

UNIVERSIDADE FEDERAL DE PERNAMBUCO

GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
CENTRO DE INFORMÁTICA

2009.1



RT² - Real Time Ray Tracer

TRABALHO DE GRADUAÇÃO

Aluno
Orientadora

Artur Lira dos Santos
Veronica Teichrieb

{als3@cin.ufpe.br}
{vt@cin.ufpe.br}

23 de Junho de 2009

Resumo

A incessante evolução dos microprocessadores tem levado a um ritmo cada vez maior ao surgimento de aplicativos e dispositivos eletrônicos com as mais diversas funcionalidades. É interessante observar que muitas dessas novas ideias foram, na verdade, conceitos já discutidos há três ou mais décadas atrás pelo meio acadêmico e militar. Inclusive, algumas dessas ideias foram transformadas em propagadas para a sociedade daquele tempo, como parte do imaginário em filmes de ficção científica, como o raio *laser* e a estranha porta “mágica” que se abre sozinha, enquanto que hoje o estranho é quando a tal porta não se abre ao entrarmos em um aeroporto, por exemplo. É fácil então notar que este grande número de novidades aos poucos não só altera a maneira do homem realizar uma certa atividade, como também influencia na sua percepção do ambiente.

Este Trabalho de Graduação está contextualizado em uma das mais antigas áreas da Computação, que lida diretamente com a percepção humana: a Computação Gráfica. Com o propósito de trazer conceitos básicos de foto-realismo para o contexto de simulação em tempo real, este trabalho propõe o RT² – *Real Time Ray Tracer*. O RT² é uma biblioteca que possibilita o uso de técnicas de síntese de imagens de alta qualidade como o *ray tracing* para a criação em *software* de um *pipeline* gráfico especialmente designado para aplicativos interativos, como jogos e programas de simulação em geral. Além disso, este trabalho descreve uma extensa análise do estado da arte sobre *ray tracing* em placas gráficas e propõe uma nova abordagem para buscas em *kD-Trees*.

Palavras-chave: *ray tracing*, *kD-Tree*, *CUDA*, tempo real.

Agradecimentos

Última parte a ser escrita. Depois de mais e mais folhas virtuais, chego à esta, que antes mesmo da escrita da primeira página deste documento, eu já sabia que ia ser a parte que iria me dar mais trabalho em escrever. Há aqui uma grande dificuldade de colocar em palavras o sentimento de alegria e verdadeira gratidão que eu sinto ao ter o apoio de pessoas que me acompanharam nessa luta. Luta? Não, para mim eu acho que foi uma guerra! Afinal, em que outra situação você passa noites e noites acordado atento, com altas doses de cafeína para conseguir ficar de pé? Só em guerra... e na universidade! Para encarar esta guerra, tive a sorte de contar com a ajuda direta e indireta de muita gente, que espero não esquecer de citar aqui.

A primeira fatia do bolo eu dedico à meus pais, que tiveram uma vida de luta e sacrifícios para oferecer aos filhos aquilo que seus pais não foram capazes de oferecer, por conta de um mundo injusto como o nosso, onde a aleatoriedade de onde você nasce muitas vezes impõe dificuldades imensuráveis a serem superadas. A meus pais, verdadeiros heróis!

A segunda leva de agradecimentos vai às minhas duas irmãs, a Rê e a Rô, dois exemplos de pessoa que tento sempre seguir. À minha tia Socorro, que assim como meus pais, sempre esteve presente em qualquer problema.

Agradecimento especialíssimo à professora Verônica, orientadora deste trabalho de graduação, que participou desde o começo deste projeto e com suas detalhistas revisões beirando sem exageros à perfeição. Obrigado, VT! Agradeço também à Joma, que foi de grande auxílio no processo de implementação do projeto, tendo a visão de águia daquilo que iria funcionar e do que não teria chance alguma de dar certo, me poupando um bocado de tempo. Sem esquecer de Mouse

RT² - Real Time Ray Tracer

e Pedro, duas outras figuras de extrema criatividade, que me ajudaram bastante na implementação.

