UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

2010.2

Uma Bounding Volume Hierarchy Para Um Ray Tracer de Tempo Real

TRABALHO DE GRADUAÇÃO

Aluno	Jorge Eduardo Falcão Lindoso	jefl@cin.ufpe.br
Orientadora	Veronica Teichrieb	vt@cin.ufpe.br

Recife, 15 de Dezembro de 2010

Resumo

Com o avanço da tecnologia dos dispositivos gráficos, a qualidade visual de aplicações de computador interativas melhora. Embora a ocorrência de melhorias produzir uma cena 3D que segue fielmente uma cena real ainda é um desafio. A rasterização, técnica amplamente utilizada atualmente e base dos jogos 3D, não possui ampla precisão em muitos casos que ocorrem no mundo físico, tais como a existência de sombras, reflexões precisas entre objetos e refrações em diferentes meios. Para resolver tal problema existe a técnica do *Ray Tracing*. Embora sendo uma técnica altamente custosa, existem estruturas de aceleração desta técnica que fornecem um ganho significativo de performance. Entre essas estruturas se destacam a *KD-Tree*, BIH e BVH. Neste trabalho ocorre uma maior contextualização nessas estruturas de aceleração dando mais atenção a estrutura da BVH, tema deste trabalho. Durante a realização de testes de performance e consumo de memória, constatou-se que a BVH com os modelos testados obteve a travessia mais dispendiosa computacionalmente do que a *KD-Tree*. No caso da memória, a BVH mostrou, na maioria dos casos testados, exigir mais espaço que a *KD-Tree* padrão e, em todos os casos testados, menos que a *KD-Tree Ropes*, em geral.

Agradecimentos

Agradeço a Deus, aos meus pais, que sempre me ajudaram, me deram força em todos os momentos, motivo de eu ter chegado até onde estou e influência fortíssima. Agradeço também a professora Veronica, minha orientadora deste trabalho, por ter me ajudado nas revisões deste documento e por ter me introduzido nesta área da computação e ter me apresentado esta proposta como idéia. A Artur, que foi de grande ajuda na parte técnica do trabalho, principalmente na parte de programação em GPU e coleta de dados. Ao povo da empresa Jynx Playware, por sempre terem compreendido a minha ausência em alguns dias no expediente devido a esse trabalho e a outras disciplinas. Agradeço também ao professor Sílvio Melo, por se disponibilizar a avaliar este trabalho e ter me apresentado outras propostas durante a fase de decisão do tema. A Geber Ramalho, por também ter me apresentado outras propostas durante a fase de decisão do tema. Agradeço também a Ícaro Malta, aluno de graduação do cin, pela tentativa de colocar uma idéia para meu trabalho de graduação. A Ana Idalina, minha irmã, por criticar problemas de formatação durante a construção deste documento.

Índice

1. Introdução

Com a crescente evolução dos microprocessadores e a demanda por renderização de cenas fotorealísticas, técnicas de iluminação global passaram a ser uma área de interesse para os pesquisadores de diversas áreas, incluindo computação gráfica. Técnicas de iluminação global fornecem como benefício uma solução natural para simular elementos visuais tais como sombras, reflexões de alta precisão, refrações em diferentes meios físicos e, dependendo do grau de sofisticação da técnica, fornecem fenômenos de reflexão difusa, iluminação indireta e cáusticas. O *ray tracing* [7], um exemplo de técnica de iluminação global, passou a ser amplamente utilizado e estudado. Devido ao seu alto custo computacional para renderização, a renderização utilizando o algoritmo convencional da técnica torna-se proibitivo. Para solução de tal problema, existem as chamadas estruturas de aceleração que são utilizadas para otimizar o algoritmo convencional e produzem o mesmo resultado com um ganho de performance significativo. Neste trabalho de graduação são destacadas três estruturas de aceleração, a saber *KD-Tree* (*K Dimensions Tree*) [12], BIH [13] e BVH (*Bounding Volume Hierarchy*) [9]. A última é tema deste trabalho e será analizada e comparada com a estrutura da *KD-Tree* em termos de performance e memória consumida.

1.1. Objetivo

Este trabalho tem como finalidade principal abordar a estrutura de aceleração da BVH a qual, como toda estrutura de aceleração, possui as etapas de construção da mesma e aplicação desta durante a renderização do *ray tracing* em tempo real. Serão abordadas as duas etapas, assim como análises comparativas da execução da implementação da estrutura com a *KD-Tree* com destaque maior na etapa de travessia.

Além deste propósito principal, este trabalho visa abordar conceitos no estado da arte, o que está sendo feito na área de computação gráfica,

conceitos de rasterização e *ray tracing*, assim como conceitos de programação em GPU e tecnologia CUDA (*Compute Unified Device Architecture*) [11], produzida pela NVIDIA, empresa de produção de placas gráficas.

1.2. Estrutura do Documento

Este documento é organizado da seguinte forma. No capítulo seguinte (capítulo 2) é discutido o contexto em que se insere este trabalho, assim como o estado da arte onde se encaixam os conceitos de rasterização, *ray tracing*, estruturas de aceleração principais, advento das placas gráficas, tecnologias existentes de programação em GPU e é abordado o projeto RT^2, aplicação interativa de *ray tracing*. No capítulo 3, é apresentada em detalhes a estrutura de aceleração da BVH, assim como sua construção, aplicação num *ray tracing* interativo e comparação com a estrutura da *KD-Tree*. No capítulo 4, são exibidos os resultados da execução deste trabalho e inclusive dados comparativos em relação a aplicação da estrutura da *KD-Tree*. No capítulo 5 é concluído este trabalho e realizado o levantamento de trabalhos futuros.

2. Contexto

Construir uma aplicação computacional que executa em tempo real e que apresenta resultados visuais realísticos é um desafio. Este desafio vem sendo estudado com mais intensidade na medida em que ocorrem os avanços no desenvolvimento dos dispositivos gráficos. Durante muito tempo, desde 1968, com a criação do conceito conhecido como *ray casting* [8] a partir do qual foi introduzido o modelo de iluminação chamado de ray tracing, as áreas que necessitavam de aplicações gráficas em tempo real ficaram limitadas ao uso da rasterização [1], principalmente por causa de limitações dos dispositivos gráficos. Atualmente, já é possível realizar simulações em tempo real utilizando técnicas de iluminação global com o auxílio de estruturas de aceleração, que são abordadas neste trabalho com foco na estrutura chamada BVH [9]. Nas próximas seções os conceitos de rasterização e *ray tracing* serão brevemente apresentados, bem como as características das estruturas de aceleração adotadas em *ray tracing*. Com relação aos dispositivos gráficos, este trabalho utiliza uma arquitetura chamada CUDA [11], oferecida por placas gráficas da NVIDIA; CUDA também será descrita na sequência. Finalmente, o *Real Time Ray Tracer* ou simplesmente RT^2 [5], *ray tracer* utilizado neste trabalho para analisar comparativamente a BVH e uma outra estrutura de aceleração amplamente utilizada em *ray tracing* conhecida como *KD-Tree* [12], é introduzido.

2.1. Rasterização

A Rasterização é um modelo de renderização que consiste no processo de conversão de dados vetoriais em *pixels* [1]. Este modelo parte da idéia de elementos gráficos consistirem em primitivas de pontos, retas e polígonos. Atualmente, é o modelo mais utilizado em aplicações gráficas em tempo real, e tais como ferramentas de modelagem (3ds Max e Maya, por exemplo), simulações físicas e visuais e jogos. As placas gráficas, desde a sua primeira geração (1994-1998), possuem *hardware* otimizado para realizar

a rasterização. O processo de rasterização consiste em discretizar primitivas projetadas no plano de visão da câmera virtual, transformando-as em *pixels*. A figura 2.1 mostra um exemplo desse processo, aplicado a arestas.

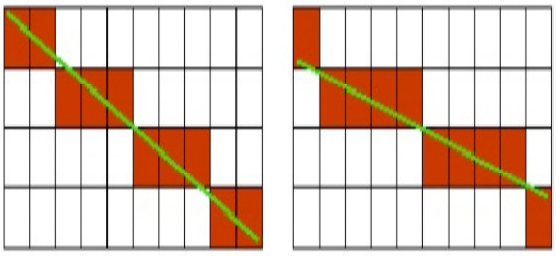

Fig. 2.1 Processo de rasterização sendo aplicado em segmentos de reta.

Durante o processo de rasterização, são realizados os cálculos de iluminação em cada ponto do espaço 3D correspondente a cada *pixel* obtido. A rasterização se destaca pela performance, viabilizando aplicações em tempo real. A área de jogos 3D é um exemplo de utilização intensa da rasterização, utilizando *engines* 3D desenvolver a aplicação em tempo real. Entre os engines de jogos pode-se citar o Unity3d [3], que possui como ferramenta gráfica o sistema de *shaders* pré-definidos, permite programar *shaders* personalizados com linguagens de *shader* como *C for Graphics* (Cg) [17] da NVIDIA, *High Level Shader Language* (HLSL) [18] da Microsoft e *OpenGL Shading Language* (GLSL) [4], linguagem que utiliza recursos da tecnologia *Open Graphics Library* (OpenGL) [19] permitindo criar aplicações para dispositivos móveis. Um *shader* [4] é um programa que define propriedades de vértices 3D e *pixels*. Subdivide-se nos passos de transformação de primitivas no espaço 3D, de interpolação e rasterização e de processamento das cores dos *pixels*. Ele utiliza como base o passo a passo clássico de rederização, onde se destaca a rasterização em dispositivo gráfico.

Embora muito utilizada atualmente, a rasterização possui limitações. O problema principal está na restrição quanto ao tipo de primitivas que são utilizadas nas aplicações, pois a rasterização depende de primitivas triangularizadas para ser aplicada, não podendo trabalhar com outros tipos

tais como quádricas, curvas e superfícies de *bézier* ou *Non Uniform Rational Basis Spline* (NURBS). Além disso, os cálculos de iluminação ocorrem de forma local, dependendo somente do ponto do objeto 3D sendo trabalhado e, portanto, não existindo características como sombras, reflexões e transparências. Com isto, o que se usa muito atualmente para simulações destas características são projeções de objetos sobre superfícies para gerar sombras, espelhamento sobre superfícies para gerar reflexões e canais *alpha* com processamento de imagens para simular transparências e refrações. Outra limitação é que o cálculo da cor de cada *pixel* se baseia em uma aproximação do mapeamento do *pixel* sendo trabalhado para o ponto correspondente no espaço 3D e, finalmente, calculando a cor deste ponto e atribuindo este valor para o *pixel*.

Atualmente existe um modelo de renderização que está em evolução, a saber *ray tracing*, que realiza os cálculos de iluminação apresentando algumas das propriedades do modelo de iluminação global e não depende de primitivas triangularizadas para realizar a renderização. Pode-se dizer que o *ray tracing* é o ponto de partida para modelos de iluminação global devido a existência de tais propriedades. Esta técnica é abordada na seção seguinte.

2.2. Ray Tracing

Modelos de iluminação global são capazes de produzir a renderização de cenas com uma qualidade visual bem próxima da realidade, pois a cor de um dado *pixel* de uma cena depende, além do objeto local sendo tratado, depende também de todos os objetos que compõem a mesma para a simulação das características do modelo. O *ray tracing* [5] é uma técnica que se baseia em conceitos básicos da física ótica para renderização da cena virtual com um aspecto mais real. Mais especificamente, o *ray tracing* se baseia no processo de emissão de raios luminosos das fontes de luz na cena. Tais raios podem ser refletidos entre objetos e transmitidos através de objetos, produzindo o efeito de refração explicado pela física ótica. Embora esta técnica utilize conceitos básicos da física ótica, fugindo um pouco dos fenômenos naturais que ocorrem no mundo real tais como

sombras “moles” (que consistem em umbras e penumbras), iluminação indireta de um objeto não luminoso sobre outros, e os fenômenos de cáusticas (objeto não especular aparentando especularidade), que justifica o fato do *ray tracing* não se encaixar na categoria do modelo de iluminação global, oferece uma imagem final com muito mais riqueza visual se comparada com cenas rasterizadas. Na figura 2.2 é mostrado um comparativo entre uma cena rasterizada com adaptações e uma cena produzida com *ray tracing*.

Fig. 2.2 No lado esquerdo, uma cena rasterizada e no lado direito, uma cena renderizada usando ray tracing.

