UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA

GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

2010.2

A Bounding Interval Hierarchy e a Renderização Em Tempo Real de Cenas Dinâmicas Com Ray Tracing e PBA

TRABALHO DE GRADUAÇÃO

Aluno	Rafael de Melo Arôxa	rma5@cin.ufpe.br
Orientadora	Veronica Teichrieb	vt@cin.ufpe.br

Recife, 14 de Dezembro de 2010

A Bounding Interval Hierarchy e a Renderização Em Tempo Real de Cenas Dinâmicas Com Ray Tracing e PBA

Resumo

O ciclo virtuoso que envolve a evolução das placas gráficas e o aumento da demanda por aplicações cada vez mais realísticas e interativas, como jogos e programas de modelagem 3D, tem feito crescer o estudo sobre técnicas de iluminação global como o *ray tracing*. Este tipo de técnica consegue representar o mundo real com mais fidelidade no que diz respeito a fenômenos físicos visuais como refração e reflexão, em troca de algoritmos mais custosos computacionalmente.

Inserido neste contexto, este Trabalho de Graduação realiza o estudo comparativo entre as estruturas de dados mais utilizadas no processo de aceleração do *ray tracing* aplicado a cenas completamente dinâmicas. Este trabalho também propõe algumas otimizações nos algoritmos da *Bounding Interval Hierarchy*, fazendo desta uma forte candidata à adoção em aplicativos 3D interativos e de tempo real.

Palavras-chave: *ray tracing*, *BIH*, *BVH*, *kD-Tree*, *PBA*, tempo real

Agradecimentos

Inicialmente, eu devo agradecimentos à minha mãe e irmãos por sempre terem servido de inspiração para mim. Inspiração de quão atenciosas, preocupadas e memoráveis as pessoas são. Obrigado, Dona Maria, Timão e Baboa.

A VT, também conhecida como Veronica Teichrieb, orientadora deste trabalho, por toda sua atenção e disponibilidade nas horas que eu precisei. À sua objetividade e perfeccionismo na ajuda à concepção e desenvolvimento deste trabalho. Por tudo, valeu, VT!

A Artur Santos, por toda a atenção, ajuda e disponibilidade no desenvolvimento deste trabalho. Obrigado pelas dicas e sugestões que elevaram e muito a qualidade deste trabalho. Valeu, Artur!

Minha profunda gratidão e plena admiração a Isadora por ser a pessoa fenomenal que ela é e por sempre estar perto de mim, inclusive nos momentos de explosão do meu eu. À história que estamos construindo juntos e aos nossos planos mirabolantes. Beijing, amo tu.

A Daniel, meu amigaço e companheiro para todos os momentos. Às nossas peripécias guitarrísticas musicais e conversas-filosóficas-sobre-temas-aleatórios-que-duram-um-mol. Valeu, djow!

Às amizades que construí durante o curso. Em especial a (ordem alfabética para ninguém chorar...) Antônio, Bruno, Diego, Marcelo e Reynaldo. Valeu, Tonin, Mihiroso, Cocão, Matchelo e Lago.

Ao mundo que de algum modo influenciou em algo de bom para que eu me tornasse a pessoa que sou.

Índice

Resumo 2

Agradecimentos 3

Índice 4

1. Introdução 6

1.1. Realismo Visual 6

1.2. Realismo Comportamental 7

1.3. Objetivo 8

1.4. Estrutura do Documento 9

2. Contexto 10

2.1. Rasterização vs Ray Tracing 10

2.2. Estruturas de Aceleração 13

2.2.1. BVH 14

2.2.2. KD-Tree 15

2.3. CUDA 16

2.4. Real Time Ray Tracer – RT² 18

2.5. Point Based Animation – PBA 19

3. Bounding Interval Hierarchy 21

3.1. Conceitos gerais 21

3.1.1. Plano de corte e espaços vazios 22

3.1.2. Estrutura dos nós 24

3.1.3. Ordenação de primitivas: 25

3.1.4. Precisão numérica: 26

3.2. Construção 26

3.2.1. Detalhes de implementação 29

3.3. Travessia 31

3.3.1. Detalhes de implementação 34

3.4. Análise Comparativa com Outras Estruturas 36

3.4.1. BIH vs BVH 36
3.4.2. BIH vs KD-Tree 37
3.5. Integração com o RT² 38
3.6. Integração com o PBA 39
4. Resultados 41
4.1. Aumento de Performance 41
4.2. Consumo de Memória 43
4.3. *Time To Image* 45
5. Conclusão 48
5.1. Trabalhos Futuros 48
6. Referências 50

1. Introdução

Cada vez mais a busca por resultados realistas quando da renderização de cenas virtuais tem aumentado. Técnicas como o *raytracing* (traçado de raios) utilizam características especiais dos objetos da cena para renderizar cenas com mais realismo. Para alcançar este resultado, nuances como reflexão e refração devem ser consideradas e com isso, o custo computacional também aumenta consideravelmente.

Motivando esta demanda por renderizações mais realistas de cenas 3D virtuais, tem-se a evolução da capacidade de processamento das *GPUs* (*Graphics Processing Units*). Com processadores gráficos que possuem até 480 núcleos [7], a utilização de técnicas de renderização mais realísticas se torna muito mais viável. Esta evolução dos dispositivos gráficos pode levar a uma mudança radical na forma como as cenas virtuais são renderizadas atualmente. Gigantes da indústria, como a NVIDIA e a Intel, vislumbram em um futuro próximo a possibilidade da utilização de *ray tracing* em aplicações interativas como jogos, em face à utilização das mais evoluídas técnicas de rasterização de hoje.

1.1. Realismo Visual

As técnicas de iluminação global se baseiam na idéia de raios luminosos que são emitidos pelas fontes de luz e que se propagam pelo ambiente, sendo refletidos e refratados conforme o material dos objetos em que os raios colidem. Do mesmo modo, os raios que atingem objetos que possuem superfície rugosa são refletidos de um modo não uniforme. Este fenômeno é chamado de reflexão difusa e também é levado em conta no processo de renderização.

O cálculo da travessia do raio no ambiente é regido pelos princípios da física ótica clássica [2] e mesmo utilizando os conceitos mais básicos, a técnica de *ray tracing* consegue entregar imagens com uma qualidade gráfica bem superior às das cenas rasterizadas.

Na natureza, os raios primários sempre têm origem definida em uma fonte de luz. Isto significa que sempre os raios partem da luz, sofrem a ação de fenômenos como reflexão, refração e absorção para então chegarem aos olhos do observador. A implementação ao "pé da letra" de um sistema de renderização que funciona exatamente como acontece na natureza se tornaria inviável devido à incontável quantidade de raios emitidos pelas fontes de luz. Para tornar este problema viável computacionalmente, a idéia é fazer com que o raio percorra o caminho oposto, do olho do observador até os objetos. Neste caso, a quantidade de raios traçados é dada pela resolução da imagem gerada, em *pixels*, se tornando um fator decisivo para uma boa qualidade final da imagem gerada.

Em conjunto com as técnicas de traçado de raios está a utilização de estruturas de dados que aceleram o processo de travessia dos raios pelo ambiente virtual. Estas estruturas particionam a cena em sub-partes e fazem com que os raios traçados só realizem teste de colisão para os objetos contidos na(s) sub-área(s) em que o raio colidir.

Existem diversas estruturas que se propõem a acelerar este processo de travessia [5]. Entre as mais utilizadas e estabelecidas se encontram a *KD-Tree* (*k-Dimensional Tree*) [3][4] e a *BVH* (*Bounding Volume Hierarchy*) [6]. Neste trabalho de graduação iremos focar na *BIH* (*Bounding Interval Hierarchy*) [1], que é relativamente recente e data de 2006, mostrando suas vantagens e desvantagens em termos de tempo de construção e travessia, consumo de memória e performance geral de renderização.

1.2. Realismo Comportamental

A demanda por aplicações que simulem o mundo real através de uma interação com um mundo virtual tem crescido bastante. Não apenas na área de computação gráfica e técnicas de áudio imersivo, mas também em técnicas de simulação comportamental, como a simulação física [9]. Aplicações de Realidade Virtual e Aumentada, principalmente os jogos e simuladores, tem gerado uma preocupação especial em relação a esta última técnica. A preocupação com o realismo no comportamento dos objetos da cena tem surgido recentemente através da grande popularização do uso destas aplicações.

Técnicas de simulação física baseada em pontos, ou *point-based*, têm sido objeto de interesse e estudo, pois não exigem a existência de malhas de triângulos. Este é uma característica importante do ponto de vista da performance pois não há a necessidade de conectividade entre os pontos que compõem o objeto da cena e a implementação da técnica pode ser facilmente paralelizada. Modelos de simulação que não utilizam malha, ou *meshless*, conseguem resultados realistas e possuem um custo computacional associado à execução baixo, se comparado aos modelos que usam malha de triângulos [9].

1.3. Objetivo

Este trabalho de graduação tem por objetivo implementar uma *BIH* para acelerar a travessia de raios em um *ray tracer* de tempo real existente, chamado RT² [10]. Também foram contempladas a pesquisa e análise detalhadas dos algoritmos que envolvem a construção e travessia desta estrutura, bem como diversas otimizações que foram implementadas sobre a proposta inicial de [1].

Para comprovar a real eficiência desta estrutura no que ela se propõe a fazer, foi realizada uma integração com um sistema de simulação física baseado em pontos existente, que é capaz de realizar mudanças na forma dos objetos presentes no ambiente 3D. Estas modificações levam em consideração fatores como gravidade, densidade e elasticidade dos objetos para tornar a simulação mais real e precisa possível [8][9].

Para chegar a este objetivo, foram estudadas e analisadas as principais estruturas que são usadas para acelerar sistemas de *ray tracer*, tais como a *KD-Tree*, a *BVH* e a própria *BIH*. O estudo do estado da arte destas estruturas, realizado neste trabalho de graduação, consistiu em analisar as vantagens, desvantagens e complexidade de cada uma delas, bem como a performance temporal e a utilização de memória. Como tratam-se de *BSP Trees* (*Binary Space Partitioning Trees*) [15][16], seus algoritmos de construção e travessia possuem um custo computacional relativamente alto.

Para sanar este problema do custo computacional, foi realizado um estudo sobre *CUDA* (*Compute Unified Device Architecture*) [11] e *GPGPU* (*General Purpose computation on Graphics Processing Units*) da NVIDIA [7] que, juntos,

permitem processamento massivamente paralelo e altamente escalável de alta performance.

1.4. Estrutura do Documento

Na sequência, o capítulo 2 explica todo o contexto relacionado a este trabalho, desde os conceitos básicos de rasterização e *ray tracing*, passando pelos conceitos de simulação física baseada em pontos e as estruturas de aceleração usadas no *ray tracing*. Também é dada uma breve explicação de como funciona o *framework CUDA* para *GPGPU* assim como uma visão geral de como funciona o sistema RT^2 e o *PBA* (*Point Based Animation*) que será usado para realizar a simulação física. Em seguida o capítulo 3 apresenta em detalhes a estrutura de dados implementada neste trabalho, mostrando suas vantagens e desvantagens em relação às outras estruturas, bem como o detalhamento dos algoritmos envolvidos e a sua integração com o sistema RT^2 e o *PBA*. No capítulo 4 são mostrados os resultados obtidos do estudo da *BIH* em relação a performance e utilização de memória. E por fim, o capítulo 5 apresenta as considerações finais em relação a este trabalho e também propõe futuras melhorias.

2. Contexto

Este capítulo explica os principais temas relacionados ao desenvolvimento deste trabalho de graduação. Estes temas são de fundamental importância para o entendimento do trabalho.

