ null then
37:      return HIT(intersected,t);
38:   end if
39:   exitFace = minLocalFarAxis + (r.dir[minLocalFarAxis]≥ 0)*3;
40:   node = node.ropes[exitFace];
41: until node == null
42: return MISS;
```

Algoritmo A-11. Travessia *Ropes* para *kD-Tree*.

Algoritmo A-12: Travessia *Ropes++* para *kD-Tree*

```
1: Ray r = (org, dir); {Ponto de Origem + Vetor de direção}
2: Node node = root; {Nó inicializado como raiz da árvore}
3: Float tMin = Entry distance of r in the tree; {Ponto de entrada de r na árvore}
4: Float t; {Distância paramétrica raio-primitiva}
5: Primitive intersected; {Possível primitiva intersectada}
6: repeat
7:    Point pEntry = r.org + tMin * r.dir;
8:    while ¬node.isLeaf() do
9:       if pEntry[node.axis] ≤ node.splitPosition then
10:         node = node.left;
11:      else
12:         node = node.right;
13:      end if
14:   end while
15:   {AABB representado por 6 Floats (pontos mínimos e máximos)}
16:   {É preciso testar com apenas 3 valores parametrizados}
17:   {Teste raio-AABB sem ramificações: r.dir.axis ≥ 0}
18:   localFarX = (node.AABB[(r.dir.x ≥ 0)*3] − r.org.x)/r.dir.x;
19:   localFarY = (node.AABB[(r.dir.y ≥ 0)*3 + 1] − r.org.y)/r.dir.y;
20:   localFarZ = (node.AABB[(r.dir.z ≥ 0)*3 + 2] − r.org.z)/r.dir.z;
21:   tMax = min(localFarX, localFarY, localFarZ);
22:   if tMax ≤ tMin then
23:      return MISS;
24:   end if
25:   for all Primitive p in node do
26:      {Testa r com todas as primitivas do nó folha}
27:      I = Intersection(r,p,tMin,tMax);
28:      if I ≠ null then
29:         {Encontrou uma intersecção}
30:         Update t and intersected, using best (smaller t) result;
31:      end if
32:   end for
33:   if intersected ≠ null then
34:      return HIT(intersected,t);
35:   end if
36:   exitFace = minLocalFarAxis + (r.dir[minLocalFarAxis]≥ 0)*3;
37:   node = node.ropes[exitFace];
38: until node == null
39: return MISS;
```

Algoritmo A-12. Travessia *Ropes++* para *kD-Tree*.

Algoritmo A-13: Travessia Padrão (*Standard*) para *kD-Tree* com pilha de 8 *bytes* por elemento

```
1: Ray r = (org, dir); {Ponto de Origem + Vetor de direção}
2: Primitive intersected; {Possível primitiva intersectada}
3: Float t; {Distância paramétrica raio-primitiva}
4: Stack < Node, Float > stack; {Pilha da travessia (8 bytes por elemento)}
5: Float tMin,tMax = Entry/Exit distance of r in the tree; {Pontos de entrada e saída de
   r na árvore}
6: Node node = root; {Nó inicializado como raiz da árvore}
7: while TRUE do
8:    while ¬node.isLeaf() do
9:       {Enquanto não encontrar um nó folha}
10:      axis = node.axis();
11:      diff = node.splitPosition - r.org[axis];
12:      tDist = diff / r.dir[axis];
13:      (nearNode,farNode) = getOrderedNodes(r,node);
14:      if tDist < 0 ∨ tDist ≥ tMax then
15:         {Visita apenas filho mais próximo}
16:         node = nearNode;
17:      else
18:         if tDist ≤ tMin then
19:            {Visita apenas o mais distante}
20:            node = farNode;
21:         else
22:            {Visita ambos os filhos. O mais distante é empilhado}
23:            stack.push(farNode,tMax);
24:            node = nearNode;
25:            tMax = tDist;
26:         end if
27:      end if
28:   end while
29:   {Alcançou um nó folha}
30:   for all Primitive p in node do
31:      {Testa r com todas as primitivas do nó folha}
32:      I = Intersection(r,p,tMin,tMax);
33:      if I ≠ null then
34:         Update t and intersected, using best (smaller t) result;
35:      end if
36:   end for
37:   if intersected ≠ null then
38:      {Encontrou uma intersecção}
39:      return HIT(intersected,t);
40:   end if
41:   if stack.isEmpty() then
42:      return MISS;
43:   end if
44:   tMin = tMax; {No Standard Traversal, tMin é atualizado a partir da pilha}
45:   (node,tMax) = stack.pop();
46: end while
47: return MISS;
```

Algoritmo A-13. Travessia *"8-byte Standard"* para *kD-Tree*.

Algoritmo A-14: Travessia da BVH