Uma propriedade muito importante que se pode destacar no ray tracing é, ao contrário da rasterização, a independência de triangularização de objetos para a renderização. Com isso, é possível representar mais entidades geométricas e espaciais, de forma mais precisa. Um exemplo destas primitivas não triangularizadas são as superfícies de quádricas [6], um conjunto de primitivas constituído por esferas, elipsóides, cones, cilindros, parabolóides e hiperbolóides. A figura 2.3 mostra alguns exemplos de superfícies quádricas.

O *ray tracing* demanda maior poder computacional para gerar imagens de qualidade visual superior devido à cor de cada *pixel* da cena depender de todos os objetos da mesma. Essa dependência deve-se pelo fato do *ray tracing* simular parte das propriedades dos modelos de iluminação global, a saber reflexões, refrações e sombras duras (*hard shadows*). O *ray tracing* permite a renderização de sombras, reflexões, transparências e refrações como solução da renderização, já na rasterização

se faz adaptações para simular tais propriedades, tornando-as muitas vezes imprecisas.

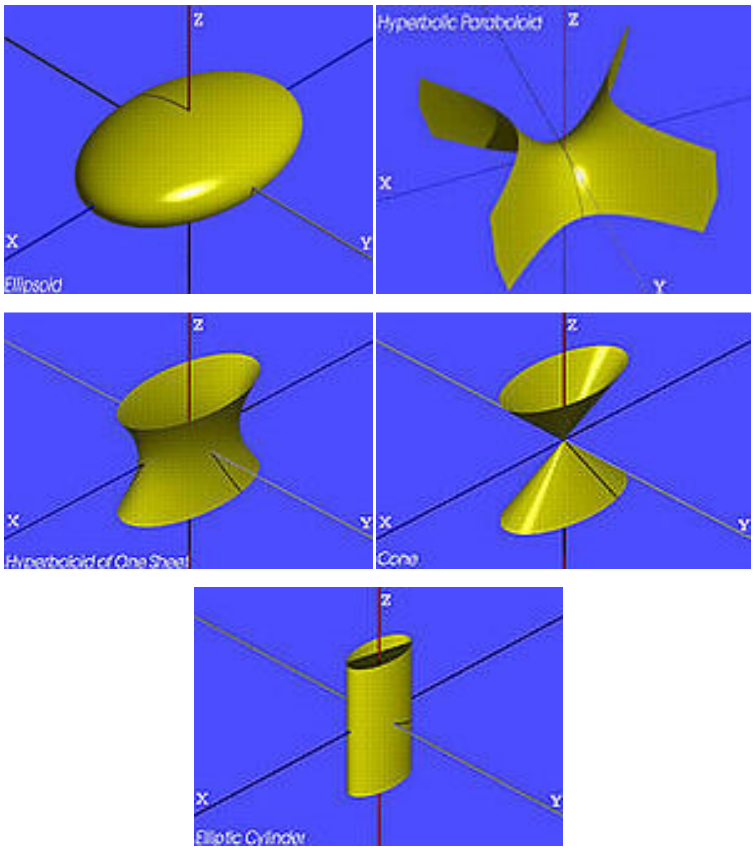

Fig. 2.3 Exemplos de superfícies quádricas, de cima e da esquerda para baixo e a direita, elipsóide, parabolóide hiperbólico, hiperbolóide de uma folha, cone, cilindro elíptico.

No mundo real, de acordo com a física ótica, os raios luminosos partem das fontes de luz e se transportam pelo espaço físico, atingindo os nossos olhos. No *ray tracing* ocorre o processo inverso onde os raios luminosos partem do observador, e atingem os elementos da cena e as fontes de luz. Em virtude de grande parte dos raios de luz do mundo real originados pelas fontes luminosas não atingirem nossos olhos, o *ray tracing* foca no que o observador está vendo e daí o processo inverso. No *ray tracing* existe o observador conhecido como câmera virtual ou sintética que possui uma janela ou plano de visão dividido em pequenos quadrados, e o número de quadrados no plano define a resolução da cena final. Para cada quadrado é criado um raio que parte do observador em direção àquele quadrado; esse processo também é conhecido como *ray casting*. A figura 2.4 ilustra o processo descrito.

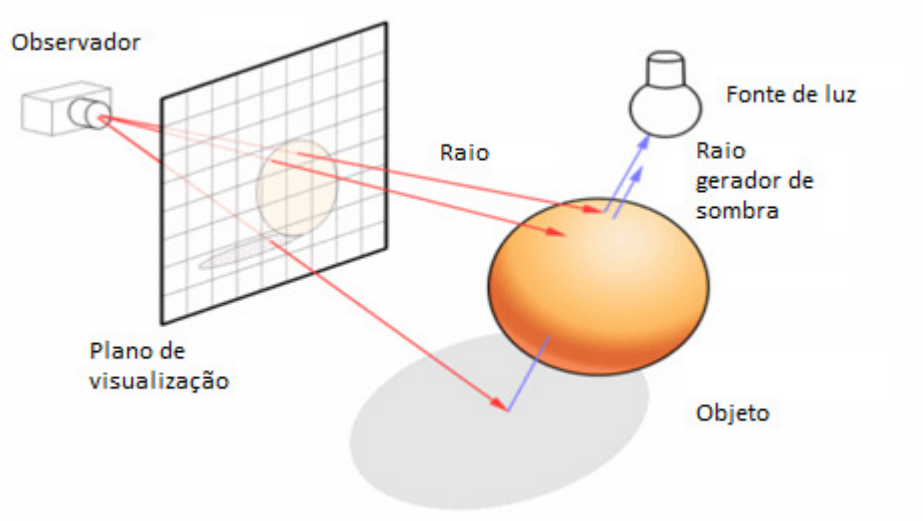

Fig. 2.4 O *ray casting* sendo aplicado na cena de uma esfera com uma fonte de luz, com ocorrência de sombra.

O *ray casting* também é conhecido como o *ray tracing* básico, consistindo em determinar a cor do quadrado através do objeto interceptado pelo raio, criado em função da posição do observador. Quando ocorre uma intersecção, é realizado o cálculo de cor naquele quadrado que, por sua vez, é convertido em um *pixel* da cena final. O ray tracing completo envolve não somente os raios gerados pelo *ray casting*, também conhecidos como raios primários, mas também os raios refletidos e transmitidos na cena, também chamados de raios secundários. Os raios secundários são originados a partir da intersecção de um raio com um objeto da cena. Uma ilustração da ocorrência de raios secundários pode ser vista na figura 2.5.

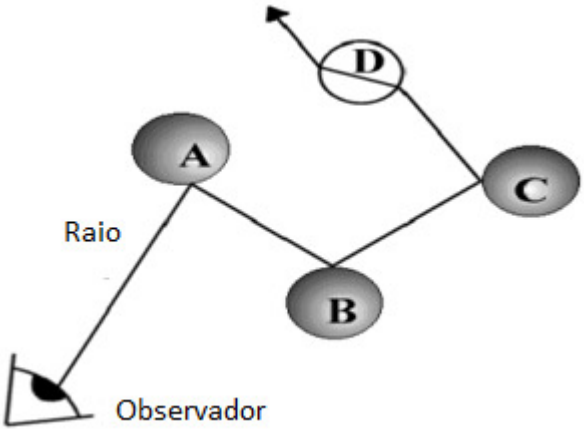

Fig. 2.5 Raio primário partindo do observador, raios refletidos partindo de A, B e C, raio transmitido em D.

Os raios refletidos são gerados a partir da lei da reflexão [7], enquanto os transmitidos obedecem a lei da refração de Snell [7]. Estas regras são demonstradas na figura 2.6. Pode-se observar que os raios refletidos são calculados vetorialmente, em função do vetor normal do ponto da superfície e do raio incidente. Enquanto os raios transmitidos são calculados em função dos índices de refração dos meios e do raio incidente.

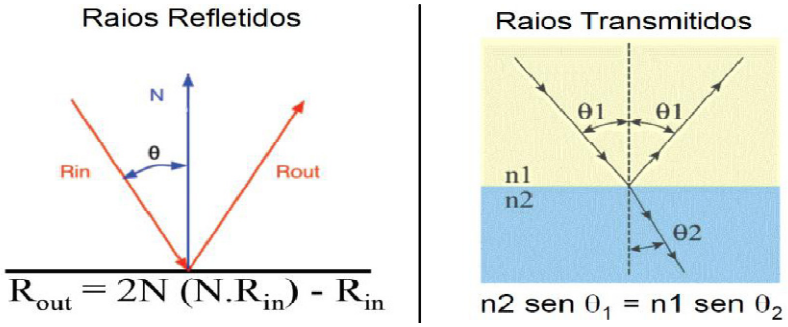

Fig. 2.6 Cálculo dos raios refletidos e transmitidos. A esquerda, para o cálculo do raio refletido é necessário conhecer o vetor normal N e o raio incidente Rin. A direita, o raio tranmitido é determinado pelo ângulo do raio incidente e pelos índices de refração n1 e n2 dos meios.

Existe um limite definido numa aplicação de *ray tracing* que determina a quantidade de raios secundários criados a partir de um raio primário; esse limite também é conhecido como profundidade. O grande custo computacional do *ray tracing* está na quantidade de testes de intersecção entre raios e primitivas da cena. Esse custo pode ser amenizado drasticamente através da aplicação de estruturas de aceleração. Na próxima seção elas são brevemente apresentadas.

2.3. Estruturas de Aceleração

O grande custo computacional do *ray tracing* é devido à necessidade, para cada raio lançado na cena, da verificação, em todas as primitivas contidas na cena, da intersecção mais próxima. Esse problema de varredura em todas as primitivas pode ser resolvido partindo da idéia de amenizar o número de verificações de intersecção entre raio e primitiva. O algoritmo do *ray tracing* utiliza 95% do tempo de processamento para realizar a verificação de intersecções entre raios e elementos da cena, tornando esta redução do número de verificações indispensável para uma aplicação em tempo real. Para atacar tal problema, são utilizadas tradicionalmente as

estruturas de aceleração, que são constituídas, em sua maioria, por estruturas de árvores de busca. Uma árvore de busca que serviu de origem às estruturas de aceleração é a chamada *Binary Space Partitioning Tree* ou *BSP Tree* [12], que consiste em uma estrutura de dados de árvore onde cada nó consiste em uma cena 3D e os nós filhos em subdivisões binárias do espaço 3D do nó pai. Essa subdivisão que cada nó da árvore BSP possui é definida por um plano de divisão no espaço 3D. Estas estruturas consideram objetos com maior probabilidade de intersecção com o raio construído e descartam os demais, melhorando drasticamente a performance. Para determinar tal probabilidade são construídos objetos mais simples que envolvem subgrupos de objetos contidos numa cena 3D, podendo ser constituídos por caixas ou *bounding boxes* que subdividem a cena em sub-cenas até que se construa uma hierarquia desses objetos. Essas caixas podem ser constituídas pelos chamados *Axis Aligned Bounding Boxes* ou simplesmente AABBs [12], tratando-se de caixas alinhadas aos eixos como mostra o próprio nome. Essa abordagem faz com que a verificação seja realizada em objetos que envolvem grupos de primitivas, para depois verificar os elementos no interior desses objetos permitindo, assim, um ganho de performance significativo do *ray tracing*.

A estrutura a ser abordada neste trabalho é a chamada *Bounding Volume Hierarchy* ou BVH que consiste em etapas de pré-processamento (construção da árvore) e busca, assim como outras estruturas existentes na literatura. No presente trabalho o autor fará comparações da BVH com a estrutura amplamente utilizada em *ray tracing*, a chamada *KD-Tree*. Serão realizadas comparações tanto da etapa de pré-processamento como de busca, no sentido de identificar as vantagens e desvantagens das estruturas quando aplicadas a diferentes cenas renderizadas com o *ray tracing*. Nesta seção serão descritas as estruturas da *KD-Tree* e da BIH. A estrutura da BVH, que é o foco de investigação deste trabalho, será detalhada no capítulo 3.

2.3.1. KD-Tree

KD-Tree ou *K Dimensions Tree* [12] é uma estrutura específica da árvore BSP, uma vez que, na sua construção, os planos de divisão de uma

cena 3D presentes em cada nó da árvore são restritos aos eixos padrões do espaço 3D. Cada um desses planos possui como vetor normal um vetor correspondente a um vetor canônico do espaço 3D que, por sua vez, corresponde à direção dos eixos ordenados. Na etapa de construção ocorre a criação do nó da cena ou raiz, onde, partindo deste, ocorrem partições da cena através de determinação dos planos de corte alinhados a um dos eixos. Isso consiste no caso base da construção, uma vez que o caso indutivo consiste em aplicar esse processo de divisão dos nós filhos que vão sendo criados a partir da divisão do nó raiz. A figura 2.7 ilustra um processo de partição de um AABB do nó raiz em nós filhos; pode-se observar que o plano vermelho divide o nó raiz, o plano verde divide os nós filhos do nó raiz, e assim por diante.