2.1. Rasterização vs Ray Tracing

Os algoritmos de rasterização se baseiam principalmente na idéia de projetar um mundo virtual 3D em um plano, chamado de plano projetivo. Um algoritmo bastante simples que implementa esta técnica é o Algoritmo de Visibilidade por Prioridade [24], também conhecido como Algoritmo do Pintor, onde os objetos da cena são ordenados pela distância à câmera e processados um a um, do mais distante, para o mais próximo, sendo subdivididos em triângulos e "jogados" sobre o plano projetivo para a obtenção da imagem final (Figura 1).

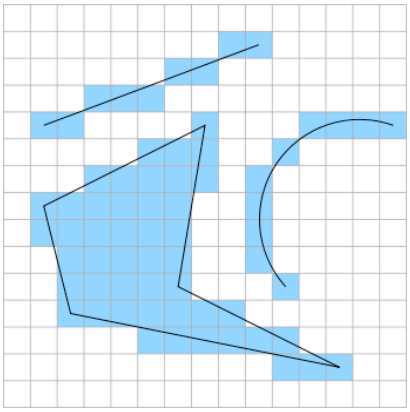


Figura 1. Resultado da rasterização de algumas primitivas como linhas, polígonos e curvas. Nota-se o surgimento de serrilhado (*aliasing*) na imagem resultante ao realizar o processo.

O estabelecimento da rasterização como principal técnica de renderização se dá pelo fato da grande adoção desta técnica nos mais diversos tipos de aplicação. Esta absorção em massa se dá pela implementação bem estabelecida de um *pipeline* gráfico que executa diretamente nas placas de vídeo. *Frameworks*

como *OpenGL* [17] e *DirectX* [14] implementam a comunicação com o *hardware* para realizar a renderização.

Esta técnica traz resultados aceitáveis do ponto de vista visual a um custo computacional relativamente baixo, diferentemente das técnicas como o *ray tracing*, que são capazes de entregar resultados com uma fidelidade bem maior (Figura 2).

Figura 2. Demonstrativo do *ray tracing* proporcionando imagens mais realísticas.

O alto custo computacional que envolve o algoritmo de *ray tracing* reside na necessidade do tratamento de colisão do raio traçado com os muitos objetos da cena. Mais precisamente, cerca de 95% do tempo total gasto no processo de renderização se dá no cálculo de interseção raio-objeto [18]. Uma cena com 100 primitivas renderizada a uma resolução de 800x600 *pixels* realizará 48.000.000 (100 testes de colisão para cada *pixel*) testes de colisões na renderização de um único *frame*. Estes testes de colisão executam uma grande quantidade de operações sobre números de ponto flutuante, em especial a multiplicação e a divisão, que são bastante custosas.

Felizmente, os testes de colisão são totalmente independentes para cada *pixel*, o que torna o algoritmo facilmente paralelizável. Mesmo com placas gráficas mais antigas e menos potentes, a utilização de *CUDA* [11] para implementação de *ray tracing* leva a altas taxas de *fps* (*frames* por segundo), devido a esta natureza altamente paralelizável deste algoritmo.

Todo sistema que utiliza o traçado de raios inicializa o processo de criação da imagem lançando raios a partir da posição da câmera em direção aos

objetos da cena. Estes raios que partem diretamente do ponto de vista são chamados de raios primários. Na implementação simples do algoritmo de *ray tracing*, um raio é lançado para cada *pixel* do plano projetivo e esta operação é denominada de *ray casting* (Figura 3).

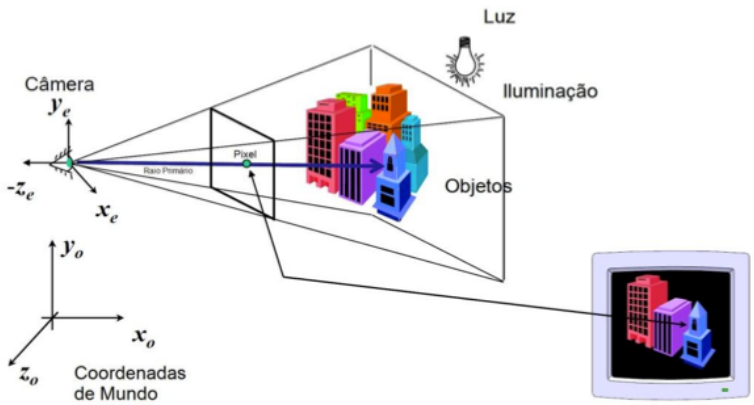


Figura 3. Representação do funcionamento do algoritmo *ray tracing*. Note que a direção do raio é definida pelo vetor que parte da origem do eixo de coordenadas da câmera e vai até o *pixel* no plano projetivo.

Dependendo do nível escolhido de recursão, ao haver uma interseção de um raio com um objeto na cena, outros raios podem ser lançados a partir do ponto de interseção. Estes são os chamados raios secundários e podem ser raios de sombra, refletidos, transmitidos ou de iluminação.

Como foi dito anteriormente, os raios de reflexão e transmissão se baseiam nas leis da física ótica clássica. Os raios refletidos têm seu cálculo obtido a partir da lei da reflexão [2] e os raios transmitidos são regidos pela lei de Snell [2] (Figura 4).

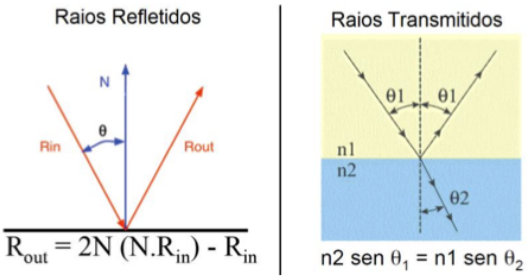


Figura 4. Esquema que ilustra o cálculo matemático envolvido na criação de raios de reflexão e de transmissão.

Uma grande vantagem ao utilizar *ray tracing* no processo de renderização é que, ao contrário da rasterização, não há a transformação das primitivas da cena em triângulos para serem projetados no plano de visão. Isto implica que os raios atingem diretamente cada objeto na cena, sejam eles primitivas, polígonos, poliedros, etc. Superfícies paramétricas [19], a exemplo das esferas, cones, quádricas, bem como as superfícies de Bèzier, podem ser renderizadas no máximo de sua qualidade gráfica. Esta característica é de grande importância para a qualidade final da renderização, pois ao aproximar a câmera deste tipo de objeto a qualidade é mantida, e vice-versa.

2.2. Estruturas de Aceleração

Tanto a *KD-Tree* quanto a *BVH* se configuram como árvores binárias que conseguem realizar a travessia do raio de modo direcionado, evitando testes de colisão desnecessários, porém com algoritmos de construção complexos. A estrutura proposta por [1], que foi implementada neste trabalho de graduação, possui o mesmo princípio de divisão binária do espaço encontrado na *KD-Tree* e na *BVH*, porém com o algoritmo de construção relativamente mais simplificado. Esta abordagem traz vantagens quando da utilização da *BIH* para cenas dinâmicas, já que se faz necessária a reconstrução de toda a estrutura para cada *frame* a taxas interativas de *fps*.

As diversas estruturas presentes na literatura atacam pontos diferentes para ser a chave de sua boa performance. Os principais pontos estão definidos nas etapas de construção e travessia dos raios pela estrutura de dados. Um algoritmo que despende um tempo maior na etapa de construção, a exemplo da heurística *SAH* (*Surface Area Heuristic*) [13]de melhor escolha do plano de corte, consegue entregar uma estrutura de alta qualidade, maximizando os espaços vazios e minimizando o custo de travessia relativo aos nós. Cenas relativamente complexas podem facilmente requerer uma grande quantidade de tempo, que pode variar de minutos até dias [12][13].

Uma estrutura construída com este tipo de heurística com uma boa qualidade, leva a uma performance alta no processo de travessia, o que é muito bom para cenas estáticas, mas se torna inviável pois a reconstrução da estrutura

a cada *frame* tiraria do usuário a sensação de interatividade, efeito de uma baixa taxa de *fps*. O tempo compreendido entre o início da etapa de construção da estrutura até o fim da travessia, incluindo o processo de *shading* [15], é conhecido como *time to image*, equivalente ao tempo total de renderização. A etapa de construção influencia o tempo em uma proporção inversa à da etapa de travessia, ou seja, quanto mais tempo despendido na construção, menor o tempo necessário para atravessar a estrutura.

2.2.1. BVH

A *Bounding Volume Hierarchy*, ou simplesmente *BVH* [6], é uma estrutura de dados de particionamento de objetos que utiliza o conceito de *AABB* (*Axis Aligned Bounding Box*) para realizar a construção da estrutura, bem como a sua travessia.

Cada nó da *BVH* contém os pontos máximo e mínimo da *AABB* que o envolve, bem como um ponteiro para cada nó filho. Embora isso aumente consideravelmente a utilização de memória, é possível garantir que as primitivas estarão localizadas dentro de caixas que possuem o volume mínimo necessário para contê-las (Figura 5). Em geral, é possível afirmar que o volume total ocupado pelas *AABBs* dos nós filhos é sempre menor que a *AABB* do nó pai. Esta característica indica uma maximização dos espaços vazios, que aumenta a performance.

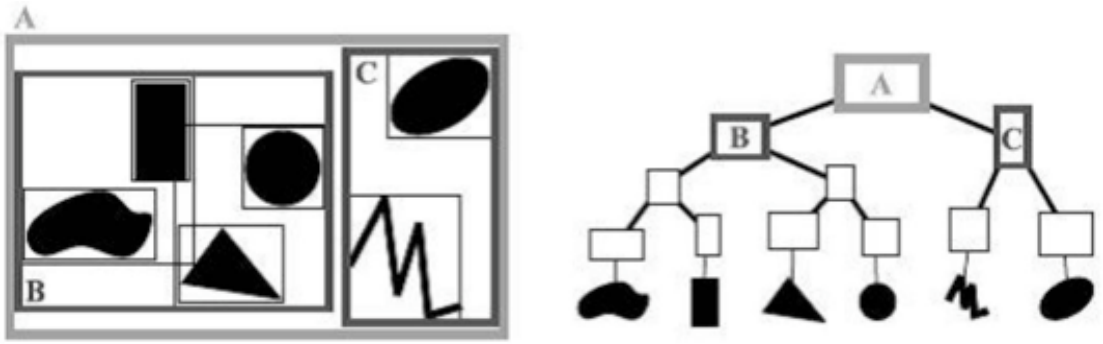


Figura 5. Esquema mostrando como se dá a organização de nós na *BVH*. A visão das *AABBs* que representam cada nó (à esquerda) e a árvore montada no processo de construção (à direita).

Devido a esta característica das primitivas serem agrupadas na menor *AABB* que as contém, é possível obter uma boa performance na etapa de

travessia. A grande quantidade de espaço vazio entre os nós faz com que o algoritmo de travessia acabe a recursão precocemente quando o raio encontra um espaço vazio.

Por se tratar de um esquema de particionamento de objetos, a *BVH* permite que os nós se sobreponham no espaço. Sendo assim, caso um raio atravesse os nós da esquerda e da direita de um mesmo nó pai, 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.

2.2.2. KD-Tree

A *KD-Tree* [3][4] é um tipo especial de árvore *BSP* (*Binary Space Partitioning Tree*) que consegue entregar resultados muito bons de performance em aplicações de *ray tracing* com cenas estáticas.

Diferentemente da *BVH*, a *KD-Tree* só tem referência a uma *AABB*, a do nó raiz. A partir desta *AABB*, o algoritmo de construção consegue estabelecer um plano de corte para cada nó, onde todos os objetos presentes em cada lado vão fazer parte de cada um dos filhos.