Queria agradecer em especial à professora Judith, que no meio da minha graduação me deu a oportunidade de fazer parte de um grupo que tanto tenho orgulho, respeito e que admiro, o GRVM.

A todos do GRVM, em especial a Gabriel (o cara), Ronaldo (o “*bizarro*”), Rodrigo (meu avô querido), Mozart (o “*traidor*”), Chico (o mestre), Lucas (Beça) e Java (Thiago), companheiros do dia a dia e do tão esperado momento treloso =).

A meus amigos pro resto da vida: Luana, Manel, Lipinho, Samara, Jozie, Guiminha, Chico e mais uma tuia de gente do Colégio de Aplicação. Pessoal, “tô” terminando a graduação! Quem sabe eu não “*fulero*” menos com vocês agora? =D!

Gostaria de finalmente agradecer a Blizzard por ter atendido aos meus pedidos em adiar o lançamento de Starcraft 2 para depois da minha conclusão do curso. Afinal, como já diziam: “*Hell, it's about time!*”.

“*O computador foi feito para resolver todos os problemas que a gente nunca teve*”

Autor desconhecido

Índice

RT² - REAL TIME RAY TRACER	1	
1.	INTRODUÇÃO	6
1.1.	Objetivo	6
1.2.	Estrutura do Documento.....	7
2.	CONTEXTO	8
2.1.	Rasterização – A base do mercado de Jogos 3D	8
2.2.	Ray Tracing	10
2.3.	Estruturas de Aceleração	13
2.4.	Ray Tracing em GPU	18
2.5.	Programação em GPU	27
2.6.	CUDA Framework	30
3.	RT² - REAL TIME RAY TRACER	33
3.1.	Arquitetura de <i>Software</i> do RT ²	36
3.2.	Ciclo de Renderização do RT ²	38
3.3.	Núcleo do RT ²	40
3.3.1.	Tratamento de Raios Secundários.....	44
3.3.2.	Espaços de Memória	45
3.4.	Ropes++.....	46
4.	RESULTADOS	49
4.1.	Análise Comparativa de Travessias em <i>kD-Tree</i>	50
4.2.	Análise Comparativa do Ropes++ em CPU e GPU	55
5.	CONCLUSÕES	56
5.1.	Trabalhos Futuros.....	57
6.	REFERÊNCIAS BIBLIOGRÁFICAS	58

1. Introdução

Com o crescente poder computacional dos microprocessadores e a constante busca por cenas 3D renderizadas de forma mais realística, técnicas de síntese de imagem por iluminação global e de alto custo computacional como *ray tracing* (traçado de raios) [1] passaram a atrair a atenção dos grupos de pesquisa da área de Computação Gráfica em tempo real. As técnicas baseadas em iluminação global oferecem, de maneira simples e elegante, o suporte a materiais reflexivos e transparentes, além de efeitos de sombra e iluminação indireta, como *color bleeding* [2]. Tais atributos enriquecem consideravelmente o grau de realismo da imagem. Com os processadores da atualidade já é possível processar em tempo real algumas dessas custosas técnicas, sendo este o caso da renderização utilizando *whitted ray tracing* [3].

Esta possibilidade de processar *ray tracing* em tempo real gerou recentemente uma silenciosa disputa entre a Intel [4] e a NVIDIA [5], duas “gigantes” da indústria de microprocessadores. Ambas visualizam [6] [7] num futuro próximo a possibilidade do uso de *ray tracing* em aplicativos interativos como jogos e sistemas de simulação 3D, de tal maneira que tais empresas já dão alguns sinais de que muito em breve grandes mudanças na forma de se renderizar jogos e aplicativos interativos 3D ocorrerão.

1.1. Objetivo

Neste contexto, visando um novo *pipeline* de renderização, foi concebido neste Trabalho de Graduação o sistema RT² – Real Time Ray Tracer, propondo renderizar em tempo real cenas 3D ao utilizar *ray tracing* como principal técnica de renderização. Assim, o RT² tem a proposta direta de substituir o uso de

rasterização [2] (principal técnica utilizada em aplicativos interativos) por técnicas de iluminação global.