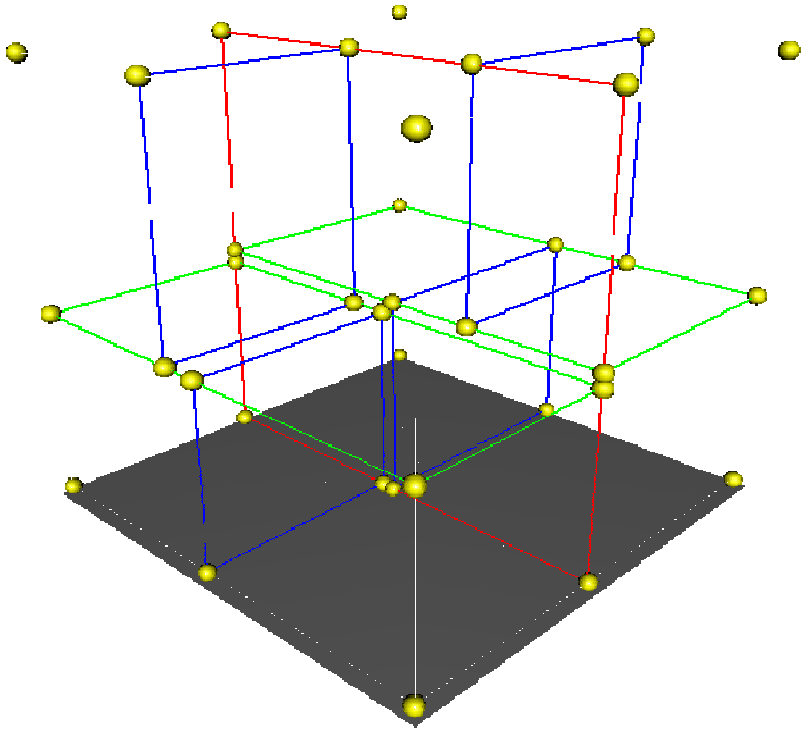

Fig. 2.7 Cena 3D sendo subdividida por planos na etapa de construção da *KD-Tree*.

A etapa de travessia consiste em percorrer a estrutura da *KD-Tree* através da construção de um raio do *ray tracing*, verificar se há intersecção com o AABB da cena para, assim, realizar a busca na estrutura. Como consequencia da etapa de construção da *KD-Tree*, um nó da estrutura possui um AABB cujo volume é igual a soma dos volumes dos AABBs dos nós filhos,

não havendo uma área vazia no interior de um nó. Numa travessia da *KD-Tree* ocorrem os seguintes casos de intersecção raio-caixa:

- O raio intercepta somente um nó filho contido na caixa do nó corrente sendo analisado.
- O raio intercepta os dois nós filhos contidos na caixa do nó corrente sendo analisado.

A figura 2.8 ilustra esses dois casos.

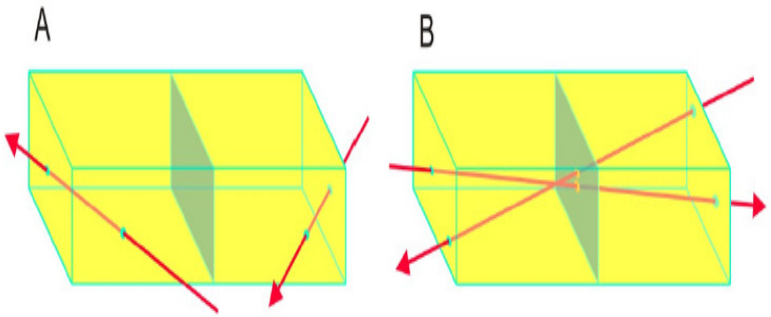

Fig. 2.8 Raio interceptando uma caixa (A) e ambas as caixas (B).

No caso em que ocorre intersecção em dois nós, cabe ao algoritmo decidir qual nó tratar primeiro e ao mesmo tempo considerar que houve intersecção em outro nó não tratado. Para resolver tal problema, existe uma abordagem utilizada no algoritmo de travessia padrão da *KD-Tree* que é a utilização de uma estrutura de pilha para armazenar os nós não tratados na travessia em caso de intersecção de dois nós. Quando há falha na busca de uma primitiva em um determinado nó, o último elemento inserido na pilha é removido para dar continuidade à busca. Se não houver mais elementos na pilha a serem tratados e ocorrer falha de busca na travessia, esta é encerrada. Na literatura, há várias adaptações [12] do algoritmo de travessia que fornecem vantagens e/ou desvantagens a este. A figura 2.9 ilustra estas adaptações incluindo a travessia padrão (*Stack-Based* ou Baseada em Pilha).

Em [12] há maiores informações a respeito dessas adaptações, as quais não fazem parte do escopo deste trabalho.

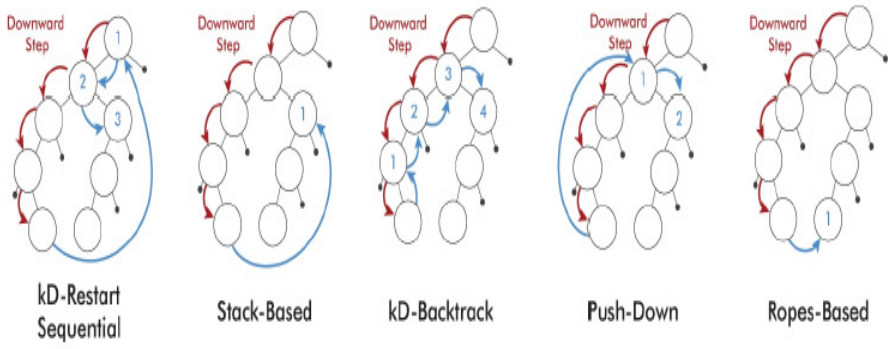

Fig. 2.9 Adaptações ao algoritmo de travessia padrão da *KD-Tree*; as setas vermelhas representam a busca de cima para baixo e as setas azuis representam a busca em nós não visitados.

2.3.2. BIH

A *Bounding Interval Hierarchy* ou *BIH* [13], assim como a *KD-Tree*, é uma estrutura de aceleração que consiste numa árvore binária de nós correspondentes às subcenas da cena 3D. A diferença ocorre na etapa de construção na qual, dado o AABB do nó sendo tratado nesta etapa, a subdivisão da subcena é determinada por dois planos de corte perpendiculares aos eixos x, y ou z. Esta subdivisão de dois planos também é conhecida como *clipping*, onde os planos separam os objetos da cena podendo haver ou não regiões vazias do nó pai que não fazem parte dos nós filhos, diferentemente da *KD-Tree*, onde, por sua vez, os nós filhos sempre ocupam todo o volume do nó pai.

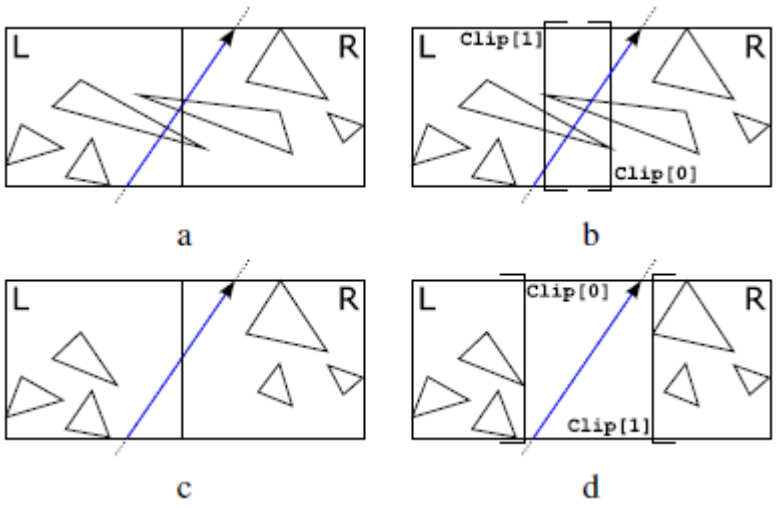

Fig. 2.10 Em (a) a travessia ocorre nos dois nós utilizando como estrutura a *KD-Tree*, assim como também em (b) utilizando a BIH como estrutura. Em (c), utilizando *KD-Tree*, ocorre novamente travessia em dois nós, enquanto em (d), com a estrutura da BIH, nenhum nó é interceptado.

Na figura 2.10 os casos da ocorrência de clippings de um nó em conjunto com um raio de travessia são ilustrados. Esta também ilustra um comparativo de travessias na KD-Tree e na BIH, em dois casos de uma cena.

2.4. O Advento da GPU

Através de *hardware* gráfico, tornou-se possível criar aplicações que requerem cáculos de ponto flutuante constantes e de grande custo computacional e que podem ser pararelizados em *Graphics Processing Unit* (GPU), podendo subdividir um processo em unidades de processo chamadas *threads*. Os dispositivos gráficos são otimizados para realizar várias operações ao mesmo tempo. Para que tais operações ocorram de forma correta, é preciso que estas sejam paralelizáveis. O passo a passo de operações de uma GPU consiste no chamado *pipeline gráfico* [10]. A figura 2.11 ilustra o mesmo. Pode-se observar que inicialmente ocorre no processo a tranferência de primitivas 3D triangularizadas do estágio de aplicação para o estágio de geometria. No estágio de geometria, ocorre a projeção das primitivas para o plano 3D de visualização da cena e, por fim, ocorre a rasterização das primitivas 2D e verificações de visibilidade.

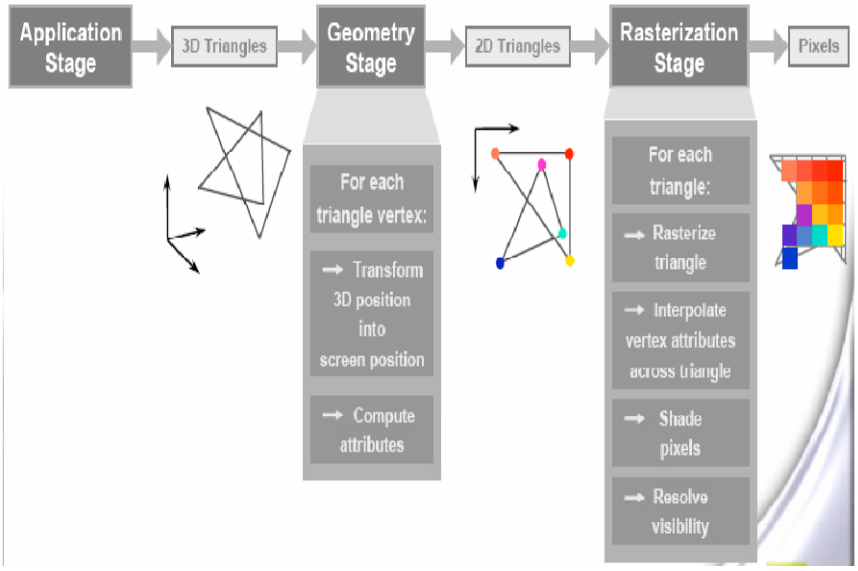

Fig. 2.11 Passo a passo de um pipeline gráfico, transformações geométricas e rastirazação são feitos na GPU.

Os dispositivos gráficos, também chamados GPUs, desde a primeira geração [2], são otimizados para realizar cálculos de interpolações de pontos de uma cena 3D, mapeamento de texturas e rasterização dos pontos

em *pixels* de forma paralela. Nos anos 1999 e 2000, surgiu a segunda geração das GPUs com recursos de transformações de vértices, cálculo de iluminação no próprio *hardware* e mapeamento de texturas cúbicas em modelos 3D. Em 2001, surgiu a terceira geração, na qual tornou-se possível, com poucas instruções, programar *shaders*, além de surgirem mais texturas 3D e mapas de sombras. Na quarta geração, entre 2002 e 2003, surgiram mais instruções e linguagens de GPU em mais alto nível para programação e, na geração seguinte, nos anos 2004 e 2005, surgiram as multi-GPUs.