Por se tratar de uma árvore de particionamento de espaço, os nós filhos de um mesmo pai não se sobrepõem. Esta característica faz com que o algoritmo de construção se torne mais complexo ao ter que cuidar de casos específicos onde o plano escolhido para corte intersecta alguma primitiva. Quando esta condição é encontrada, cálculos de interseção objeto-plano são realizados para dividir a *AABB* que envolve o objeto em duas caixas menores, cada uma contendo apenas a parte da primitiva que está em cada lado do plano de corte.

Esta abordagem de construção é eficiente do ponto de vista da qualidade da árvore, porém aumenta a quantidade de memória necessária para armazenar a estrutura por completo. Ao dividir a *AABB* de uma primitiva em duas, a referência para a primitiva em questão é duplicada entre os nós, gerando redundância.

A maioria das implementações de *KD-Tree* utiliza abordagens heurísticas para a escolha do plano de corte (Figura 6), fazendo com que o custo de travessia da árvore fique mais balanceado entre os nós e o desempenho seja maior.

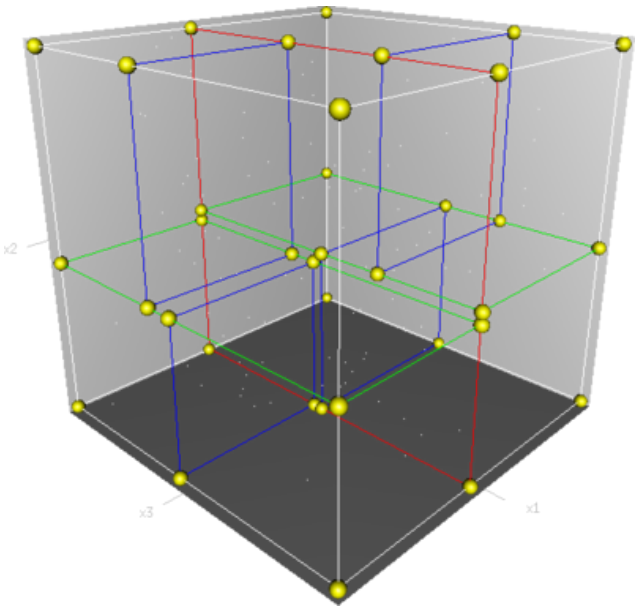


Figura 6. Esquema representando uma *AABB* geral para a estrutura e as divisões hierárquicas em cada nível da árvore. Primeiro, a divisão do primeiro nível é executada no nó raiz (em vermelho). Em seguida a divisão de nível 2 (em verde). Por fim, as divisões que geram os nós-folha (em azul).

Uma heurística bastante utilizada no momento da escolha do plano de corte durante o algoritmo de construção de estruturas como a *KD-Tree* é a *SAH*. Este cálculo consiste em medir o custo de travessia de um nó e as primitivas que ele contém e leva em consideração vários fatores, dentre eles: a área da superfície da *AABB* do nó atual, bem como as áreas dos dois nós filho e o número de triângulos. Também são levados em consideração parâmetros constantes a exemplo do custo de travessia de um nó e o custo de interseção raio-objeto. O uso desta heurística resulta em árvores de alta qualidade cuja performance de travessia é bastante elevada, porém, como o algoritmo inerente à heurística é bastante custoso, torna-se inviável a utilização dele para cenas dinâmicas.

2.3. CUDA

*Compute Unified Device Architecture*ou *CUDA* [11] é um *framework* criado pela NVIDIA para o uso no desenvolvimento de aplicações utilizando o conceito

de *GPGPU* (*General Purpose computation on Graphics Processint Units*). Este *framework* data de 2007 e foi criado para executar código nas placas da NVIDIA que suportam esta tecnologia. As primeiras placas que registram o suporte a este *framework* são as que possuem processadores G80. O *framework* de *CUDA* é composto basicamente por três componentes: o *SDK* (*Software Development Kit*), o *Toolkit* e o *Driver*.

O *CUDA SDK* contém diversos códigos de exemplo que contemplam vários tipos de aplicação, assim como as bibliotecas necessárias para compilar os códigos escritos com o uso deste *framework* e a documentação relativa às funcionalidades do mesmo.

O *CUDA Toolkit* provê ferramentas adicionais para executar e analisar o desempenho de aplicativos escritos sobre *CUDA*, o chamado *CUDA Profiler*. Além disso, contém bibliotecas adicionais para a solução de problemas corriqueiros como cálculos de álgebra linear e transformadas de Fourier.

O *CUDA Driver* realiza a comunicação de baixo nível entre o sistema operacional e a placa de vídeo e vem instalado em conjunto com o *driver* da placa de vídeo.

Os programas escritos em *CUDA* estão presentes em arquivos com a extensão ".cu" e são compilados através da chamada do *nvcc.exe* presente no *Toolkit*. Estes programas suportam uma sintaxe que muito se assemelha à de C, porém com a inclusão de algumas funcionalidades necessárias para realizar a comunicação com a *GPU*.

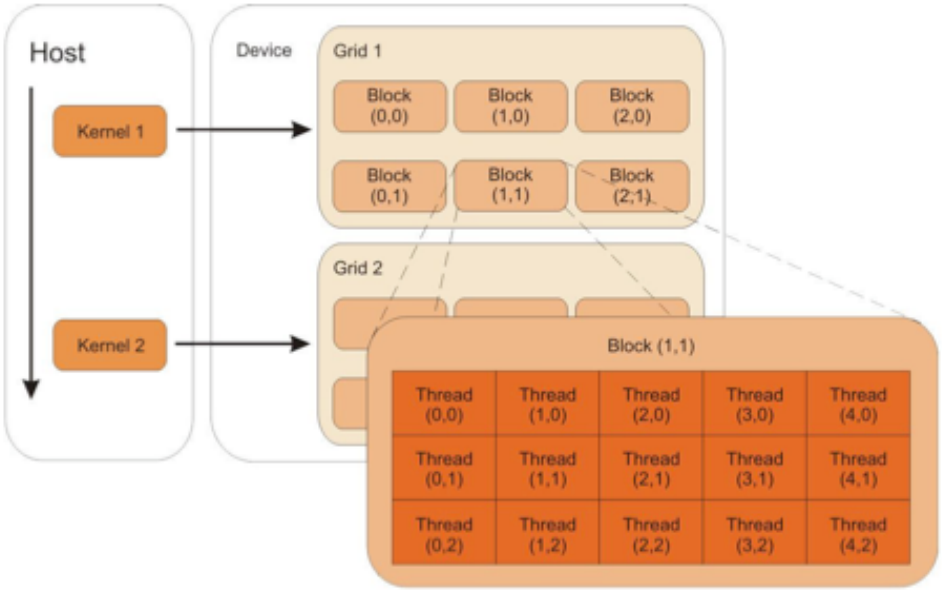


Figura7. Esquema representativo da organização de *grids*, *blocks*, e *threads* no *framework CUDA*.

Um programa que está sendo executado na *GPU* possui uma estrutura fixa e bem estabelecida de funcionamento que gira em torno de conceitos como *kernel*, *grid*, *block* e *thread* (Figura 7).

A chamada de um *kernel* configura a execução de vários blocos (*blocks*), organizados em até três dimensões. Os blocos, por sua vez, são compostos por *threads*, também organizadas em até três dimensões. Todas as *threads* executam exatamente o mesmo código. Este código é definido na declaração da função que corresponde ao *kernel.*

A linguagem permite o uso de algumas palavras-chave especiais que especificam tipos especiais de funções e variáveis. São elas: *__global__*, *__host__* e *__device__*. Os modificadores *__host__* e *__device__* são usados tanto em funções como em variáveis e servem para especificar onde elas serão alocadas e de onde elas podem ser chamadas (a partir da *CPU* ou da *GPU*), no caso das funções.

Para maiores informações sobre CUDA, pode-se consultar[20].

2.4. Real Time Ray Tracer – RT²

O sistema RT² [10][21][22] foi concebido e desenvolvido nos trabalhos de conclusão de curso e de mestrado de Artur Santos, iniciados em 2009. Os tópicos seguintes desta seção explicam as idéias gerais do que é o sistema RT², seu

objetivo e funcionamento, tais como as principais classes envolvidas no processo de renderização implementado pela aplicação.

O RT² foi concebido seguindo o conceito de que deve haver uma camada de abstração entre o código que o programador de uma aplicação 3D escreve e o código que vai executar as rotinas gráficas na placa de vídeo. Qualquer aplicação que precise acessar estas rotinas de pintura em baixo nível sempre tem que utilizar uma *API* gráfica que realize a comunicação da *CPU* com a *GPU*.

Estas *APIs* possuem um certo grau de estabilidade, em termos de compatibilidade, que garantem que, ao serem lançadas placas gráficas novas, o custo de adaptação do código que executava na placa antiga para um que execute nas placas novas seja mínimo. Esta garantia se dá pelo fato da existência de uma camada de abstração que há entre as aplicações 3D desenvolvidas e o *pipeline* gráfico de rasterização bastante otimizado e estável implementado em *hardware* nas placas de vídeo.

Seguindo essa idéia, o RT² se caracteriza como uma *API* gráfica que implementa, em *software*, um *pipeline* gráfico de renderização utilizando a técnica de *ray tracing* para gerar imagens com efeitos mais realistas em tempo real. Sobre este ponto de performance, é válido ressaltar que, embora o RT² seja capaz de renderizar imagens em resoluções de alta definição, ele dificilmente será capaz de superar a performance do *pipeline* gráfico de rasterização implementado no *hardware* das placas de vídeo.

A grande vantagem do uso do sistema RT² é fazer o balanceamento entre a vantagem da performance presente na rasterização e a melhora na qualidade das imagens geradas usando a técnica do *ray tracing*. Este objetivo é alcançado no RT² através de uma implementação massivamente paralela dos algoritmos do *ray tracing* utilizando *GPGPU* e *CUDA*.

2.5. Point Based Animation – PBA

A implementação da técnica de animação baseada em pontos proposta por [9], bem como suas otimizações, apresenta diversas vantagens que favorecem seu uso em aplicações que desejam retratar com certa fidelidade o comportamento físico dos objetos em um ambiente virtual.

Uma das principais características que permitem este feito é a forte adesão desta implementação aos conceitos da mecânica do contínuo, área da mecânica que prega que os corpos são entidades que podem ser divididas em partículas infinitesimais e que estas partículas sempre mantêm as propriedades físicas se comparadas às propriedades da entidade completa. Esta característica permite descrever objetos completos como um conjunto de partículas discretas sem informação de conectividade entre elas.

3. Bounding Interval Hierarchy

Tendo em mente os conceitos apresentados nos capítulos anteriores, este capítulo explica em detalhes como funciona a *BIH*, começando com uma breve explicação das nuances principais desta estrutura, seguindo com as etapas de construção e travessia, e apresentando as otimizações realizadas nas mesmas. Após a apresentação dos conceitos relativos à *BIH*, será realizada uma breve comparação desta com as estruturas mais utilizadas para o propósito de aceleração da travessia de raios no *ray tracing*. Feito isso, será explicada a integração com o RT^2 e com o *PBA*.

3.1. Conceitos gerais

Assim como as principais estruturas usadas para a aceleração do traçado de raios, a *BIH* também se configura como uma árvore binária. Ela une características da *BVH* e da *KD-Tree*, otimizando o processo de construção da estrutura sem perder eficiência na etapa de travessia. Seu principal diferencial, que torna possível sua implementação para cenas dinâmicas, é a abordagem simplificada na etapa de construção. A *BIH* também é considerada o meio termo entre particionamento de espaço e de objetos, pois utiliza vantagens de cada uma destas abordagens para melhorar seu desempenho e precisão na renderização de ambientes 3D.