```
1: Ray r = (org, dir); {Ponto de Origem + Vetor de direção}
2: Primitive intersected = null; {Possível primitiva intersectada}
3: Float t; {Distância paramétrica raio-primitiva}
4: Stack <BVHNode> stack; {Pilha da travessia (4 bytes por elemento)}
5: BVHNode node = root; {Nó inicializado como raiz da árvore}
6: while TRUE do
7:    Bool intersectPrimitives = TRUE;
8:    while ¬node.isLeaf() do
9:       {Enquanto não encontrar um nó folha}
10:      (tMinLeft,tMaxLeft) = getAABBIntersection(r,node.leftAABB());
11:      (tMinRight,tMaxRight) = getAABBIntersection(r,node.rightAABB());
12:      Bool intersectLeft = tMinLeft ≤ tMaxLeft;
13:      Bool intersectRight = tMinRight ≤ tMaxRight;
14:      if ¬intersectLeft ∧ ¬intersectRight then
15:         { Raio não intersecta nós filhos}
16:         intersectPrimitives = FALSE;
17:         break;
18:      else
19:         if intersectLeft ∧ intersectRight then
20:            {Visita ambos os filhos. O mais distante é empilhado}
21:            (nearNode,farNode) = getOrderedNodes(r,node);
22:            node = nearNode; stack.push(farNode);
23:         else
24:            if intersectLeft then
25:               {Visita apenas o filho esquerdo}
26:               node = node.leftChild();
27:            else
28:               {Visita apenas o filho direito}
29:               node = node.rightChild();
30:            end if
31:         end if
32:      end if
33:   end while
34:   {Alcançou um nó folha}
35:   if intersectPrimitives then
36:      (tMin,tMax) = getAABBIntersection(r,node.AABB());
37:      for all Primitive p in node do
38:         {Testa r com todas as primitivas do nó folha}
39:         I = Intersection(r,p,tMin,tMax);
40:         if I ≠ null then
41:            Update t and intersected, using best (smaller t) result;
42:         end if
43:      end for
44:   end if
45:   if stack.isEmpty() then
46:      {Retorna resultado, sendo p == null == MISS}
47:      return (p,t);
48:   end if
49:   {remove da pilha o próximo nó a ser visitado}
50:   (node) = stack.pop();
51: end while
52: return MISS;
```

Algoritmo A-14. Travessia da BVH.

Algoritmo A-15: Travessia da BIH