Com o avanço da tecnologia de GPUs tornou-se possível programar de forma mais flexível aplicações que funcionam no *hardware* gráfico. Uma das tecnologias que se destaca e será discutida na seção seguinte é a chamada CUDA.

2.4.1. CUDA

Em Novembro de 2006 [11], a NVIDIA introduziu a tecnologia *Compute Unified Device Architecture* (CUDA), uma arquitetura que permite programar aplicações de propósito geral com paralelismo nas execuções nos dispositivos gráficos. Uma propriedade que se destaca em CUDA é a sua integração com linguagens de programação de propósito geral. As linguagens que podem ser integradas com CUDA e que são suportadas são C/C++, FORTRAN, OpenCL e DirectCompute. Uma aplicação CUDA consiste em uma aplicação de GPU escrita em uma linguagem de propósito geral com funcionalidades da arquitetura CUDA. Atualmente, as GPUs compatíveis com CUDA são as dos modelos fabricados pela NVIDIA desde a GeForce 8, que suporta a versão 1.0 desta tecnologia, a versão mais antiga. A figura 2.12 ilustra as camadas de uma aplicação CUDA. A camada mais acima corresponde à própria aplicação de GPU; a camada intermediária consiste em um compilador da linguagem de propósito geral com extensões de CUDA; e, por fim a camada do hardware gráfico que suporta CUDA.

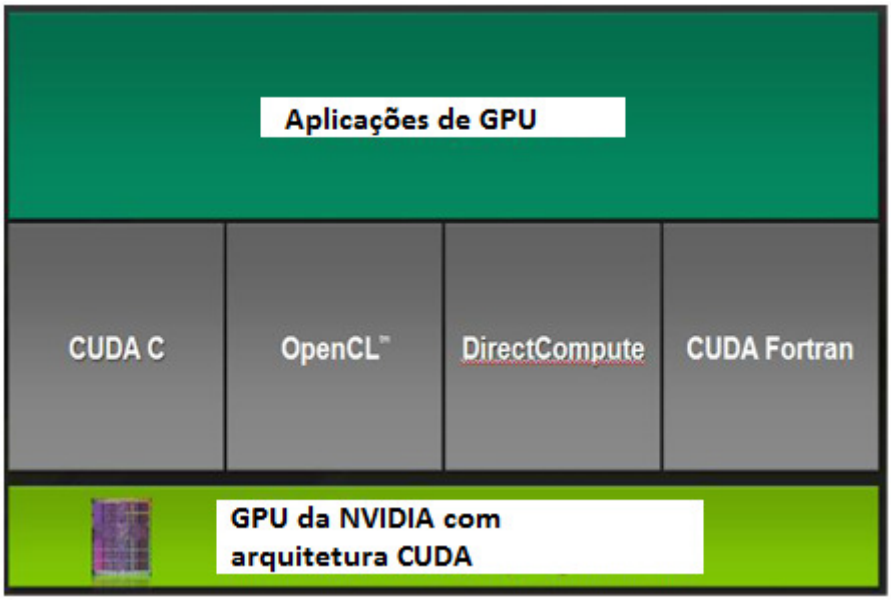

Fig. 2.12 A camada acima consiste da aplicação, abaixo é do compilador da linguagem utilizada e, mais abaixo, a arquitetura CUDA com funcionalidades que auxiliam a programação na linguagem utilizada.

A tecnologia CUDA possui três principais componentes de software:

- CUDA SDK – Fornece exemplos de código e bibliotecas necessárias para compilação destes códigos para CUDA, além de fornecer documentações sobre programação em CUDA.
- CUDA toolkit – Contém o compilador e as bibliotecas adicionais. A versão atual é a 3.2.
- CUDA driver – Componente que vem instalado junto com o driver da própria placa gráfica, necessário para o sistema operacional. Tem como funcionalidade realizar toda a comunicação entre o sistema operacional e a GPU.

Neste trabalho é utilizado o CUDA integrado com a linguagem C/C++, utilizada na implementação. CUDA usufrui das propriedades do *hardware* gráfico que são o número de núcleos de processamento e a capacidade de cada núcleo. Dentro deste contexto estão os conceitos de *grids*, estruturas que definem o número de unidades de processamento na GPU, de blocos, unidades contidas nos *grids* e de *threads*, entidades do programa que se localizam em um bloco e que fazem operações de uma fração do programa na GPU. O número total de *threads* ou processos que ocorrem em um programa, a cada iteração no caso de aplicações em tempo real ou a cada

execução, é o produto do número de blocos do *grid* e do tamanho do bloco (número de *threads* contidas no bloco). A performance de um programa CUDA depende da GPU utilizada e da forma que foi implementada a aplicação, tomando vantagem dos recursos da GPU. A figura 2.13 mostra um exemplo de uma GPU de 2 núcleos e uma de 4 núcleos executando o mesmo programa. A seta vertical denota o tempo de execução.

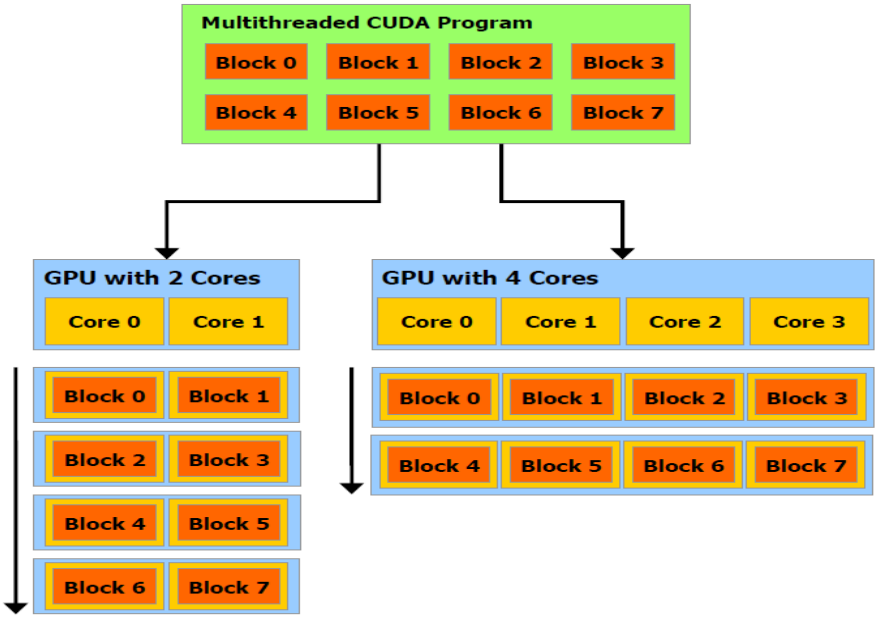

Fig 2.13 Programa de tamanho de *grid* 8 sendo executado em GPU's de 2 núcleos (esquerda) e de 4 núcleos (direita). Observe que a GPU de 4 núcleos consegue terminar antes a execução.

Um conceito importante a ser observado ao desenvolver aplicações em CUDA é o das hierarquias de memória, onde o tempo de acesso varia dependendo do tipo de memória utilizada. Nesta hierarquia existem as memórias de leitura que podem ser acessadas por todas as *threads* do programa se destacando a memória global, memória de textura e memória constante. Memórias de texturas e constantes possuem tempo de acesso de dados menor que a memória global, onde cada uma possui suas restrições quanto a tipos de dados que devem ser utilizados e seu funcionamento depende também do modelo da placa de vídeo. Há também as chamadas memórias compartilhadas ou memórias de bloco, as quais podem ser acessadas por *threads* que estão num mesmo bloco. Finalmente, existe a memória local que é exclusiva para uma *thread*. Cada *thread* possui uma memória local. Em placas de vídeo mais modernas, as chamadas GeForce da série GTX, tornou-se desnecessário o uso de memórias que diminuiam o

tempo de acesso aos dados. Numa aplicação em CUDA, existem funções chamadas *kernels* as quais são executadas na placa de vídeo, dividindo-se nos chamados *kernel global* e *kernel device*. Um *kernel global* é uma função que dá início ou continuidade à execução do programa na GPU, podendo ser chamado dentro de uma função em CPU ou em outra função de GPU. Um *kernel device* é aquele que só pode ser utilizado dentro de uma função na GPU. Em CUDA C, há palavras chaves que podem ser utilizadas em códigos C para denotar o tipo de função que está sendo desenvolvida; a palavra para *kernel global* é "__global__", para *kernel device* é "__device__". Existe também uma palavra chave para denotar funções de CPU com sintaxe de CUDA, a saber "__host__". Neste tipo de aplicação, há dois compiladores que são atuados, o compilador de código C/C++ em CPU e o compilador nvcc que compila arquivos com extensão ".cu" que possuem palavras chave e funções CUDA com código C/C++. Os arquivos ".cu" possuem trechos de código que atuam na CPU para chamada de *kernels* e/ou códigos que atuam na GPU. Para maiores informações sobre CUDA, pode-se consultar [11].

Neste trabalho, foi utilizado o projeto RT^2 [5] que consiste em um *ray tracer* em tempo real que utiliza como estrutura de aceleração a *KD-Tree*. A arquitetura deste projeto foi aproveitada para a integração deste trabalho e tal projeto será discutido na seção seguinte.

2.5. Real Time Ray Tracer – RT^2

Real Time Ray Tracer ou RT^2 [5] [14] [15] é um projeto iniciado no trabalho de graduação de Artur Lira dos Santos, sendo continuado durante o seu curso de mestrado, no Centro de Informática da UFPE. O RT^2 consiste em uma API (*Application Programming Interface*) gráfica para renderização de cenas 3D interativas. Mas, especificamente, esta renderização é feita por *ray tracing*. A API tira proveito do *hardware* do dispositivo gráfico em que é executada, pois possui rotinas de detecção da quantidade de processos que a CPU pode realizar ao mesmo tempo, assim como a placa de vídeo instalada no computador e suas propriedades, tais como número de núcleos e memória de vídeo total. Uma característica forte do RT^2 em relação a outras API's gráficas é a sua independência do hardware gráfico, pois

diferentemente das API's como OpenGL e DirectX, que dependem do hardware para inserir novas funcionalidades, o RT² pode ser atualizado a qualquer momento. É importante observar que o RT² não supera em performance a rasterização em hardware e sim realiza uma troca de performance por maior qualidade gráfica. O RT² já possui rotinas de shading prontas para realizar cálculos de iluminação após a ocorrência de travessia. Em consequência do uso de ray tracing, o RT² possui uma maior compatibilidade com um maior conjunto de primitivas, reflexões nativas de alta precisão, não ocorrem transformações de elementos para coordenadas de câmera e tampouco teste de visibilidade, uma vez que a própria travessia determina a visibilidade dos objetos na cena, e possui *hard shadows* ou sombras dura nativos.

2.5.1. Arquitetura

A arquitetura do RT² consiste em uma camada de software que fornece funcionalidades de maneira transparente e eficiente. Esta camada realiza a comunicação com duas arquiteturas de hardware, de CPU e GPU. A figura abaixo ilustra as camadas da arquitetura do RT².

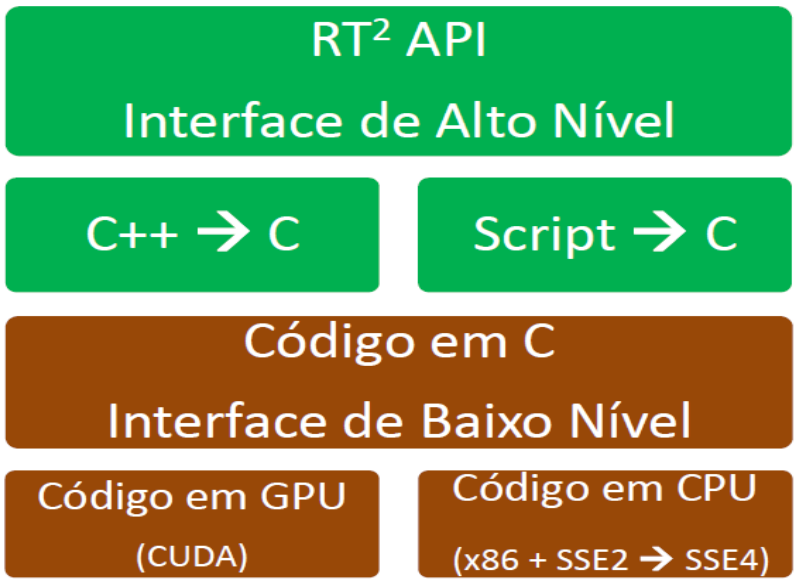

Fig. 2.14 Arquitetura de software do RT².