Para alcançar tal objetivo, foi realizado um estudo do estado da arte em *ray tracing*, assim como de algoritmos relacionados, como estruturas de dados de divisão espacial. Com relação a estas estruturas, foi dada especial atenção à *kD-Tree* [8], um tipo de árvore por partição binária de extrema importância para um eficiente sistema de *ray tracing*. Como tais algoritmos demandam alto poder computacional, também foi feito um estudo da arquitetura de CUDA [9] (*Compute Unified Device Architecture*), que possibilita a programação de propósito geral em placas gráficas (GPGPU – *General-Purpose computation on Graphics Processing Units*) de alta performance da NVIDIA. Assim, este documento descreve todo o processo de pesquisa, planejamento e implementação do RT².

1.2. Estrutura do Documento

Este documento está organizado da seguinte forma. Na sequência, o capítulo 2 descreve todo o contexto relacionado ao RT², como conceitos básicos sobre *ray tracing* e travessias em *kD-Tree*. Também é descrita a arquitetura de CUDA, além do levantamento de trabalhos relacionados. O capítulo 3 descreve a arquitetura do RT², detalhando toda a implementação do sistema. O capítulo 4 demonstra os resultados obtidos com o RT² em termos de performance e uso de memória. Por fim, o capítulo 5 apresenta algumas considerações finais, além de propor futuras melhorias para o RT².

2. Contexto

Fazer uma imagem gerada por computador parecer real para a visão humana é um desafio que vem sendo estudado desde 1968, ano em que Arthur Appel define os primeiros conceitos de *ray casting* [10]. Por limitações tanto de poder computacional como de dispositivos de visualização, ideias para síntese de imagens pseudo-realísticas como *ray tracing* [1] e *photon mapping* [2] ficaram por muito tempo limitadas ao uso no cinema e em *design* de produtos industriais. Entretanto, hoje em dia esta realidade é alterada aos poucos, fazendo com que tais técnicas sejam utilizadas em novos e diferentes produtos de mercado, como será mostrado na primeira seção deste capítulo. As seções restantes descrevem todo o contexto relacionado ao RT², como *ray tracing*, estruturas de aceleração e o *framework* de CUDA [9].

2.1. Rasterização – A base dos Jogos 3D

Em meados de 1993 o mercado de computadores domésticos passou a disponibilizar para venda as primeiras placas gráficas com suporte à aceleração 3D [5], sendo tais dispositivos hoje comumente chamados de GPUs – *Graphics Processing Units*. Com este ganho computacional, proveniente principalmente da arquitetura de microprocessadores paralelos e operações aritméticas otimizadas em ponto flutuante, tornou-se possível criar jogos 3D interativos [11], em que o jogador explora o ambiente virtual de uma forma mais rica e diversificada do que nos jogos 2D. Esta interatividade atraiu o público de tal maneira que em menos de meia década se tornou um requisito a presença de uma placa gráfica com suporte à aceleração 3D em um computador doméstico. Com esta forte demanda, o mercado de jogos cresce até os dias de hoje em conjunto com a evolução das GPUs.

Além de jogos, aplicativos voltados para diversas áreas profissionais também passaram a tirar proveito das GPUs. Programas de modelagem 3D [12] são hoje em dia muito utilizados em produções cinematográficas, e também como ferramentas de criação de jogos. Programas CAD (*Computer-Aided Design*) [13] são muito utilizados na criação de desenhos técnicos, sendo parte fundamental de projetos em diversas áreas, como Engenharia Civil, Geologia, Arquitetura e *Design*.

Para tornar possível a simulação 3D, tais GPUs oferecem uma implementação otimizada em *hardware* de um *pipeline* de renderização por rasterização [2]. As técnicas de rasterização, de maneira resumida, se baseiam principalmente nos seguintes procedimentos: 1) transformar todos os objetos da cena 3D em polígonos; 2) ordená-los espacialmente em relação ao espaço de câmera; e 3) finalmente, projetá-los no plano projetivo, definindo a imagem final. Para renderizar uma esfera, por exemplo, a mesma é transformada em/representada por um conjunto de polígonos, que posteriormente torna-se um grupo de triângulos e, por fim, dependendo de sua projeção no plano projetivo, tais triângulos são exibidos ou não na tela. A tela é representada por uma matriz de *pixels*, chamada de *raster* (origem do termo *rasterization*), mas comumente conhecida como *bitmap*.