A *BIH* foi concebida com o objetivo de minimizar o chamado *time to image*, conseguindo entregar altas taxas de *fps* tanto para cenas estáticas quanto dinâmicas. Para alcançar tal objetivo, a *BIH* simplifica seu algoritmo de construção, fazendo com que o tempo gasto nesta etapa seja realmente pequeno. Esta característica faz da *BIH* uma ótima candidata para implementações em cenas completamente dinâmicas [1].

Apesar de simplificar bastante o algoritmo de construção, a perda de eficiência relativa à etapa de travessia é aceitável se for levado em consideração

o ganho geral representado pela baixa no *time to image*. Algumas características inerentes à própria proposta da *BIH*, aliadas a uma implementação bem otimizada, fazem com que o ganho no tempo total de renderização seja grande.

A seguir, são apresentadas as principais características da *BIH*.

3.1.1. Plano de corte e espaços vazios

Para compensar a falta de heurística no processo de construção da estrutura, o que leva a um maior custo de travessia, a *BIH* implementa um esquema de intervalos na estrutura de cada nó interno.Estes intervalos representam o volume do nó onde existe alguma probabilidade de haver primitivas passíveis de teste de interseção. Diferente das estruturas de particionamento de espaço, cujos nós são totalmente disjuntos, na *BIH* os nós filhos de um mesmo pai podem compartilhar uma área em comum, o chamado *node overlap*.

Inicialmente, os planos de corte candidatos sempre se encontram exatamente na metade da medida do maior eixo da *AABB* que envolve o nó. A distribuição das primitivas em cada lado do nó se dá através de seus centróides, sempre englobando a primitiva por completo.

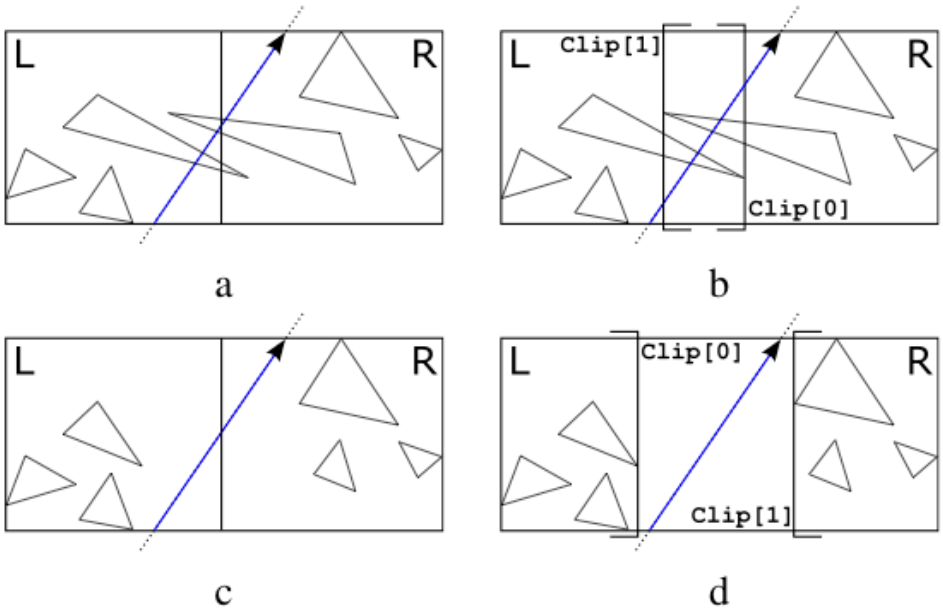


Figura 8. Comparativo entre o plano de corte da *KD-Tree* em a) e c) com os planos de corte da *BIH* em b) e d). No caso da *KD-Tree* as primitivas que são atravessadas pelo plano de corte têm que ser referenciadas por ambos os filhos da esquerda L e da direita R.

Através desta abordagem de intervalos, a *BIH* implementa a busca binária na estrutura, haja visto que cada primitiva é atribuída a somente um dos lados do nó. Caso haja uma primitiva atingida pelo plano de corte e ela esteja no mesmo lado do plano de corte, este é expandido para incluir a primitiva por completo (Figura 8), baseando-se nos valores máximos e mínimos da *AABB* que envolve a primitiva.

Uma das características principais da *BIH* é o modo como esta trata os espaços vazios no ambiente. Inicialmente, na etapa de construção da estrutura, os planos de corte candidatos são definidos. Logo neste passo, é possível notar a presença de nós vazios que são prontamente descartados.

A existência de dois planos de corte abre a possibilidade do surgimento de espaços vazios entre os nós da esquerda e da direita de um mesmo nó pai (Figuras 8-9). Esta característica é bem aproveitada em cenas virtuais onde as primitivas estão distribuídas de forma heterogênea no espaço.

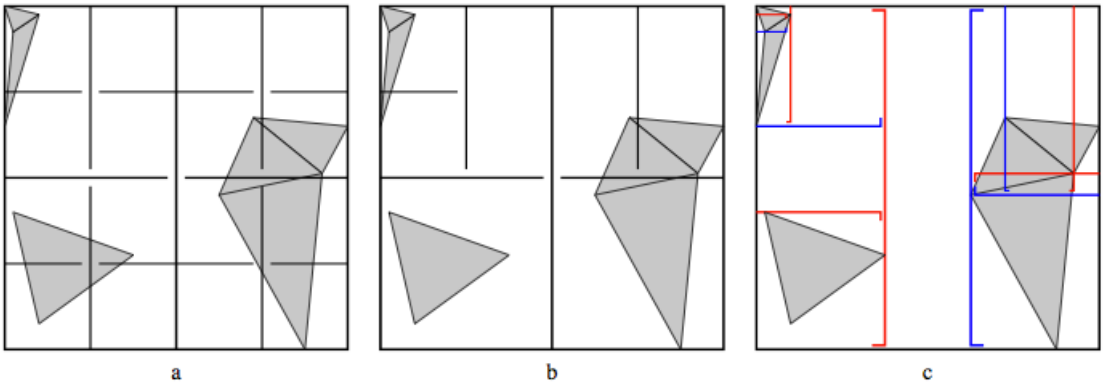


Figura 9. a) Planos candidatos obtidos ao dividir as *AABBs* no maior eixo. b) Nós vazios foram descartados e os candidatos, selecionados. c) Resultado da expansão dos candidatos para englobar as primitivas.

A Figura 10 mostra a *BIH* criada com o modelo *Hornbug*. As regiões azuladas representam as áreas limitadas pelos planos de corte da *BIH* nas folhas da árvore. As áreas mais escuras são as áreas onde há menor densidade de folhas, seja pelo grande aproveitamento de espaços vazios ou pela ausência de primitivas nestas áreas.

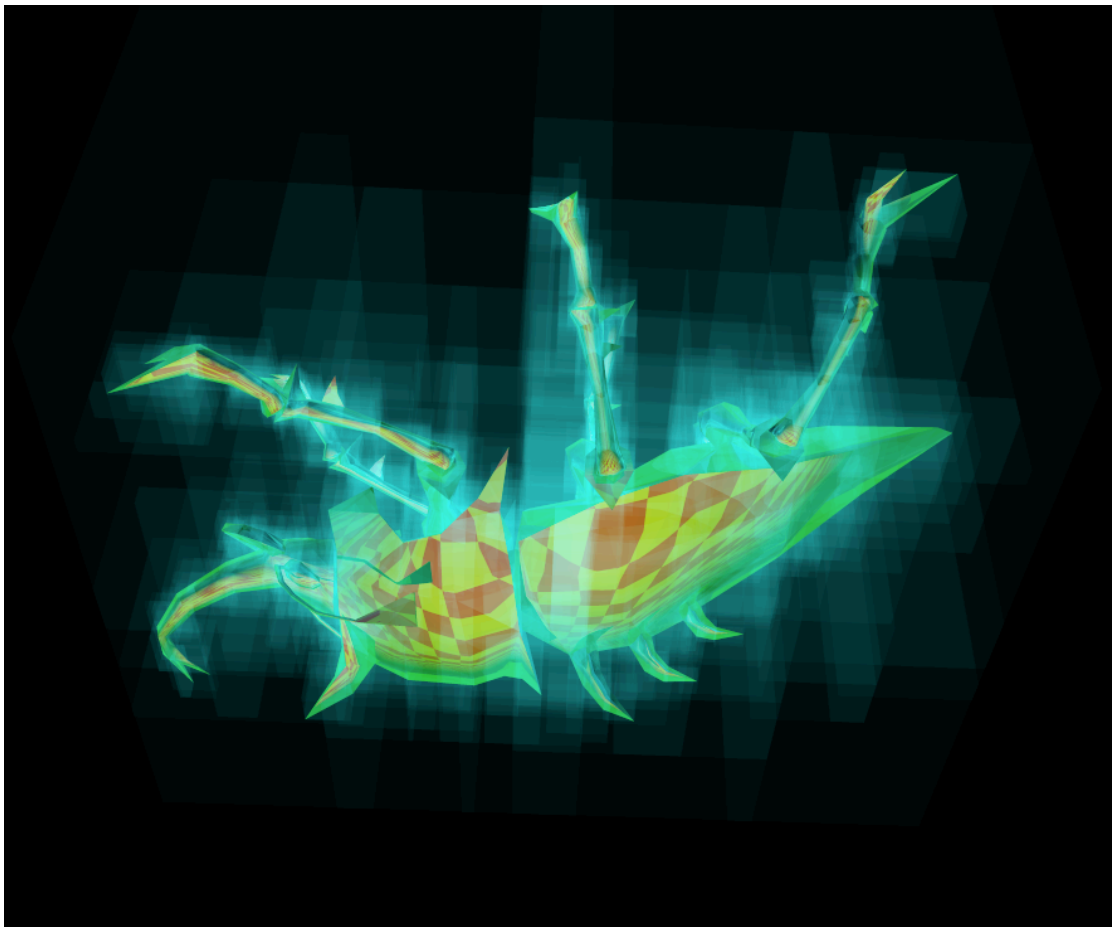


Figura 10. Representação visual da *BIH* (regiões azuladas) sobre o modelo *Hornbug*.

3.1.2. Estrutura dos nós

Os nós internos da *BIH* são compostos pelos dois planos de corte e por um ponteiro para o filho da direita. Este ponteiro incorpora em seu valor a informação do eixo de corte. Esta informação é armazenada nos dois *bits* menos significativos, da seguinte forma: 00 para o eixo X, 01 para o eixo Y, e 10 para o eixo Z. Caso seja um nó folha, isto é facilmente identificável através do valor 11 nos *bits* menos significativos.

No processo de construção, o filho da esquerda é instanciado sempre uma posição após o pai; por isso, não há a necessidade de apontar diretamente o filho da esquerda.