A versão atual do RT² implementa o módulo em C++ faltando o modulo de Script. Mais abaixo das camadas C++ e Script existe uma camada de controle que corresponde a interface de baixo nível. Esta interface

escalona todas a funções chamadas pelas camadas mais altas e decide qual arquitetura utilizar tais funções. A exemplo da *KD-Tree*, ocorre o repasse de construção da estrutura para a arquitetura de CPU, já a travessia, ocorre na GPU. O módulo para GPU é escrito em C com extensões de CUDA enquanto o módulo para CPU é escrito em C e também em assembly x86 com extensões SSE4 [5].

2.5.2. Núcleo

O RT^2 usufrui da característica do *ray tracing* ser um processo paralelizável, uma vez que os valores de cor dos pixels a serem computados pelo algoritmo são independentes entre si em relação aos demais *pixels*. Seguindo esta idéia, o RT^2 distribui o trabalho do algoritmo para diferentes threads de CUDA para cada pixel ocorrendo computação simultânea. Por questões de otimização, CUDA não possui suporte a chamadas recursivas, assim o RT^2 utiliza a adaptação do algoritmo padrão [5] do *ray tracing* para a sua versão iterativa. O RT^2 também possui um número reduzido de acesso à memória global da GPU com intuito de executar o projeto em placas mais antigas. Para tal redução, foram utilizado os benefícios de CUDA de memórias de acesso mais rápido tais como memória compartilhada e memória de texturas (ver seção 2.4.1).

2.5.3. Raios Secundários

A versão do RT^2 utilizada possui como raios secundários apenas os raios refletidos, uma vez que, para os casos de refrações, ocorre o problema de criar uma nova estrutura de pilha para tratar casos de refração e reflexão em que raio atual se encontra, causando uma penalidade na performance em CUDA [5]. Para tratar os casos de raios secundários, a execução de cada *pixel* é subdividida em etapas de forma que cada etapa corresponda a um novo raio refletido a ser tratado. Desta forma, inicialmente o *kernel* chamado responsabiliza por processar todos os raios primários. No fim deste kernel é realizado um mapeamento dos *pixels* que irão gerar raios secundários.

3. Bounding Volume Hierarchy

Neste capítulo será descrita a estrutura de aceleração, chamada *Bounding Volume Hierarchy* ou simplesmente BVH [9], que foi estudada e implementada no *ray tracer* RT^2. Esta integração teve o objetivo principal de analisar comparativamente a BVH e uma estrutura de aceleração comumente utilizada em *ray tracing*, a *KD-Tree*, quanto ao seu desempenho. Primeiramente serão apresentados os conceitos principais a respeito da estrutura, para em seguida abordar em detalhes as etapas de construção da estrutura assim como de travessia. Logo depois, são realizadas comparações com a estrutura da *KD-Tree*, já presente no RT^2, levando em consideração a performance no tempo de construção da estrutura e durante a navegação numa cena 3D em *ray tracing* onde serão aplicados constantemente os algoritmos de travessia em ambas estruturas.

3.1. Conceitos Principais

A BVH é uma estrutura de dados correspondente a uma árvore binária onde cada elemento desta corresponde a uma subcena 3D. Sua principal diferença, em relação a outras estruturas de aceleração, é a composição interna de cada nó da estrutura. Assim como outras estruturas, são utilizadas bounding boxes como elementos que envolvem uma cena 3D ou subcena. O destaque desta estrutura é a maneira que os AABBs, bounding boxes utilizados neste trabalho, envolvem os nós filhos presentes em um determinado nó da BVH. A estrutura tem como conceito que cada subcena deve ser envolvida pelo menor AABB possível. Uma vez ocorrendo este fato, pode-se dizer que os nós filhos de uma subcena possuem seus AABBs com volumes somados geralmente menores que o volume da subcena que os compõe. Em consequência disso, há áreas vazias, também conhecidas como *void areas* [9], num AABB de uma subcena que não fazem parte dos nós filhos do nó correspondente a esta. A figura 3.1 ilustra a estrutura da BVH.

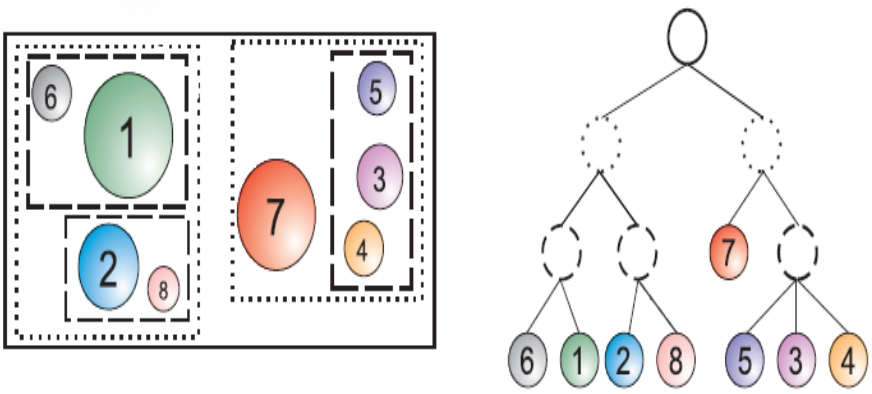

Fig. 3.1 Um exemplo de estrutura da BVH. A figura da esquerda ilustra como cada subcena é envolvida por um *bounding box*. À direita, a BVH é ilustrada como uma hierarquia de cenas. Ambas são a mesma cena com representações diferentes.

Através da figura 3.1, pode-se notar que, diferentemente da KD-Tree e da BIH, cada cena é composta pelo menor AABB possível. Os nós não subdivididos são denominados nós folhas da estrutura, onde sua composição é exclusivamente formada por objetos, também chamados de primitivas. As primitivas utilizadas neste trabalho são triângulos. Assim como outras estruturas de aceleração, a otimização do algoritmo de travessia do *ray tracing* através da BVH baseia-se em descartar os objetos 3D que não tem nenhuma probabilidade de serem exibidos na cena. Essa probabilidade é determinada pelo cálculo de intersecção do raio, traçado do observador com o AABB que envolve um ou mais objetos. A idéia é descartar todos os objetos cujo AABB não tem intersecção com o raio traçado. A figura 3.2 ilustra uma câmera virtual numa cena 3D, em dois casos de visualização determinada pela câmera.

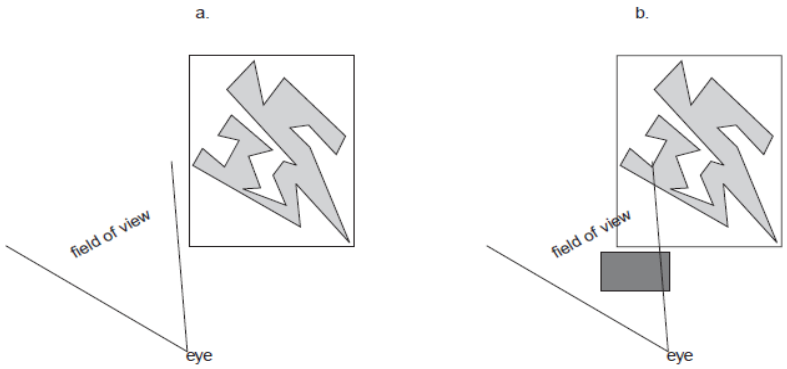

Fig. 3.2 Em (a) todos os objetos são desconsiderados para o algoritmo de travessia; em (b), uma parte da cena é considerada.

As próximas seções abordarão as etapas de construção e travessia da BVH, inclusive os algoritmos utilizados, a implementação e integração da estrutura de aceleração com o RT[2].

3.2. Construção

Na etapa de construção da estrutura, há os chamados parâmetros de construção nos quais são feitas atribuições de valores aos parâmetros que, por sua vez, podem alterar o desempenho nas etapas de construção e travessia. Os parâmetros são:

- Profundidade Máxima: Consiste em determinar até qual limite os nós internos são subdivididos; se a profundidade atual atingir a máxima, a subdivisão do nó é encerrada na estrutura.
- Número Máximo de Primitivas: Corresponde à quantidade limite de primitivas que cada folha pode possuir. Esse parâmetro, assim como o da Profundidade Máxima, também funciona como critério de parada da subdivisão de um nó, uma vez que se um nó folha possuir um número de primitivas igual ao limite, a subdivisão é encerrada.
- Profundidade Máxima para Decisão de Forma de Subdivisão: Na construção da BVH, existem duas maneiras de realizar a subdivisão de um nó, a saber Subdivisão Espacial [9] e Subdivisão de Objeto [9] (explicadas com maiores detalhes a seguir). Quando o nó a ser subdividido atinge a profundidade máxima para decisão, um só tipo de subdivisão é feito deste ponto em diante, que por sua vez se torna a subdivisão padrão. Esta subdivisão padrão corresponde à Subdivisão de Objeto.
- Coeficiente de Sobreposição Mínima: Um número real que contribui para determinar o limite de sobreposição que os AABBs dos nós filhos de um nó podem possuir entre si. Este limite contribui para a decisão dos critérios de subdivisão.

Há também os chamados processos candidatos à realização da subdivisão de uma cena. Primeiramente, será abordada a heurística utilizada neste trabalho para realização da subdivisão de um nó. A heurística utilizada para auxílio nas etapas da subdivisão foi a chamada

Surface Area Heuristic ou simplesmente SAH [14]. Esta heurística consiste em determinar a probabilidade ou custo de um nó e seus nós filhos serem interceptados por um raio de travessia; este custo depende das áreas das superfícies dos AABBs dos nós. A idéia é realizar uma subdivisão que possua o menor custo de SAH possível. O custo do SAH é calculado pela fórmula a seguir:

$$SAH = (TriangleCost\ /\ NodeArea) * (Child0Area * Numtris0 + Child1Area * Numtris1) + NodeCost \quad (1)$$

Onde:

NodeArea é o valor real da área da superfície do nó a ser subdividido.

NodeCost é o custo computacional do cálculo de intersecção de travessia em um AABB,

Child0Area é o valor real da área da superfície do nó filho da esquerda,

TriangleCost é o custo computacional de cálculo de intersecção de travessia em um triângulo,

Numtris0 é o número de triângulos do nó filho da esquerda,

Child1Area é o valor real da área da superfície do nó filho da direita E

Numtris1 é o número de triângulos do nó filho da direita.

Esta heurística é utilizada nos dois processos de determinação de subdivisão de uma cena.

O processo de Subdivisão de Objeto em um nó, utilizado neste trabalho, consiste em realizar ordenações de todas as primitivas (neste trabalho, triângulos) em função de cada eixo ordenado (x, y e z) e, para cada eixo, encontrar e atualizar a melhor configuração para a subdivisão através da expansão do nó filho da esquerda, adicionando progressivamente primitivas, e através da compressão do nó filho da direita, o qual está inicialmente com boa parte das primitivas da cena, removendo progressivamente primitivas. Para cada iteração, a qual é composta de expansões e compressões dos nós filhos a serem descobertos, é verificado o custo SAH para possível atualização de configuração da subdivisão. O pseudocódigo abaixo apresenta o algoritmo utilizado para encontrar a melhor configuração de subdivisão de uma cena.

Algoritmo 1: Encontrar a melhor configuração de subdivisão de uma cena

```
ConfiguracaoSubdivisao encontrarSubdivisao(){
        ConfiguracaoSubdivisao subdivisao;
        Primitivas primitivas = pegarPrimitivasDoNóAtual();
         Para cada dimensão correspondente aos eixos
                Ordenar(primitivas);
                AABB caixaDireita;
                Para cada primitiva em primitivas menos o primeiro
                elemento
                        expandirCaixa(caixaDireita, primitiva);
                }
                AABB caixaEsquerda;
                Para cada primitiva em primitivas com contador i
                        expandirCaixa(caixaEsquerda, primitiva);
                        areaCaixaDireita = areaTotal – caixaEsquerda.area;
                        float sah =  (CustoTrianguloTravessia/  areaTotal )*(
                        caixaEsquerda.area * i + areaCaixaDireita *
                        (primitivas.quantidade - i)) + CustoNoTravessia;
                        Se sah for menor que subdivisao.sah
                                AtualizarConfiguracao();
                        }
                }
        }
        Retornar subdivisao;
}
```

A atualização da configuração consiste em armazenar a dimensão correspondente a um dos eixos em que ocorre o menor SAH, caixaEsquerda, caixaDireita, número de elementos em cada caixa e o próprio SAH para uma possível posterior aplicação de subdivisão com estes dados. A subdivisão padrão se aplica com mais eficiência em cenas cuja distribuição de objetos 3D é heterogênea, já a Subdivisão Espacial, que será explicada na sequencia,

se aplica melhor em cenas com distribuição homogênea de objetos 3D. Na subdivisão padrão podem ocorrer casos de sobreposição entre os nós filhos. O espaço ocupado por esta sobreposição, sendo significativa em relação ao volume do AABB do nó raiz, é condição para realizar uma subdivisão que seja mais eficiente no momento da etapa de travessia. A Subdivisão Espacial elimina a sobreposição entre os AABBs dos nós filhos, sendo também conhecida como *binning*. Ela consiste em dividir o espaço, do nó a ser subdividido, em *voxels* [9](volumetric pixels) ou *spatial bins*, que são blocos cúbicos de volumes iguais. Daí surge um novo parâmetro de construção denominado *número de blocos* ou *bins* a dividir o espaço. A figura 3.3 ilustra um exemplo de divisão do espaço em *bins*.