```
1: Ray r = (org, dir); {Ponto de Origem + Vetor de direção}
2: Primitive intersected = null; {Possível primitiva intersectada}
3: Float t; {Distância paramétrica raio-primitiva}
4: Stack < Node, Float , Float > stack; {Pilha da travessia (12 bytes por elemento)}
5: Float tMin,tMax = Entry/Exit distance of r in the tree; {Pontos de entrada e saída de
   r na árvore}
6: Node node = root; {Nó inicializado como raiz da árvore}
7: while TRUE do
8:    Bool intersectPrimitives = TRUE;
9:    while ¬node.isLeaf() do
10:      {Enquanto não encontrar um nó folha}
11:      (nearSplit,farSplit) = getOrderedSplits(r.dir,node);
12:      (nearNode,farNode) = getOrderedNodes(r,node);
13:      if nearSplit < tMin ∨ farSplit ≥ tMax then
14:         { Raio atravessa a zona de clip}
15:         intersectPrimitives = FALSE;
16:         break;
17:      end if
18:      node = farNode;
19:      if nearSplit < tMin then
20:         {Visita apenas o mais distante}
21:         tMin = max(tMin,farSplit);
22:         continue;
23:      end if
24:      node = nearNode;
25:      if farSplit ≤ tMax then
26:         {Visita ambos os filhos. O mais distante é empilhado}
27:         stack.push(farNode,max(tMin,farSplit),tMax);
28:         node = nearNode;
29:      end if
30:      {Visita apenas filho mais próximo}
31:      tMax = min(tMax,nearSplit);
32:   end while
33:   {Alcançou um nó folha}
34:   if intersectPrimitives then
35:      for all Primitive p in node do
36:         {Testa r com todas as primitivas do nó folha}
37:         I = Intersection(r,p,tMin,tMax);
38:         if I ≠ null then
39:            Update t and intersected, using best (smaller t) result;
40:         end if
41:      end for
42:   end if
43:   repeat
44:      if stack.isEmpty() then
45:         {Retorna resultado, sendo p == null == MISS}
46:         return (p,t);
47:      end if
48:      {remove da pilha o próximo nó a ser visitado}
49:      (node,tMin,tMax) = stack.pop();
50:   until t ≥ tMin
51: end while
52: return MISS;
```

Algoritmo A-15. Travessia da BIH.

Algoritmo A-16: Travessia do SBG

```
1: Ray r = (org, dir); {Ponto de Origem + Vetor de direção}
2: Float tMin = Entry distance of r in the SBG; {Ponto de entrada de r no SBG}
3: Float t; {Distância paramétrica raio-primitiva}
4: Primitive intersected; {Possível primitiva intersectada}
5: while TRUE do
6:    Point pEntry = r.org + tMin * r.dir;
7:    {Ponto de entrada será discretizado para obter o índice do AABB a ser visitado}
8:    pEntry -= volumeAABB.min;
9:    pEntry *= volumeAABB.invGridDim;
10:   Int3 index3D = pEntry;
11:   Int linearIndex = index3D.x+index3D.y*dimensions.x+index3D.z*dimXMulDimY;
12:   Int boxIndex = cells[index3D]; {indice do AABB a ser visitado]}
13:   {AABB representado por 6 Floats (pontos mínimos e máximos)}
14:   {É preciso testar com apenas 3 valores parametrizados}
15:   {Teste raio-AABB sem ramificações: r.dir.axis ≥ 0}
16:   localFarX = (AABBs[boxIndex][(r.dir.x ≥ 0)*3] − r.org.x)/r.dir.x;
17:   localFarY = (AABBs[boxIndex][(r.dir.y ≥ 0)*3 + 1] − r.org.y)/r.dir.y;
18:   localFarZ = (AABBs[boxIndex][(r.dir.z ≥ 0)*3 + 2] − r.org.z)/r.dir.z;
19:   tMax = min(localFarX, localFarY, localFarZ);
20:   if tMax ≤ tMin then
21:      return MISS;
22:   end if
23:   for all Primitive p in AABBs[boxIndex].primitives do
24:      {Testa r com todas as primitivas do AABB}
25:      I = Intersection(r,p,tMin,tMax);
26:      if I ≠ null then
27:         {Encontrou uma intersecção}
28:         Update t and intersected, using best (smaller t) result;
29:      end if
30:   end for
31:   if intersected ≠ null then
32:      return HIT(intersected,t);
33:   end if
34:   tMin = tMax;
35: end while
36: return MISS;
```

Algoritmo A-16. Travessia do SBG.