Todos os nós são alinhados em estruturas de 16 *bytes* e sua disposição em relação aos *bytes* varia caso o nó seja interno, ou seja, nó folha. A distribuição dos *bytes* funciona do seguinte modo:

```
typedefstruct {
	/*
	Aponta para o filho da direita.
	Guarda informações do eixo de corte. (00, 01, 10) ou folha (11).
	Caso seja nó folha, aponta para a posição inicial no array de primitivas
	*/
	int indiceFilho; // 4 bytes

	/*
	Não utilizado em caso de nó interno.
	Caso seja nó folha, representa a quantidade de primitivas presente no nó.
	*/
	int itens; // 4 bytes

	/*
	Não utilizado em caso de nó folha.
	Caso seja nó interno, representa os planos de corte da seguinte forma:
	Posição 0 corresponde ao plano de corte da esquerda.
	Posição 1 corresponde ao plano de corte da direita.
	*/
	float planosDeCorte[2]; //8 bytes
} BIH_Node;
```

3.1.3. Ordenação de primitivas:

O algoritmo executado na etapa de construção da *BIH* é de fato um algoritmo de ordenação com uma estrutura idêntica à de um *quicksort* [1] que, consequentemente, executa em $O(n \log n)$. Este algoritmo de ordenação age em cima do *array* de primitivas ao mesmo tempo em que a estrutura que representa a *BIH* propriamente dita é criada. O objetivo principal do uso deste algoritmo quando da construção da árvore é assegurar que, dado um nó e um nível da árvore, todas as primitivas que estão localizadas à esquerda do pivô no *array* de primitivas, em relação ao espaço, sejam, de fato, menores que o pivô.

Esta comparação de localização espacial entre as primitivas sempre se dá em torno do eixo escolhido. Como explicado nas seções anteriores, o eixo de corte escolhido é sempre o que possui a maior extensão. O valor considerado nesta comparação é o centro da *AABB* que engloba a primitiva que está sendo iterada.

A idéia de usar as *AABBs* que envolvem as primitivas em vez delas mesmas aumenta a velocidade do processo de construção em cerca de duas a três vezes [1]. Embora esta abordagem possa gerar árvores menos otimizadas, a perda de performance relativa à etapa de renderização é quase nula. Do mesmo modo, as árvores geradas deste modo possuem altura e profundidade equivalentes.

3.1.4. Precisão numérica:

Técnicas de particionamento do espaço requerem operações matemáticas específicas de interseção entre as primitivas e os planos de corte e *AABBs*. O uso deste tipo de operação abre margem à problemas de imprecisão de ponto flutuante. Para que os algoritmos implementados pelas estruturas sejam executados de forma estável, cuidados adicionais têm que ser tomados, o que leva a uma perda de performance [1].

O algoritmo implementado na construção da *BIH* utiliza apenas as informações de coordenadas das *AABBs* sobre os eixos canônicos e operações de máximo e mínimo para fazer os cálculos da construção. Esta abordagem torna a estrutura mais robusta e estável, além de não sofrer perda de performance com os cálculos extras de multiplicação e divisão sobre ponto flutuante, que dão margem à imprecisão.

3.2. Construção

Esta é a etapa mais importante na implementação da *BIH* e, das otimizações e novas abordagens propostas por esta nova estrutura, esta é a que mais contribui para o alto desempenho final de renderização. Neste tópico serão explicados os detalhes do algoritmo de construção e o porquê da relativa grande performance nesta etapa.

Inicialmente, para cada primitiva presente na cena, são realizadas operações de máximo e mínimo sobre seus vértices para que seja calculada a mínima *AABB* que engloba a primitiva. Ao mesmo tempo, também é calculada a *AABB* da *BIH* como um todo. Esta *AABB* geral da cena engloba todas as primitivas e ela é utilizada logo no início do processo de travessia, de modo que raios que

não intersectam a *AABB* geral da cena são prontamente descartados e nem iniciam a visita aos nós da árvore. O resultado desta etapa será utilizado posteriormente para a comparação e ordenação do *array* de primitivas e otimização dos planos de corte na etapa de subdivisão.

No início da subdivisão, o plano de corte ou *split* é computado como sendo a mediana entre o máximo e o mínimo do maior eixo da *AABB*. Os planos de corte esquerdo e direito, que chamaremos de *clipL* e *clipR* respectivamente, são inicializados com os valores mínimo e máximo, respectivamente, sobre o eixo de corte. Em seguida, é feita uma iteração sobre cada primitiva do nó atual do seguinte modo:

- Calcula-se o centro da *AABB* que envolve a primitiva e verifica-se se o centro desta se encontra do lado esquerdo ou direito do *split*.
- Caso esteja no lado esquerdo e o valor máximo da *AABB* da primitiva atual seja maior do de *clipL*, este é substituído pelo valor máximo da *AABB* atual.
- Caso esteja no lado direito de *split*, a primitiva atual é trocada com a primitiva mais à direita na lista e a verificação é feita novamente, com a nova primitiva. Do mesmo modo, o valor de *clipR* é substituído pelo valor mínimo da *AABB* atual, caso este seja menor que *clipR*.
- Ao mesmo tempo, calcula-se a mínima *AABB* que engloba todas as primitivas deste nó, que chamaremos de *nodeNewBox*.

Em seguida, é realizada uma checagem que é implementada na *BVH*. Esta implementação adicional funciona como otimização em cima da estrutura original proposta por [1]. A idéia é que, como a *BIH* só especifica os planos de corte a partir do centro da *AABB* do nó, as primitivas podem estar concentradas em apenas uma área da *AABB*, fazendo com que boa parte do nó se configure como espaço vazio.

O objetivo desta otimização é verificar através da *nodeNewBox* a relação entre o tamanho ocupado pelas primitivas e o tamanho total do nó sobre o eixo de corte. Caso a dimensão da *nodeNewBox* multiplicada por um fator de 1,3

(definido através de testes empíricos) seja menor do que a *AABB* do nó, este será iterado novamente levando em consideração a *nodeNewBox* como sendo a *AABB* do nó, pois há uma grande probabilidade de que o plano de corte escolhido não tenha sido bom o suficiente e pode levar a uma defasagem na performance de renderização.

Caso todas as primitivas se encontrem do lado esquerdo e o valor de *split* do eixo de corte seja o mesmo da divisão do nó do nível anterior, a recursão é finalizada criando-se um nó folha contendo as primitivas da iteração atual. Em seguida, já que todas as primitivas se encontram do lado esquerdo, a *AABB* do nó atual é reduzida pela metade, tendo o seu máximo limitado a *split* sobre o eixo de corte e o nó atual é iterado novamente para encontrar um melhor plano de corte. Seguindo a mesma idéia, a mesma checagem é realizada caso todas as primitivas se encontrem do lado direito da *split*.

Caso existam primitivas dos dois lados do plano de corte, significando que realmente o algoritmo está progredindo em dividir os objetos, os nós da esquerda e direita são alocados. Em seguida, o nó atual é atualizado com a referência para o nó filho e com as informações de eixo de planos de corte, seguindo com a recursão sobre o filho da esquerda e da direita.

Como explicado anteriormente, este algoritmo funciona exatamente da mesma forma que um algoritmo de *quicksort*. Uma das condições de parada para a construção recursiva da árvore é justamente a mesma condição de parada do algoritmo *quicksort*, que é quando o intervalo de ação do algoritmo sobre o *array* de primitivas é menor ou igual a um.

No caso da *BIH*, isto é tratado de um modo um pouco diferente, pois como foi explicado, o nó folha da *BIH* pode conter mais de uma primitiva. Sendo assim, a condição de parada do algoritmo *quicksort* da *BIH* se dá quando o número de primitivas no nó chega a um valor menor que o máximo permitido por nó, ou quando a profundidade máxima da árvore é alcançada.

3.2.1. Detalhes de implementação

Na seção anterior, foram vistos os detalhes de como funciona a etapa de construção da *BIH*, incluindo as escolhas de plano de corte, compressão e expansão destes, casos especiais de particionamento e algumas otimizações.

O algoritmo completo, desde a chamada a partir do RT², até o fim da construção da estrutura com a efetivação das condições de parada explicadas na seção anterior, foi dividido em quatro partes.

A primeira fase é chamada pela classe `CUDATracerThread` do RT², na execução do método `update` desta classe, no momento exatamente anterior à renderização da cena, onde a malha de primitivas é atualizada pelo sistema *PBA* e os dados copiados para a *GPU* (Algoritmo 1).

Algoritmo 1:

```
void CUDATracerThread::update() {
        Inicializar vetor de normais associadas a cada vértice;
        Atualizar malha do objeto com base na simulação física;

        Para cada triangulo presente na malha {
                Calcular a normal do triangulo e normalizar;
                Somar a normal do triângulo à normal resultante de cada vértice;
        }

        Calcular parâmetros de interseção dos triângulos;
        Reconstruir a estrutura da BIH; // BIH::build
}
```

Algoritmo 2:

```
void BIH::build() {
        Inicializa o array que contém as AABBs das primitivas;
        Inicializa a AABB global da cena como inválida;

        Para cada primitiva presente na malha {
                Calcular AABB da primitiva e guardar no array de AABBs;
                Recalcular a AABB global para conter a primitiva atual;
        }

        Construir árvore; //BIH::buildHierarchy
}
```

Em seguida, o processo de construção da *BIH* é iniciado, onde é feita a inicialização de todas as estruturas necessárias e a computação das *AABBs* das primitivas e da cena completa (Algoritmo 2).

Após esta etapa de preparação, inicia-se de fato o processo de construção, onde é alocado espaço para o nó raiz e feita a preparação para o início da recursão (Algoritmo 3).

Algoritmo 3:

```
void BIH::buildHierarchy() {
        Insere o nó raiz no array de nós;

        Inicializar a AABB do nó atual igual à AABB global;

        //Início da recursão
        Subdividir(0, TotalDePrimitivas – 1, AABB do nó atual);
}
```

Finalmente, inicia-se o processo de subdivisão dos nós através do uso da recursão e implementação do algoritmo *quicksort* (Algoritmo 4).

Algoritmo 4:

```
void BIH::subdivide(int left, int right, AABB nodeBox) {
        Se (right – left + 1) for menor que o máximo permitido OU
        A profundidade máxima da árvore foi alcançada {
                Criar nó folha;
                Retornar;
        }

        Inicialize rightOriginal com o valor de right;
        Inicialize o plano de corte esquerdo como +INFINITY;// clipL
        Inicialize o plano de corte direito como –INFINITY;// clipR
        Inicialize intervalo esquerdo do nó como +INFINITY;// nodeL
        Inicialize intervalo direito do nó como -INFINITY;// nodeR

        Calcule o valor da mediana da AABB no maior eixo e guarde em split;
        Para cada primitiva do intervalo [left, right] {
                Se o centro da AABB da primitiva no eixo de corte for menor que
                split {
```

```
                    clipL = mínimo(clipL, máximo da AABB da primitiva);
                } caso contrario {
                    Move a primitiva atual para o fim do intervalo [left, right];
                    Decrementa right de 1;
                    clipR = máximo(clipR, mínimo da AABB da primitiva);
                }
                nodeL = mínimo(nodeL, mínimo da AABB da primitiva);
                nodeR = máximo(nodeR, máximo da AABB da primitiva);
        }

        Se a distância entre nodeL e nodeR * 1,3 for menor que o tamanho da
        AABB do nó {
            Modifique a AABB do nó atual para os novos limites nodeL e nodeR;
            Iterar novamente o nó atual;
        }

        Se todas as primitivas estiverem do lado esquerdo de split {
            Atribua o valor de split ao valor máximo da AABB do nó atual;
            Iterar novamente o nó atual;
        } se não se todas as primitivas estiverem do lado direito de split {
            Atribua o valor de split ao valor mínimo da AABB do nó atual;
            Iterar novamente o nó atual;
        } se não {
            Crie os nós da esquerda e direita no array de nós;
            //recursão
            Subdividir(left, right, AABB do nó da esquerda);
            Subdividir(right + 1, rightOriginal, AABB do nó da direita);
        }
}
```

3.3. Travessia

Um diferencial crucial do processo de travessia da *BIH* que afeta positivamente a performance é que, devido ao fato da construção da estrutura ser um algoritmo de ordenação no espaço, é possível saber qual o objeto que está mais próximo da câmera, dependendo da direção do raio. Com essa característica, a travessia da *BIH* se torna praticamente idêntica à da *KD-Tree*.