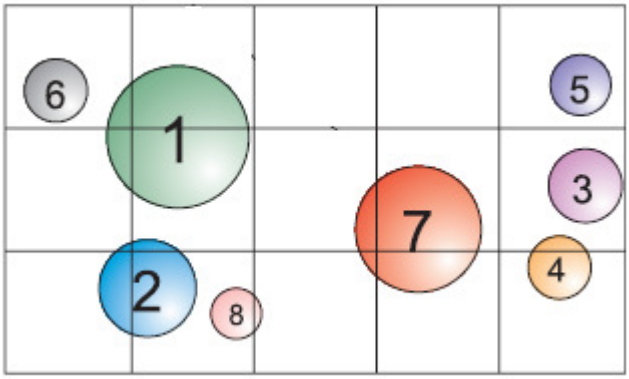

Fig. 3.3 Divisão do espaço de um nó em *bins* de tamanhos idênticos.

Em seguida, é realizada uma varredura em todas as primitivas para checar a possível ocorrência da ocupação de mais de um *bin* ou um determinado *bin* por uma das primitivas. Ao ocorrer este caso, são identificados quais *bins* estão ocupados. Nota-se que no início se sabe apenas do tamanho de cada *bin* e a quantidade de *bins* , mas suas posições no espaço não foram inicializadas e que nem todos os *bins* serão necessariamente utilizados para envolver as primitivas do nó, uma vez que podem surgir *bins* vazios. Logo, somente *bins* que envolvem primitivas terão posições no espaço inicializadas e estes influenciarão a determinação dos AABBs dos possíveis futuros nós filhos. O termo “possíveis futuros nós filhos” deve-se à influência do SAH na configuração da subdivisão, pois mesmo eliminando sobreposições, através do *binning*, pode ocorrer que o

SAH é maior do que o SAH da subdivisão padrão. Depois da varredura de todas as primitivas contidas no nó, é feita mais uma varredura, só que desta vez em todos os *bins* com posições inicializadas. *Bins* de posições não inicializadas são descartados. Durante a varredura, ocorre algo semelhante à subdivisão padrão que é justamente a expansão do nó filho da direita, essa expansão ocorre com adições dos bins no nó; uma vez expandido com um *Bin*, a menos do número total de *bins*, pois o *bin* que sobrou está no nó filho da esquerda, ocorre a checagem do SAH da configuração atual, e ocorre progressivamente a compressão, através da remoção de *bins*, do nó filho da direita, e a expansão do nó filho da esquerda para checar e determinar a melhor configuração da subdivisão espacial, aquela com menor SAH. Para a determinação do SAH de uma configuração é aplicada a mesma fórmula apresentada anteriormente nesta seção.

Nem sempre é válido realizar a subdivisão do nó, e sim tomá-lo como uma folha da estrutura, devido ao SAH da folha ser menor. O SAH da folha consiste em:

$$leafSAH = numTriangles * triCost \quad (2)$$

Onde:
NumTriangles é a quantidade de triângulos presentes no nó e
TriCost é o custo computacional da travessia no triângulo.

Vale ressaltar que neste trabalho foram utilizados apenas triângulos como primitivas, daí a consideração somente destes nas fórmulas do SAH.

A construção da BVH consiste nos dois conceitos de subdivisão discutidos anteriormente e de custo SAH como elemento chave. O algoritmo abaixo apresenta a construção da estrutura sendo feita recursivamente.

Algoritmo 2: Construção recursiva da estrutura

```
Construir (dadosDoNó, nível){
    Se (NúmeroDeTriangulos (dadosDoNó) < número maximo de
        triangulos ou atingiuNivelLimite(nível)){
        criarFolha(dadosDoNó);
        retornar;
```

```
        }
        area = dadosDoNó.aabb.area;
        folhaSAH = custoDeTravessiaTriangulo *
                dadosDoNó.númeroDeTriangulos;
        subdivisãoObjetoConfig = encontrarSubdivisãoObjeto(dadosDoNó);
        Se (nivel < profundidadeMaximaDeDecisãoDeSubdivisao){
                Sobreposição =
                        intersecção(subdivisãoObjetoConfig.aabbEsquerda,
                        subdivisãoObjetoConfig.aabbDireita);
                Se (sobreposicao < areaNoRaiz * coeficiente de sobreposicao
                        minima){
                        subdivisaoEspacialConfig =
                                encontrarSubdivisaoEspacial(dadosDoNó);
        }
        SAH_Minimo = minimo(folhaSAH, subdivisãoObjetoConfig.SAH,
                subdivisaoEspacialConfig.SAH);
        //se for mais vantagem criar folha
        Se(SAH_Minimo == folhaSAH){
                criarFolha(dadosDoNo);
        }
        nóAtual;
        incializarNó(nóAtual, dadosDoNó);
        adicionarEmNós(nóAtual);
        Se(SAH_Minimo == subdivisaoEspacialConfig.SAH){
                realizarSubdivisão(subdivisaoEspacialConfig);
        }
        Se(nao ocorreu subdivisao){
                realizarSubdivisão(subdivisãoObjetoConfig);
        }
        Construir (dadosDoNóFilhoDireita, nível+1);
        Construir (dadosDoNóFilhoEsquerda, nível+1);
}
```

Na seção seguinte será abordada a etapa de travessia, que será constantemente utilizada com base na estrutura construída na etapa de construção.

3.3. Travessia

A travessia ou busca trata da etapa em que o traçado de raios do *ray tracing* é feito e, para cada raio criado, ocorre a busca de uma possível intersecção do raio com uma das primitivas da cena. A partir deste ponto, aparece a utilidade da estrutura construída na etapa anterior, pois a verificação da intersecção de raio e primitiva é reduzida drasticamente. Isso se deve aos AABBs criados em cada nó da estrutura, pois, tratando-se de estruturas simples e que envolvem primitivas, evita-se verificações de intersecção desnecessárias. Neste trabalho foi realizada a travessia baseada em estrutura de pilha, utilizada na travessia padrão da *KD-Tree*. Na etapa de travessia as seguintes variáveis devem ser atualizadas:

- T: Número real que indica exatamente em que distância o raio traçado interceptou uma primitiva. Caso não ocorra uma atualização isto significa que não houve uma intersecção. Essa distância é utilizada no RT^2 para interpolar um ponto em função do ponto de origem do raio e da direção, para a etapa de *shading*.
- U: Número real que indica uma coordenada baricêntrica do ponto de intersecção de um polígono.
- V: Número real que indica outra coordenada baricêntrica do ponto de intersecção de um polígono; estas coordenadas são utilizadas também na etapa de *shading* do RT^2.
- Primitiva Interceptada: Número inteiro que indica o índice da primitiva mais próxima interceptada contida na lista de primitivas.

O algoritmo de travessia também envolve cálculos de intersecção do raio com AABB. 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 t mínimo, e outro que indica a distância do raio ao ponto de saída da caixa, chamado da t máximo. O processo para determinar a

intersecção do raio com alguma primitiva primeiramente verifica a intersecção do raio com o AABB da cena toda ou o nó raiz e realiza a travessia em caso de intersecção. Durante a travessia, a variável correspondente ao nó atual da estrutura onde está ocorrendo a travessia aparece. 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 (não folha) 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. Mas, 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 t mínimo de intersecção, surgindo o problema de a travessia neste nó escolhido falhar, não encontrando intersecção e o de outro nó com t mínimo maior ser desprezado. Para isto, o algoritmo utiliza uma estrutura de pilha para armazenar os nós com t mínimo 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. O algoritmo abaixo ilustra esse processo de travessia; este algoritmo tem como retorno o índice da primitiva de menor distância de intersecção e usa uma estrutura de pilha para armazenar os nós não verificados durante a travessia.

Algoritmo 3: Travessia

```
TravessiaBVH(raio, t, u, v){
        t = INFINITO;
        índicePrimitiva = INVALIDO;
        tMinimo, tMaximo;
        uTemporario, vTemporario;
        //calcula intersecção do raio com o nó corrente atualizando
        //tMinimo e tMaximo.
        intercepta = IntersecçãoRaioCaixa(raio, noAtual.aabb, tMinimo,
        tMaximo);
```

```
nivelDaPilha = 0;
pilha[profundidade maxima];
se(noAtual for folha){
        Para cada primitiva contida em noAtual{
        //calcula o t da intersecção do raio com triangulo assim como
        //tambem coordenadas  baricentricas u e v
                tTemporario = intersecçãoRaioTriangulo(raio,
                primitiva, uTemporario, vTemporario);
                Se tTemporario >= 0 e tTemporario <= t{
                        t = tTemporario;
                        índicePrimitiva = obterIndice(primitiva);
                        u = uTemporario;
                        v = vTemporario;
                }
        }
        nivelDaPilha = nivelDaPilha – 1;
        Se nivelDaPilha != -1{
                noAtual = pilha[nivelDaPilha];
        }caso contrario{
                Retorne índicePrimitiva;
        }
}caso contrario{
        filhoEsquerda = noAtual.filhoEsquerda;
        filhoDireita = noAtual.filhoDireita;
        intercepta0 = IntersecçãoRaioCaixa(raio filhoEsquerda,
        tMin0,tMax0);
        intercepta1 = IntersecçãoRaioCaixa(raio filhoEsquerda,
        tMin1,tMax1);
        Se intercepta0 e intercepta1{
                Se tMin1 < tMin0{
                        Pilha[nivelDaPilha] = filhoEsquerda;
                        noAtual = filhoDireita;
                }caso contrario{
```

```
                        Pilha[nivelDaPilha] = filhoDireita;
                        noAtual = filhoEsquerda;
                }
                nivelDaPilha = nivelDaPilha + 1;
        }caso contrario{
                Se intercepta0{
                        noAtual = filhoEsquerda;
                }caso contrario se intercepta1{
                        noAtual = filhoDireita;
                }caso contrario{
                        nivelDaPilha = nivelDaPilha – 1;
                        Se nivelDaPilha != -1{
                                noAtual = pilha[nivelDaPilha];
                        }caso contrario{
                                Retorne índicePrimitiva;
                        }
                }
        }
    }
    Retorne índicePrimitiva;
}
```

Na seção seguinte é realizada uma comparação da BVH com a estrutura da *KD-Tree*, apresentada no capítulo 2, levando em considração sua performance nas etapas de construção e travessia.

3.4. Comparativo com a KD-Tree

Durante o desenvolvimento deste trabalho, a BVH apresentou a etapa de travessia com performance ligeiramente inferior a travessia da *KD-Tree Standard* e significativamente inferior a da *KD-Tree ropes*. Estes resultados se baseiam nos testes feitos neste trabalho, apresentados no capítulo 4. Vale ressaltar que todas as cenas testadas em ambas estruturas são compostas por um só objeto triangularizado, pois o RT^2 não permite

atualmente carregar mais de um objeto deste tipo. No capítulo 4 serão apresentados os resultados das cenas testadas no RT^2 neste trabalho, com uma comparação mais detalhada entre as estruturas da BVH e *KD-Tree*, levando em conta o tempo de travessia, tempo de pré-processamento ou construção, consumo de memória, entradas utilizadas com respectiva quantidade de polígonos.

3.5. Implementação do Algoritmo BVH em CUDA

Neste trabalho, apenas a etapa de travessia da BVH foi implementado na GPU. O algoritmo de construção foi implementado em CPU, uma vez que possui chamadas recursivas não suportadas por CUDA. Foi implementada também uma versão do algoritmo de travessia em CPU, para posterior comparativo de performance com a versão em GPU que será abordada nesta seção.

Para um bom ganho de performance em CUDA, a etapa de travessia foi implementada considerando o uso e acesso de memória local, acesso a memória global e operações de alto e baixo custo. Para realizar a implementação com tais considerações foi aproveitada a estrutura de pilha já contida no RT^2 para implementar o algoritmo de travessia da BVH, evitando a criação de uma pilha na memória local. Foi realizado também o uso de memórias de textura para melhorar o tempo de acesso aos dados em placas de vídeo mais antigas em relação a série GTX. Na implementação, foi também evitado uso contínuo de operações de divisão e módulo, pois estão entre as operações mais custosas em GPU. A utilização destas operações foi feita mais em escopos maiores de código para armazenamento do resultado em variáveis para posterior utilização. Diferentemente da implementação em CPU, os nós da BVH foram organizados em um vetor de float4, tipo nativo de CUDA. No vetor cada sequencia de três elementos corresponde a um nó, onde o primeiro corresponde aos dados do nó como índices dos nós filhos e identificador de primitiva para verificação se o mesmo é um nó folha. O segundo e o terceiro elementos correspondem aos dois pontos que delimitam o volume do AABB do nó.

3.6. Integração da BVH no RT²

Para integrar os algoritmos e dados da BVH no RT², foram criados tipos de dados para armazenamento de informações na etapa de construção, funções de construção e de travessia. Nesta seção serão apresentados os arquivos criados e modificados para a integração da BVH no RT². No projeto, foi criado um arquivo de utilidades chamado Util.hpp, que contém a definição do AABB específico de um nó da BVH, assim como funções que auxiliam operações de expansão e compressão da caixa. Este arquivo corresponde a classe AABB que possui como atributos os dois pontos que delimitam seu volume. Este mesmo arquivo possui rotinas de cálculo de intersecção raio-AABB e raio-triângulo. Há também o arquivo BVHNode.hpp que define como um nó é estruturado, possuindo dois valores do tipo int que correspondem aos índices dos nós filhos, um booleano para identificar se um nó é do tipo folha e possui um AABB correspondente a seu volume no espaço, assim como uma definição do número máximo de primitivas que o nó pode carregar. A figura 3.4 ilustra um código C/C++ da definição do nó. A estrutura BVHNode possui dois valores inteiros (child0 e child1) que indicam os índices dos nós filhos contidos num array de nós, um valor booleano (isLeaf) que indica se este nó é do tipo folha e um dado do tipo AABB do nó.

```
#ifndef BVH_NODE_HPP
#define BVH_NODE_HPP

#include "Util.hpp"

namespace RayTracer{
typedef struct _BVHNode
{

#define MAX_LEAF_COUNT 3

public:
    int child0,child1;
    bool isLeaf;
    AABB bounds;
} BVHNode;

}

#endif
```

Fig. 3.4 Estrutura de um nó da BVH.

Há também o arquivo SplitBuilder.hpp, o qual possui estruturas utilizadas de forma temporária durante a etapa de construção, tais como configurações de subdivisão, assim como os parâmetros de construção da BVH (ver seção 3.2). Este arquivo contém o próprio algoritmo de construção que é implementado no arquivo SplitBuilder.cpp. Há também o arquivo BVH.hpp, que reutiliza o arquivo SplitBuilder.hpp para realizar a construção e contém o algoritmo de travessia. Nesse arquivo ocorre a estruturação dos nós em formato de vetor de BVHNode assim como sua conversão em um vetor de float4 para representá-lo na versão em GPU, para otimização da utilização de memória em GPU e a permissão para utilizar memória de textura. Ainda neste arquivo, há também a rotina que executa o algoritmo de travessia em CPU. Para a implementação da travessia em GPU foi realizada a edição do arquivo CUDATracerThread.cpp, arquivo do próprio RT^2 onde ocorrem cópias de dados de CPU para GPU. Neste, foi feita a passagem do vetor de float4 correspondente aos nós da BVH para a memória global da GPU, assim como a criação de um dado na memória de textura desse vetor para uma melhor velocidade de acesso ao mesmo. Foi acrescentado mais um modo de travessia na arquitetura de GPU no RT^2; este modo corresponde a BVH_TRAVERSAL, acrescido no conjunto do tipo enumerador do RT^2 chamado TraversalMode. No arquivo traversals.cu foi implementada a função de travessia da BVH de maneira otimizada, explicada na seção anterior, assim como funções auxiliares de intersecção raio-caixa e de acesso a um nó da estrutura no arquivo GlobalFunctions.cu.

4. Resultados

Este capítulo apresenta os resultados comparativos de desempenho das estruturas da *KD-Tree* e da BVH no *ray tracer* RT^2 interativo. A comparação foi realizada utilizando o algoritmo de construção por ordenação com heurística SAH tanto da BVH como da *KD-Tree*. Foram também apresentados resultados de tempo de construção da *KD-Tree* utilizando o algoritmo *min-max-binning*.

O processador utilizado para a realização dos testes da etapa de construção foi o Intel Core i7 3.0GHz e, na etapa de travessia, foi utilizada uma placa gráfica multigpu da série GTX de dois dispositivos, a saber a Geforce 480 GTX com tecnologia SLI [11].

A seguir, é mostrada uma sequencia de imagens da execução do projeto, seguidas de gráficos apresentando os dados de desempenho. Cada imagem corresponde a um modelo de entrada e os gráficos são os dados de performance correspondentes.

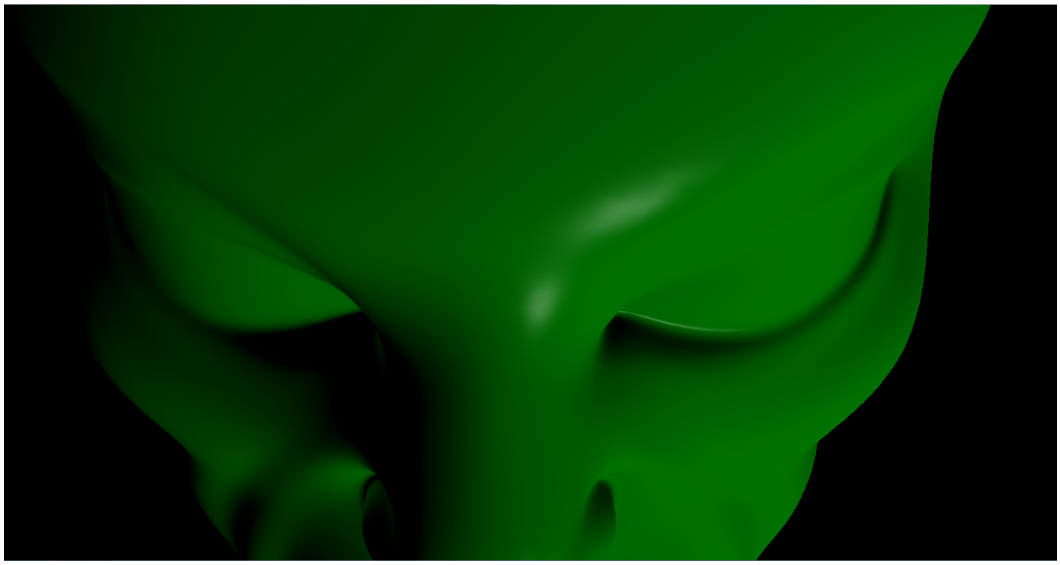

Fig. 4.1 Modelo Alien: 32 mil polígonos; resolução: 1408x768.

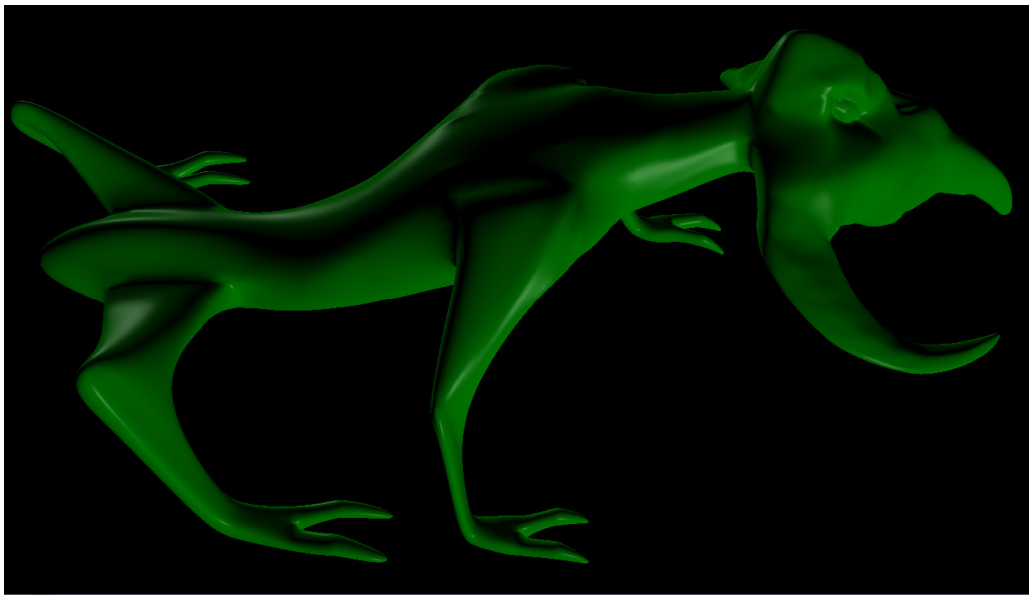

Fig. 4.2 Modelo Dino: 107 mil polígonos; resolução: 1408x768.

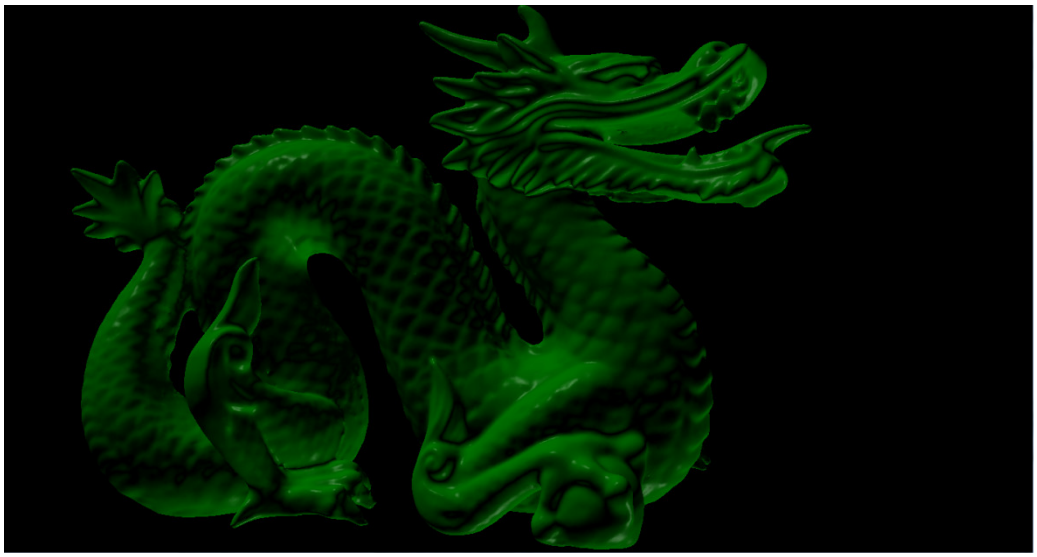

Fig. 4.3 Modelo Dragon: 847 mil polígonos; resolução: 1408x768.

Os dados coletados consistem na memória consumida, tempo de construção e quantidade de quadros por segundo na travessia. Vale ressaltar que as cenas testadas consistem em uma distribuição de um único objeto na cena.

Os valores dos parâmetros de construção utilizados na estrutura da BVH foram: Profundidade Máxima: 440, Número Máximo de Primitivas: 3. Os parâmetros "Profundidade Máxima para Decisão de Forma de Subdivisão" e "Coeficiente de Sobreposição Mínima" não foram utilizados em virtude do algoritmo de subdivisão por *binning* não estar completamente implementado neste trabalho.