Devido à sua abordagem de utilização de planos que definem intervalos válidos 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. A utilização deste caso espacial é bom do ponto de vista 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, poupando memória.

Como foi dito, os volumes que representam os nós da árvore podem se sobrepor (Figura 2). Sendo assim, a travessia não é finalizada enquanto não for encontrada uma interseção. Tão logo uma interseção é encontrada, os ramos da árvore que representam primitivas que estão mais longe do que a interseção atual podem ser podados para evitar visitas desnecessárias aos nós. Devido ao fato dos nós poderem se sobrepor, todos os nós empilhados têm que ser visitados antes do fim da recursão.

Estas abordagens implementadas no processo de travessia da *BIH* fazem com que a perda de performance na travessia proveniente de um algoritmo de construção simplificado e praticamente sem heurística seja minimizada.

Inicialmente, o processo se dá quando o raio atual é computado e passado para a estrutura. Acompanhando esta chamada à estrutura, vão o raio que foi definido para a travessia e os parâmetros T, U e V de interseção, que se configuram como parâmetros de saída do algoritmo. O parâmetro T representa a distância paramétrica entre o ponto de partida do raio e o ponto intersectado dentro da estrutura. Caso não seja encontrada nenhuma interseção, é retornado o valor INFINITY. Os parâmetros U e V representam as coordenadas baricêntricas [19]do ponto de interseção dentro da primitiva. É com base nestes parâmetros de interseção que é realizado o processo de *shading*.

Como início do processo de travessia, o raio é testado contra a *AABB* que envolve toda a *BIH*. Como resultado deste teste, temos 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 a caixa (Figura 11).

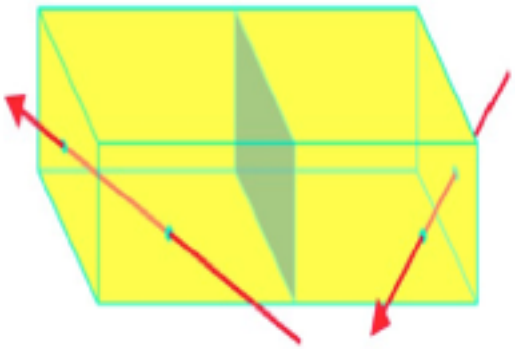


Figura 11. Os dois pontos de interseção do raio com a caixa são retornados e determinam o intervalo válido de valores para o parâmetro T.

Caso o raio seja originado dentro da *AABB*, ou seja, quando a câmera se encontra dentro da estrutura ou quando algum objeto dentro da estrutura tem propriedades reflexivas ou refrativas, o valor paramétrico *tMin* recebe um valor negativo. Após o teste de interseção com a *AABB* geral, 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 a 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 *shading* corretamente.

3.3.1. Detalhes de implementação

Na seção anterior, foram vistos os detalhes do processo de travessia de um raio dentro da estrutura da *BIH*, incluindo as condições de saída, definição de modo de empilhamento de nós e interseção com primitivas.

O algoritmo completo, desde a chamada a partir do RT² até a obtenção dos parâmetros de saída e do índice da primitiva intersectada, foi dividido em três partes.

A primeira etapa é constituída pela chamada do método `trace` da classe `CUDATracerThread` do RT², onde toda a estrutura necessária para fazer o traçado dos raios é montada e as informações são transferidas para a *GPU* (Algoritmo 5).

Algoritmo 5:

```
void CUDATracerThread::trace() {
    CUDATracerThread::update(); //atualiza a malha do objeto e recria a BIH

    Copia valores da posição da câmera para a GPU;
    Copia valores da resolução da tela para GPU;

    Dependendo do modo de travessia, invoque o kernel correto;
    Espere a execução do kernel acabar;
}
```

Em seguida, é executada a função externa `traceSingleGPU` que vai repassar as informações e invocar a execução do *kernel* correto, baseado no tipo de travessia selecionado (Algoritmo 6).

Algoritmo 6:

```
void traceSingleGPU() {
        Calcule o ponto de origem e a direção do raio;
        Aloque a pilha de execução usada na travessia;
        Aloque os parâmetros T, U e V;

        //invoca a função rayTraversao
        rayTraversal(raio, t, u, v);
}
```

Em seguida, é executada a travessia propriamente dita na *BIH*. Como foi explicado na seção anterior, os parâmetros de retorno são encontrados após a execução deste passo (Algoritmo 7).

Algoritmo 7:

```
unsigned int rayTraversal(Ray r, float T, float U, float V) {
        Inicializa tMin e tMax como sendo as distâncias paramétricas de
        interseção do raio com a AABB geral;

        Se tMin< 0 { //raio originado do interior da caixa
                tMin = 0;
        }

        Insere o nó raiz na pilha;

        Faça {

                Enquanto não encontrar uma folha {
                        Se encontrou espaço vazio entre os nós {
                                Interrompa o loop atual;
                        }
                        Se o raio atravessa apenas o nó mais distante da origem do
                        raio {
                                Pula para este nó e o processa;
                        }
                        Se o raio atravessa apenas o nó mais próximo da origem do
                        raio {
                                Pula para este nó e o processa;
                        }

                        Se o raio atravessa os dois nós {
```

```
                        Empilha o nó mais distante;
                        Processa o nó mais próximo;
                }
        }

        //Encontrou uma folha
        Para cada primitiva do nó atual {
                Testar interseção com a primitiva atual;
                Guardar a interseção mais próxima até o momento;
        }

        Desempilhar o nó;

        Caso a pilha esteja vazia, retorne o valor da interseção mais
        próxima, os parâmetros T, U e V e o índice da primitiva
        intersectada;

    } até sair;
}
```

3.4. Análise Comparativa com Outras Estruturas

Nesta seção é feita uma análise de vantagens e desvantagens da *BIH* em relação à *BVH* e à *KD-Tree*. Esta análise leva em consideração alguns pontos principais que concernem a utilização dessas estruturas em um sistema de *ray tracing*.

3.4.1. BIH vs BVH

- Estrutura no nó:

  Neste ponto, a *BIH* leva vantagem devido à sua maior simplicidade na estruturação dos nós de sua árvore. Ao contrário da *BVH*, a *BIH* não armazena uma *AABB* completa para cada nó, que significa uma economia de 24 *bytes* (coordenadas *x, y* e *z* para os pontos mínimo e máximo da *AABB*). Além disso, a *BIH* consegue fazer um reaproveitamento dos *bytes* utilizando-os de modo diferente caso o nó seja interno, ou seja, folha.

- Utilização de memória:

  Devido ao armazenamento de uma *AABB* para cada nó da *BVH*, esta estrutura requer uma grande quantidade de memória para

guardar a estrutura toda. A *BIH* se apresenta vantajosa neste ponto dada a sua simplicidade de armazenamento dos nós.

- Construção da estrutura:

  O algoritmo da *BIH* é de fato um *quicksort*. Ao construir a estrutura, o algoritmo vai ordenando as primitivas no espaço. Por outro lado, o uso exaustivo de *AABBs* durante a construção da *BVH* permite que a estrutura seja criada de um modo mais otimizado, maximizando os espaços vazios e aumentando o desempenho.

- Travessia de raios:

  Este processo se torna mais eficiente e vantajoso na *BIH* devido à ordenação das primitivas, pois é possível de imediato saber qual primitiva está mais perto da origem do raio, já que as primitivas estão ordenadas.

3.4.2. BIH vs KD-Tree

- Estrutura do nó:

  Neste ponto, a *KD-Tree* se apresenta como mais vantajosa. Uma implementação bem otimizada desta estrutura só precisa de 8 *bytes* para representar o seu nó, enquanto na *BIH* são necessários 16 *bytes*.

  Em contrapartida, a *BIH* tem a vantagem de especificar dois planos de corte para cada nó, podendo existir espaços vazios entre os nós, o que não acontece na *KD-Tree*.

- Utilização de memória:

  Nesta característica, a *BIH* leva uma grande vantagem. Primeiramente, por se tratar de um esquema de particionamento de espaço, a *KD-Tree* tem que tratar os casos onde o plano de corte intersecta a primitiva, recalculando as *AABBs* mínimas para cada lado do plano e armazenando-as, além da necessidade de referenciar a primitiva por ambos os nós, replicando informação. Mesmo havendo casos de nós vazios na *KD-Tree*, sempre é necessário instanciá-los.

- Construção da estrutura:

  O processo de construção da *BIH* se mostra muito mais eficiente por diversos motivos. Primeiramente, não é necessário recalcular *AABBs* quando o plano de corte atravessa uma primitiva. A inexistência de uma heurística como a *SAH* faz com que a construção seja bastante eficiente na *BIH*.
- Travessia de raios:

  O processo de travessia é quase idêntico nas duas estruturas, mas a *BIH* leva uma vantagem no que diz respeito a nós vazios. A *KD-Tree* exige que os nós vazios sejam armazenados e visitados quando da travessia, o que não acontece na *BIH*.

  Em compensação, o uso da *SAH* na construção da *KD-Tree* leva a árvores de altíssima qualidade e a tempos de travessia bastante baixos, implicando em uma performance superior à alcançada na *BIH*.

3.5. Integração com o RT²

Como foi visto no capítulo 2, o RT² contempla vários tipos de travessia em cima da estrutura *KD-Tree*. O objetivo principal da integração da *BIH* com o sistema RT² é incluir esta estrutura como mais uma possibilidade de travessia mantendo a transparência e rapidez na troca do modo de travessia na aplicação.

Inicialmente, foi necessário estudar a arquitetura e funcionamento do sistema para então analisar os pontos de integração e os impactos que esta pode oferecer. O pleno entendimento das classes e relacionamentos entre elas, bem como o fluxo da aplicação, desde o carregamento dos dados do objeto até o fim da aplicação foi de extrema importância para a realização do objetivo da integração.

A classe *Scene* pode ser considerada a principal entidade do sistema. Ela representa o ambiente virtual 3D e armazena todas as informações de primitivas e objetos, bem como as estruturas de dados, informações de travessia, etc. Na etapa de inicialização, esta classe é responsável por criar a janela *DirectX* [14]

onde será projetado o ambiente virtual e também por controlar os eventos de teclado e *mouse*.

Em seguida são criadas as estruturas de dados e inicializados os modos de travessia. Nesta etapa foi adicionada a criação e inicialização da *BIH*. Neste ponto, o algoritmo de construção das estruturas é executado em cima da posição inicial da malha do objeto. Após esta etapa, a classe *Scene* lança as *threads* responsáveis pela renderização.

Ao chamar o método *trace* da classe *CUDATracerThread*, um parâmetro é passado indicando o modo de travessia selecionado. Neste ponto, a *BIH* foi adicionada como mais um valor possível para o parâmetro, e a travessia dos raios é feita normalmente, sem afetar outras etapas como inicialização e *shading*.

Com estas alterações foi possível comparar com clareza as vantagens e desvantagens das duas estruturas em relação a vários critérios como desempenho de renderização, custo de travessia, utilização de memória e tempo de construção. Estes dados estão expostos no capítulo de resultados.

3.6. Integração com o PBA

O objetivo principal desta integração é poder mostrar o poder da *BIH* como estrutura de aceleração para cenas muito dinâmicas, dado o grande desempenho desta estrutura na etapa de construção da árvore. Porém, para executar esta etapa com sucesso, foi necessária muita cautela. Como o algoritmo do *PBA* faz alteração nos vértices e triângulos da malha do objeto, uma integração mal feita poderia levar a problemas na etapa de renderização do objeto.