Na cena com o modelo do Alien, apesar da BVH apresentar a vantagem de cada nó da estrutura apresentar o seu AABB compacto, a *KD-Tree* ganhou em 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 raiz, consiste em checar em qual lado do plano de corte de cada nó interno o mesmo se encontra. Já a verificação da BVH consiste em checar a intersecção do raio com a AABB em cada nó. Como só existe um objeto na cena (não havendo áreas vazias na cena), este fato favorece o resultado esperado de tempo de travessia em ambas as estruturas. No modelo Dino, estes resultados devem-se ao mesmo caso do modelo anterior. O que chama atenção é que o ganho de tempo de travessia da *KD-Tree standard* em relação a BVH diminuiu de 18% (Alien) para 3%. Isso deve-se às áreas vazias do modelo, onde ocorre uma certa distribuição (apesar de existir um só objeto) das primitivas do mesmo, favorecendo o surgimento de "*void areas*" na BVH. Os resultados de travessia com o modelo Dragon apresentaram dados que favoreceram a *KD-Tree*, pois, como foi dito nos dois casos anterios, este modelo, assim como o Alien, é compacto, favorecendo o ganho em tempo da *KD-Tree.* Pode-se notar que a *KD-Tree ropes* oferece maior performance na travessia em relação a BVH e a *KD-Tree standard*; isso deve-se ao fato da mesma utilizar estruturas adicionais na *KD-Tree* que auxiliam o ganho de performance. Os resultados referentes à etapa de travessia podem ser vistos na figura 4.4.

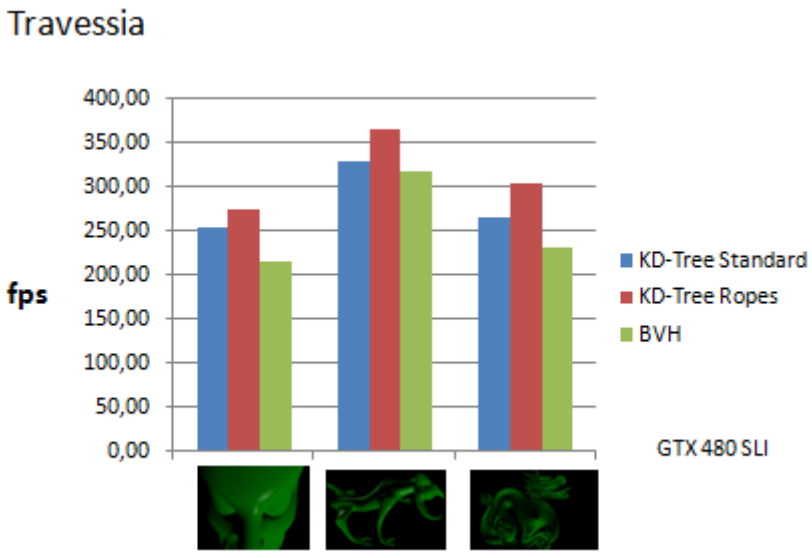

Fig. 4.4 Gráfico com dados comparativos de performance em travessia da KD-Tree e BVH.

O tempo de construção da BVH ficou entre o tempo da *KD-Tree sort* e o tempo da *KD-Tree min max* nos modelos do Alien e do Dino, enquanto que no modelo do Dragon, o tempo de construção foi o maior. O que chama atenção é que o tempo de construção da BVH em relação a *KD-Tree sort* oscilou entre um ganho de 55% (Alien) até um ganho de 75% (Dino), e depois teve queda com perda de 13% (Dragon). Os resultados estão apresentados na figura 4.5.

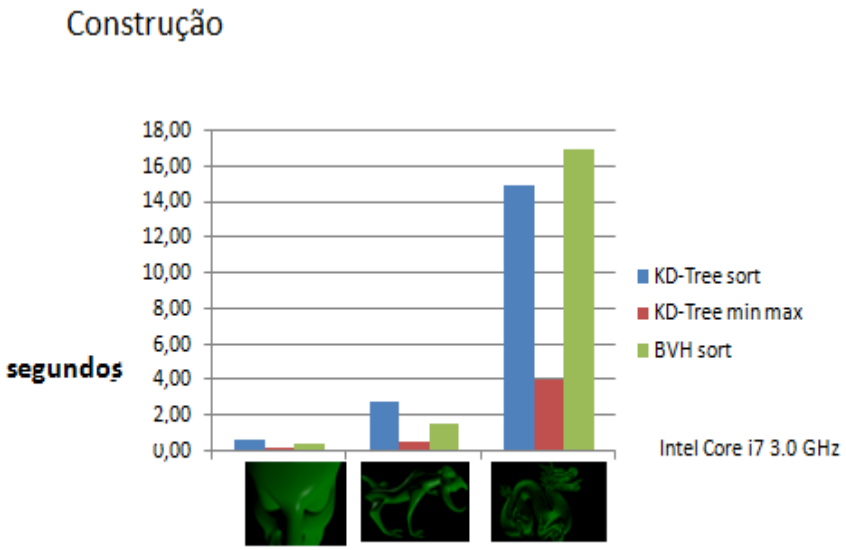

Fig. 4.5 Gráfico com dados comparativos de performance em construção da KD-Tree e BVH.

O uso de memória da *KD-Tree standard* ficou com o menor valor nos modelos do Alien e do Dragon devido a mesma utilizar planos de corte como dados dos nós enquanto que na BVH ocorre o uso de AABBs para armazenamento de dados dos nós. Na *KD-Tree ropes* ocorre a criação de AABBs para os nós folhas e, para cada um desses nós, criam-se seis referências para os nós folhas vizinhos, daí o comsumo grande de memória da *KD-Tree ropes*. No modelo do Dino, o consumo ligeiramente maior de memória da *KD-Tree standard* em relação ao consumo da BVH deve-se a existência de regiões "com pontas" do Dino, que fovorecem a criação de mais nós na estrutura. A figura 4.6 apresenta os resultados.

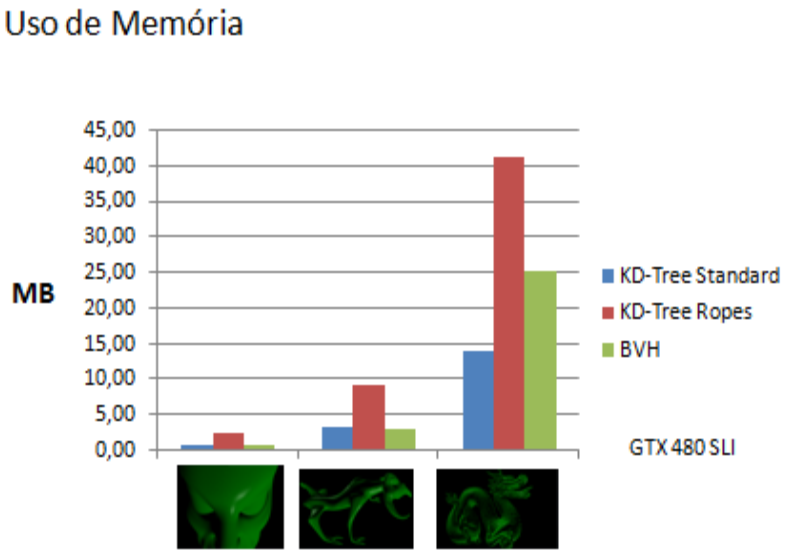

Fig. 4.6 Gráfico com dados comparativos de uso de memória de vídeo da KD-Tree e BVH.

Com os resultados acima, vale observar que a *KD-Tree ropes* apresenta ganho de performance maior na travessia, mas apresenta desvantagem quanto ao consumo de memória.

5. Conclusão

Este trabalho apresentou um estudo sobre estruturas de aceleração aplicadas ao *ray tracing* em tempo real. O objetivo principal desta pesquisa foi analisar uma estrutura de aceleração, a BVH, comparando-a com a estrutura pré-existente no projeto RT², a *KD-Tree*. Após a execução deste trabalho, notou-se que a BVH obteve o pior desempenho no tempo de travessia nas cenas testadas. Quanto ao tempo de construção, a BVH apresentou resultados medianos em relação à construção da *KD-Tree* utilizando os algoritmos de *sort* e *min max*. Quanto ao consumo de memória, a BVH apresentou resultados medianos se comparada com a *KD-Tree ropes* e *KD-Tree standard*.

Apesar dos resultados não favorecerem a BVH, não se pode comprovar se a construção da BVH é mais lenta ou mais rápida do que a construção da *KD-Tree*, uma vez que não foram utilizados os algoritmos e heurísticas mais simples de construção da estrutura. O mesmo pode-se dizer a respeito do desempenho da travessia, pois somente foram testadas cenas com pouca distribuição (um só objeto), surgindo a possibilidade de que em cenas com grande quantidade de áreas vazias o desempenho da BVH seja favorecido. Logo, pode-se afirmar que a BVH apresenta menor desempenho em travessia em cenas compactas e de pouca distribuição em relação a *KD-Tree*.

5.1. Trabalhos Futuros

Como trabalhos futuros são listadas as seguintes melhorias e pesquisas a serem realizadas:

- Suporte a um conjunto maior de primitivas, tais como superfícies quádricas, curvas e superfícies de bézier e NURBS.
- Suporte a modos de construção adicionais, tais como Subdivisão do Objeto no maior eixo e construção com Subdivisão Espacial com *min-max binning* em todos os eixos e

no maior eixo, suporte a construção híbrida que mistura *binning* com *sort* e pesquisa de outros algoritmos de construção e heurísticas.

- Pesquisa e possível incorporação de mais modos de travessia e otimizações.
- Testes com cenas de maior distribuição de objetos.

Por fim, há a intenção do autor de encontrar argumentos fortes que fujam da abordagem utilizada pela técnica da rasterização e incorporar o *ray tracing*, ou qualquer outro algoritmo que utilize uma estrutura de aceleração, no *pipeline* de renderização, tendo como técnica base a iluminação global para melhoria significativa do desempenho e de levantar vantagens e desvantagens da estrutura de aceleração da BVH.

Referências

[1]. Introdução a Computação Gráfica: Rasterização. http://www.lcg.ufrj.br/Cursos/COS-751/rasterizacao-pdf. Acessado em 24 de Novembro de 2010.

[2]. Introdução ao Hardware Gráfico. http://www.cin.ufpe.br/~marcelow/Marcelow/programacao_cg_files/historia-hw-grafico.pdf. Acessado em 24 de Novembro de 2010.

[3]. Unity3d. http://unity3d.com. Acessado em 25 de Novembro de 2010.

[4]. OpenGL Shading Language. http://cin.ufpe.br/~marcelow/Marcelow/programacao_cg_files/GLSL_gsm_sap_vap2.pdf. Acessado em 25 de Novembro de 2010.

[5]. Artur Lira dos Santos, RT2-Real Time Ray Tracer, UFPE.

[6]. Quádrica. http://pt.wikipedia.org/wiki/Quádrica. Acessado em 27 de Novembro de 2010.

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

[8]. Appel, A., *Some techniques for shading machine renderings for solids.* AFIPS. SJCC, 37-45, 1968.

[9]. Gordon Muller, *Object Hierarchies for Efficient Rendering*. Technischen Universität Braunschweig, Tese de Doutorado, 2003.

[10]. O pipeline de renderização. http://cin.ufpe.br/~marcelow/Marcelow/programacao_cg_files/review-pipeline.pdf. Acessado em 25 de Novembro de 2010.

[11]. NVIDIA CUDA C Programing Guide. http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf. Acessado em 29 de Novembro de 2010.

[12]. Artur Santos, João Marcelo Teixeira, Thiago Farias, Veronica Teichrieb, Judith Kelner, *Understanding the Efficiency of KD-tree Ray-Traversal*

Techniques over a GPGPU Architecture, Federal University of Pernambuco, Computer Science Center, Virtual Reality and Multimedia Research Group, Recife, Brazil.

[13]. Casten Wachter, Alexander Keller, *Instant Ray Tracing: The Bounding Interval Hierarchy*, University of Ulm, Alemanha.

[14]. Ingo Wald, Vlastimil Havran, *On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N)*, SCI Institute, University of Utah, Czech Technical University in Prague.

[15]. 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.

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

[17]. Cg language Specification. http://http.developer.nvidia.com/Cg/Cg_language.html. Acessado em 10 de Dezembro de 2010.

[18]. Reference for HLSL. http://msdn.microsoft.com/en-us/library/bb509638%28v=vs.85%29.aspx. Acessado em 10 de Dezembro de 2010.

[19]. The Industry's Foundation for High Performance Graphics. http://www.opengl.org. Acessado em 10 de Dezembro de 2010.