Foi implementado um compartilhamento da estrutura que comporta as primitivas do objeto entre o algoritmo de *PBA* e o algoritmo de renderização. Através desta abordagem, foi possível que, antes da renderização de cada *frame*, fosse possível ativar o algoritmo *PBA* para realizar a alteração da malha do objeto. Garantir esta condição foi de extrema importância na criação de uma *BIH* consistente com o estado atual da malha do objeto.

A animação propriamente dita é realizada através da chamada do método *update* do objeto da classe *VisualObject* presente no módulo *PBA* [9]. Na

inicialização do *VisualObject*, este carrega dados pertinentes ao cálculo da física envolvida no processo como os *phyxels* e *surfels*, tais como coeficientes de elasticidade, gravidade, etc. [8].

A implementação do *PBA* usada nesta etapa da integração possui uma performance elevada, se comparada a implementações padrões em *CPU*, pois o algoritmo foi massivamente paralelizado e executado sobre *CUDA*.

Para o entendimento da implementação do *PBA*, é necessário saber as principais entidades que estão relacionadas ao processo de simulação física: os *phyxels*, *surfels*, *body*, *material* e *surface*.

Os *phyxels*, juntamente com os *surfels* representam as menores unidades que compõem o objeto simulado. Os *phyxels* são os responsáveis por aplicar força sobre os objetos da malha, fazendo com que esta se modifique. Eles também possuem uma área de atuação onde pode-se aplicar força, sendo esta área variável.

O *body* representa um grupo de *phyxels* e os relacionamentos de vizinhança entre eles. O *body* contém a implementação de uma *spatial hash* para prover acesso constante aos *phyxels* vizinhos.

O *material* define atributos relativos à mudança na forma dos objetos simulados. A utilização de coeficientes de tensão e deformação, bem como as constantes relativas a este tipo de cálculo, são controladas pelo *material*.

A *surface* é semelhante ao *body* pois também trata um agrupamento com *spatial hash*, mas não de *phyxels* e sim de *surfels*. A *surface* é responsável por planejar e executar a etapa de simulação da física e de atualização da malha de triângulos com base na movimentação dos elementos do *PBA*.

4. Resultados

Todos os resultados apresentados neste capítulo foram obtidos através de testes realizados em um processador Core i7 3.2GHz com 8 GB de RAM DDR3 e duas placas de vídeo GTX480 (para os testes, só foi utilizada uma das placas) e sistema operacional Windows 7 Professional.

Cabe ressaltar que os testes foram realizados executando os algoritmos de construção das estruturas em *CPU* enquanto que os de travessia são executados em *GPU*.

Os dados apresentados foram obtidos através de testes realizados em vários modelos de objeto. São eles:

- *Hornbug* (ver Tabela 1) – 1,8 mil triângulos;
- *Alien* (ver Tabela 1) – 32 mil triângulos;
- *Bunny* – Representa o modelo *Stanford Bunny* [23]. Contém 69 mil triângulos;
- *Dino* (ver Tabela 1) – 107 mil triângulos;
- *Dragon* – Representa o modelo *Stanford Dragon* [23]. Contém 847 mil triângulos.

4.1. Aumento de Performance

Uma boa escolha sobre o número máximo de primitivas por nó folha pode mudar drasticamente a performance da estrutura. Isto se dá pelo fato de que, quanto maior o número máximo de primitivas por folha, menos profundas serão as sub-árvores, levando a um tempo de travessia menor e a um maior desempenho.

Entretanto, este valor tem que ser escolhido com cautela. Ao definir que um nó responde pela interseção de mais de uma primitiva, é necessário que, ao visitar este nó no processo de travessia, realize-se um teste de interseção para

cada primitiva contida no nó até que seja encontrada a primitiva intersectada que esteja mais próxima da origem do raio.

A distribuição de valores para o número máximo de primitivas por nó nos modelos *Hornbug*, *Alien* e *Dino* afeta o desempenho geral do processo de renderização, que inclui tempo de construção da *BIH* e tempo de travessia (Gráfico 1).

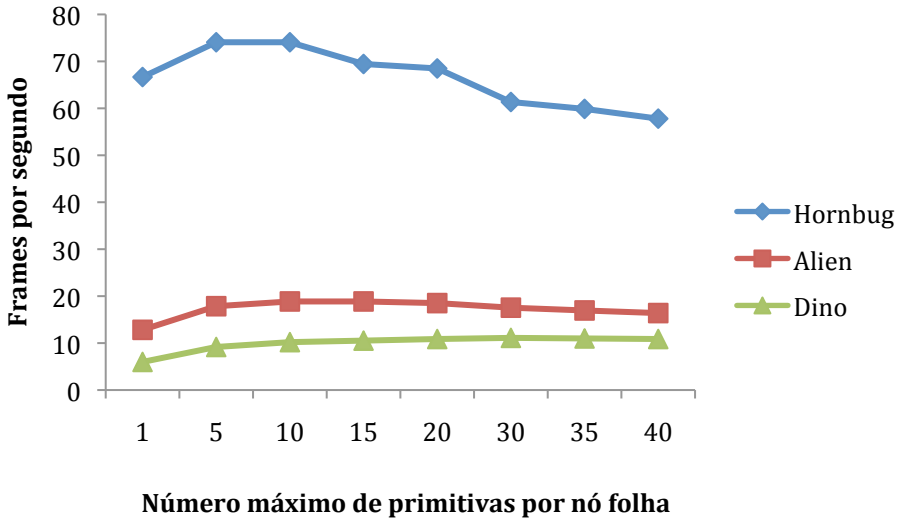


Gráfico 1. Através de diferentes valores para o número máximo de primitivas por nó folha, o desempenho geral do processo é afetado devido ao balanceamento do custo de navegação do nó e do custo de interseção das primitivas.

Como é possível observar, o ganho de performance varia bastante de acordo com o número de triângulos presente no objeto. Com aumentos que chegam a passar de 80%, este se mostra como um parâmetro decisivo na performance final da renderização.

O *Dino* foi o modelo que apresentou o maior aumento de performance ao passar de 5,9 *fps* (máximo de 1 primitiva por nó folha) para 11,1 *fps* (máximo de 30 primitivas por nó folha), representando um ganho de 85,7% no melhor caso. Em seguida, o *Alien* apresentou um ganho de 47,1% quando passou de 1 para 10 o número máximo de primitivas por nó folha. O *Hornbug* foi o que obteve o menor ganho dentre os 3 modelos testados, melhorando sua performance em 11,1% ao aumentar de 1 para 5 o número máximo de primitivas por nó folha.

Podemos notar também que, como o processo de construção da estrutura da *BIH* é executado a cada *frame* desde o ponto inicial e criando uma árvore totalmente nova, este parâmetro pode ser ajustado em tempo real. O uso de algum tipo de heurística temporal sobre os valores deste parâmetro, que levam em consideração a quantidade de triângulos da cena, podem fazer com que os ganhos de performance sejam sempre maximizados.

4.2. Consumo de Memória

A escolha do valor para o número máximo de primitivas por nó folha é um parâmetro que influi diretamente no consumo de memória. Isto ocorre pois, como foi explicado anteriormente, a árvore gerada é afetada diretamente, tornando-a mais rasa ou mais profunda, conforme o valor escolhido seja maior ou menor, respectivamente.

Tendo em vista este critério, foi realizado um teste de consumo de memória entre as estruturas *BIH* e *KD-Tree* em diversos cenários. Para a primeira estrutura, o algoritmo de construção utilizado em cada cenário levou em consideração diferentes valores para o parâmetro de número máximo de primitivas. No caso da *KD-Tree* foram usadas duas abordagens de construção: *Standard* e *Ropes++* [10]. A primeira abordagem representa a criação e travessia padrão da *KD-Tree*. Já a segunda, representa uma versão mais aprimorada onde não há o uso de pilha no algoritmo de travessia (*stackless* [4]) e possui um alto desempenho final de renderização.

Para o teste de consumo de memória, foram usados os modelos *Dragon*, *Dino*, *Bunny* e *Alien* para realizar a coleta das informações. Os dados referentes a este teste estão expressos no Gráfico 2.

As diferentes abordagens de construção da *KD-Tree* resultam em consumos de memória totalmente divergentes. Uma grande desvantagem relacionada à alta performance de travessia da abordagem *Ropes* é que para obter uma travessia eficiente, o consumo de memória é aumentado drasticamente. É possível observar no Gráfico 2 que a abordagem *Ropes* chega a consumir mais de duas vezes o total de memória que é consumido pela abordagem *Standard* (41MB na *Ropes* contra 14MB na *Standard*).

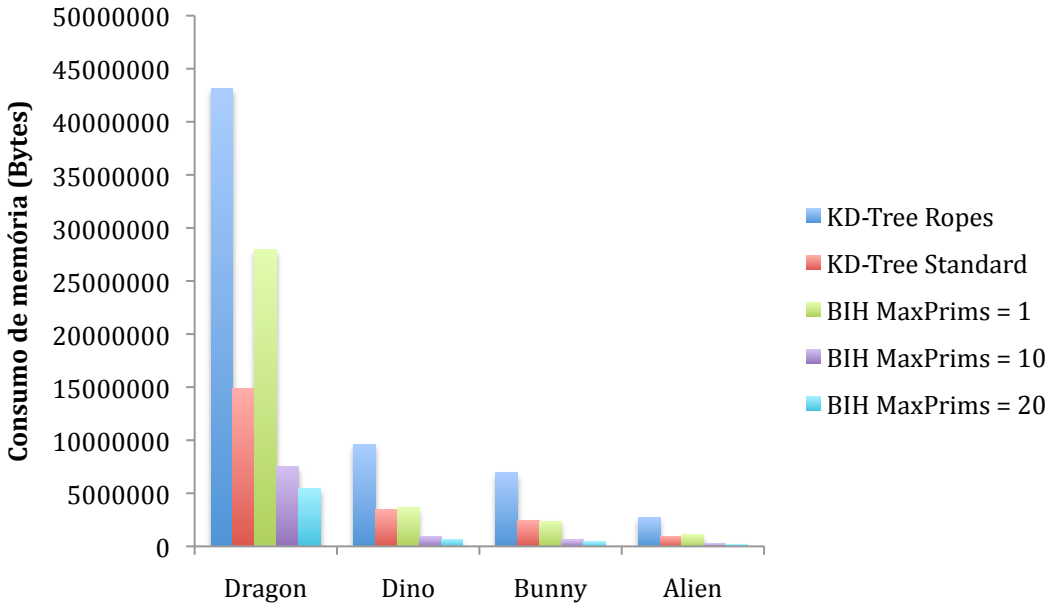


Gráfico 2. Consumo de memória variando de acordo com o modelo do objeto. Em todos os casos, a *BIH* se mostrou com o melhor desempenho em questão de consumo de memória. Quanto maior o número máximo de primitivas por nó, menor o consumo de memória.

Esta disparidade está diretamente relacionada com o fato da abordagem *Ropes* não utilizar pilha em seu algoritmo de travessia. Para ser um algoritmo *stackless* a construção tem que funcionar de um modo diferente: cada nó folha aponta para o nó que se encontra adjacente a cada lado de sua *AABB*. Ao encontrar um nó folha e não intersectar nenhuma de suas primitivas, o algoritmo de travessia pula para o nó vizinho através dessas referências, não havendo a necessidade de empilhamento dos nós.

A abordagem *Ropes* torna a travessia mais eficiente, porém a existência de 6 ponteiros para *AABBs* dentro da estrutura do nó aumenta consideravelmente o consumo de memória.

Neste quesito, a *BIH* supera com larga vantagem estruturas como a *KD-Tree*. A simplificação de seus nós, assim como a existência de espaços vazios entre eles evita a criação de mais nós. No pior cenário (modelo *Dragon* com o máximo de 1 primitiva por nó folha), a *BIH* conseguiu uma diminuição do consumo de memória de 35,2% em relação à *KD-Tree Ropes*. A existência de *SAH*

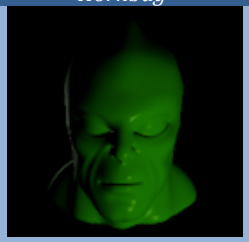
e planos de corte ótimos fez com que a *KD-Tree Standard* superasse a *BIH* neste cenário.

Por outro lado, no melhor cenário (modelo *Dragon* com 20 primitivas por nó folha), a *BIH* demonstrou uma grande vantagem sobre as duas abordagens da *KD-Tree*. A diminuição em relação à abordagem *Standard* foi de 63,1% enquanto a redução de consumo comparada à *Ropes* foi de 87,2%, levando a uma economia de quase 36MB de memória.

4.3. *Time To Image*

Por último, mas não menos importante, foram feitos testes relacionados ao desempenho final da renderização. Este conceito está relacionado à capacidade da aplicação de produzir uma alta ou baixa interatividade.

Tabela 1. Análise comparativa do tempo de construção e travessia da *BIH* comparado ao método *Ropes++* [10]. A performance de construção da *BIH* se mostra bastante superior, enquanto o tempo de travessia é menor na *KD-Tree*.

		Tempo de Construção (*ms*)	Tempo de Travessia (*ms*)	***Time To Image*** (*ms*)
 *Hornbug*	BIH MaxPrims = 1	1,7	15,0	**16,7**
	BIH MaxPrims = 5	0,8	13,5	**14,3**
	BIH MaxPrims = 10	0,6	13,5	**14,1**
	KD-Tree Ropes++	12,3	7,2	**19,5**
 *Alien*	BIH MaxPrims = 1	30,5	47,5	**78,0**
	BIH MaxPrims = 5	14,5	41,5	**56,0**
	BIH MaxPrims = 10	11,7	41,3	**53,0**
	KD-Tree Ropes++	180,0	36,0	**216,0**
 *Dino*	BIH MaxPrims = 1	120,0	47,0	**167,0**
	BIH MaxPrims = 5	67,0	42,0	**109,0**
	BIH MaxPrims = 10	54,0	44,0	**98,0**
	KD-Tree Ropes++	672,0	22,3	**694,3**

Os dados estão expressos em *ms* (milissegundos) para efeitos de comparação, embora seja bastante comum a representação de performance de renderização (*time to image*) através da unidade *fps* (*frames* por segundo). Para converter de *ms* para *fps*, basta dividir 1000 pelo valor expresso na coluna *Time To Image*. Para a coleta de dados relativos à performance da renderização foram utilizados os modelos *Hornbug*, *Alien* e *Dino*. Os dados obtidos estão expressos na Tabela 1.

Analisando os dados presentes na Tabela 1 fica clara a eficiência da *KD-Tree Ropes++* na etapa de travessia. Como explicado anteriormente, a presença de *SAH*, a travessia *stackless* e uma implementação bastante otimizada fazem com que seja praticamente impossível a superação da *BIH* sobre a *KD-Tree* em relação ao tempo de travessia.

No cenário que mais favorece a *KD-Tree* (*Ropes++* e *BIH* MaxPrims = 1 com o modelo *Hornbug*), esta se apresenta com uma vantagem de 52% na performance de travessia. No cenário que menos favorece a *KD-Tree* (*Ropes++* e BIH MaxPrims = 10 com o modelo *Alien*), esta demonstra um ganho de 12,8% sobre a *BIH*. A desvantagem da *BIH* em relação ao tempo de travessia é perfeitamente aceitável se levarmos em consideração a simplicidade do seu algoritmo de construção.

A construção da *BIH* é realmente seu ponto forte. A falta de heurísticas como a *SAH*, cálculos de replicação de *AABBs* e de referência a objetos, entre outras características, tornam esta estrutura insuperável em relação à eficiência da construção.

No pior cenário para a *BIH* (*Ropes++* e BIH MaxPrims = 1 com o modelo *Dino*), esta apresentou uma vantagem de 82,1% no tempo de construção. No melhor cenário (*Ropes++* e *BIH* MaxPrims = 10 com o modelo *Hornbug*), a *BIH* apresentou uma vantagem de 95,1% na construção.

Na avaliação de desempenho da travessia, a *BIH* perdeu para a *KD-Tree Ropes++*. Porém, na construção a primeira apresentou grande vantagem sobre a última estrutura. Como a performance final de renderização em cenas dinâmicas

leva em consideração estes dois parâmetros, se faz necessária uma análise de performance destas duas estruturas tomando como base o *time to image*.

No cenário em que a *BIH* apresenta menor vantagem (*Ropes++* e *BIH* MaxPrims = 1 com o modelo *Hornbug*), ela apresenta um ganho de 16,8%, onde a *BIH* roda a 59,9 *fps* e a *KD-Tree* roda a 51,3 *fps*. No cenário em que a *BIH* apresenta a maior vantagem em relação à *KD-Tree* (*Ropes++* e *BIH* MaxPrims = 10 com o modelo *Dino*), o ganho é de 608,5%, onde a *BIH* roda a 10,2 *fps*, enquanto a *KD-Tree* roda a apenas 1,4 *fps*.

5. Conclusão

Neste trabalho de graduação foi apresentado um estudo aprofundado do funcionamento da *Bounding Interval Hierarchy – BIH* como estrutura de dados para a aceleração de aplicações que usam *ray tracing*. As análises comparativas realizadas perante as outras estruturas como a *Bounding Volume Hierarchy – BVH* e a *k-Dimensional-Tree KD-Tree* ressaltam as vantagens e desvantagens da *BIH* em relação a outras estruturas.

Através da pesquisa realizada neste trabalho, foi possível concluir e provar que a *BIH* é uma forte candidata na renderização em tempo real de cenas 3D completamente dinâmicas. Suas características de construção e travessia, bem como a boa performance na utilização da memória, levam a concluir que é possível realizar esta tarefa sem usar estruturas como a *BVH* e a *KD-Tree* já bem estabelecidas.

Foi possível constatar também que, para obter uma boa performance na travessia da estrutura, não se faz necessária a utilização de heurísticas aprimoradas como a *Surface Area Heuristic – SAH* na etapa de construção da estrutura. A heurística global simplificada utilizada na *BIH* consegue realizar a divisão dos nós de uma forma eficiente, levando a uma alta performance.

No que diz respeito a cenas totalmente dinâmicas, foi possível concluir que a *BIH* exerce seu papel com grande excelência, se comparada a outras estruturas. Com ganhos de performance que chegam a 600% em relação à implementação *Ropes++* da *KD-Tree*, que é uma otimização do algoritmo original proposto por [4], a *BIH* se apresenta com uma ótima performance de renderização final.

5.1. Trabalhos Futuros

Embora a *BIH* já se apresente com boa performance, ela é uma estrutura relativamente recente e merece mais pesquisa sobre o seu funcionamento, e

esforço na busca e implementação de otimizações em seus algoritmos de construção e travessia.

Como futuras melhorias, deseja-se implementar o suporte completo à construção da *BIH* para cenas que contenham vários tipos de primitivas. O suporte a entidades algébricas como esferas, cones, cubos e planos, além de superfícies paramétricas como a *Bézier Surface*, trarão grande qualidade final ao processo de renderização por *ray tracing*.

6. Referências

[1]. WÄCHTER C., KELLER A.: Instant ray tracing: The bounding interval hierarchy. In Rendering Techniques 2006 (Proc.17th Eurographics Symposium on Rendering), pp. 139–149. 2006.

[2]. GLASSNER A.S. An Introduction to Ray Tracing. 1989. Academic Press. San Diego, EstadosUnidos.

[3]. SHEVTSOV M., SOUPIKOV A., KAPUSTIN A.: Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes. In Computer Graphics Forum (Proc. Eurographics 2007), pp. 395–404.2007.

[4]. POPOV S., GUNTHER J., SEIDEL H.P. AND SLUSALLEK P. 2007. Stackless kd-tree traversal for high performance GPU ray tracing. Em Eurographics'07, 415–424

[5]. GOLDSMITH J., SALMON J.: Automatic Creation of Object Hierarchies for Ray Tracing. IEEE Computer Graphics and Applications 7, 5 (May 1987), 14–20.

[6]. WALD I., BOULOS S., SHIRLEY P.: Ray Tracing De-formable Scenes using Dynamic Bounding Volume Hierarchies. ACM Transactions on Graphics (2006).

[7]. NVIDIA Home. http://www.nvidia.com. Acessado em 31 de Agosto de 2010.

[8]. FARIAS T., ALMEIDA M., TEIXEIRA J.: A High Performance Massively Parallel Approach for Real Time Deformable Body Physics Simulation. SBAC-PAD (2008).

[9]. ALMEIDA M.: Simulação de Corpos Deformáveis Baseada em Pontos em Tempo Real Através de Programação de Propósito Geral em Dispositivo Gráfico. MSc thesis, Universidade Federal de Pernambuco, 2010.

[10]. SANTOS, A.: RT2 – Real Time Ray Tracer. Universidade Federal de Pernambuco, 2009.

[11]. NVIDIA CUDA Programming Guide 3.2. Disponível em: http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf. Acessado em 10 de dezembro de 2010.

[12]. WALD I.: Realtime Ray Tracing and Interactive Global Illumination. PhD thesis, Saarland University, 2004.

[13]. WALD I., HAVRAN V.: On building fast kD-trees for Ray Tracing, and on doing that in O(N log N). Technical Report, SCI Institute, University of Utah, No UUSCI-2006-009 (2006).

[14]. DirectX SDK. Disponível em http://msdn.microsoft.com/en-us/directx/default.aspx. Acessado em 13 de Dezembro de 2010.

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

[16]. FUCHS H., KEDEM Z. M., NAYLOR, B. F.: On Visible Surface Generation by A Priori Tree Structures. ACM Computer Graphics, pp 124–133. July 1980.

[17]. OpenGL API. Disponível em http://www.opengl.org/. Acessado em 13 de Dezembro de 2010.

[18]. WHITTED T.: An improved illumination model for shaded display, Communications of ACM, v. 23, no. 6, ACM, New York, USA, 1980, pp. 343-349.

[19]. BOLDRINI, J. et al. Álgebra Linear. São Paulo: Harbra, 1986.

[20]. Farias, Thiago; Teixeira, João Marcelo; Leite, Pedro; Almeida, Gabriel; Almeida, Mozart; Teichrieb, Veronica; Kelner, Judith. High Performance

Computing: CUDA as a Supporting Technology for Next Generation Augmented Reality Applications. RITA, vol. XVI, no 1, 2009.

[21]. Santos, Artur; Teixeira, João Marcelo; Farias, Thiago; Teichrieb, Veronica; Kelner, Judith. kD-Tree Traversal Implementations for Ray Tracing on Massive Multiprocessors: a Comparative Study. SBAC-PAD 2009.

[22]. Teixeira, João Marcelo; Albuquerque, Eduardo; Santos, Artur; Teichrieb, Veronica; Kelner, Judith. Improving ray tracing anti-aliasing performance through image gradient analysis. WSCAD-SSC 2010.

[23]. The Stanford 3D Scanning Repository. Disponível em http://graphics.stanford.edu/data/3Dscanrep. Acessado em 14 de Dezembro de 2010.

[24]. Foley, James; van Dam, Andries; Feiner, Steven K.; Hughes, John F. (1990). Computer Graphics: Principles and Practice. Reading, MA, USA: Addison-Wesley. p. 1174. ISBN 0-201-12110-7.