Apesar da evolução e sofisticação do *pipeline* de rasterização, ele é muito suscetível a problemas de representação, gerando artefatos visuais que incomodam a percepção visual humana, como o serrilhado ou *aliasing* [2]. Isto se deve ao fato destas técnicas serem baseadas em “desenhar” polígonos na tela, passando por um processo de discretização relativamente simples da representação contínua de tais polígonos. Além disso, alguns efeitos visuais como reflexão em objetos metálicos são obtidos em rasterização através de custosas e imprecisas técnicas adicionais, como o *environment mapping* [2]. Entretanto, vale lembrar que apesar destes problemas, as técnicas baseadas em rasterização são bem estabelecidas e otimizadas em *hardware*, de maneira a oferecer uma excelente performance para

renderização em tempo real. Estas técnicas são hoje a base para toda a área de Computação Gráfica voltada para jogos 3D.

2.2. Ray Tracing

Técnicas de iluminação global usufruem dos conceitos básicos da Física ótica [2] para renderizar cenas fotorealísticas, ou seja, com um aspecto mais real do que as rasterizadas. Seguindo este conceito, ao invés de “desenhar” triângulos na tela, a técnica de traçado de raios ou *ray tracing* [1], por exemplo, se baseia na emissão de raios luminosos por fontes de luz da cena. Tais raios se propagam no ambiente, sendo refletidos/refratados por objetos na cena. Mesmo utilizando os conceitos mais básicos da Física ótica, o *ray tracing* oferece uma imagem final com uma riqueza de detalhes bem maior, se comparado com cenas rasterizadas, como é demonstrado na Figura 1. Materiais transparentes ou refletivos são representados com maior realismo, mesmo no mais simples *ray tracer*, do que utilizando as mais modernas técnicas de rasterização.



Figura 1. Níveis de realismo diferentes de uma cena renderizada com as técnicas de rasterização (esquerda) e *ray tracing* (direita).

Ao contrário da rasterização, *ray tracing* não depende da transformação dos objetos de toda a cena em polígonos. Tais objetos podem ser representados de

forma mais precisa. A esfera, por exemplo, pode ser considerada como uma entidade algébrica que, a partir de sua equação espacial é possível obter informações suficientes para a renderização, realizando assim menos etapas de conversão contínuo-discreto e gerando menos erros. Esta forma diferenciada de representar os objetos é uma das grandes vantagens do *ray tracing*.

Para gerar imagens de qualidade gráfica superior, a técnica de *ray tracing* demanda um alto poder computacional se comparada com a rasterização, sendo geralmente utilizada apenas para renderizações *off-line* com um foco na qualidade da imagem. Entretanto, com o crescimento da performance tanto das CPUs como das GPUs, nos últimos anos começaram a surgir trabalhos utilizando *ray tracing* também para renderizações em tempo real. Alguns pesquisadores inclusive afirmam que o *ray tracing* oferece uma melhor performance que a rasterização em cenas de alta complexidade [2], com mais de 100 milhões de polígonos visíveis, onde o algoritmo de rasterização se tornaria proibitivo. O principal argumento se refere à complexidade assintótica do *ray tracing*, que cresce de maneira sub-linear em relação ao número de objetos na cena, enquanto que na rasterização esse crescimento é linear.

Todo *ray tracer* inicia seu processamento emitindo no espaço 3D um conjunto de raios que começam a partir da posição da câmera virtual, compartilhando assim o mesmo ponto de origem. Além disto, tais raios são direcionados para cortar o chamado “*screen space*”, que neste caso representará a imagem final, como é demonstrado na Figura 2. Os raios, por serem o ponto de entrada da renderização do *ray tracing*, são chamados de raios primários. Em um *ray tracer* simples, é emitido um raio primário para cada *pixel* da imagem. O processo de emissão destes raios primários é chamado de *ray casting*.

RT² - Real Time Ray Tracer

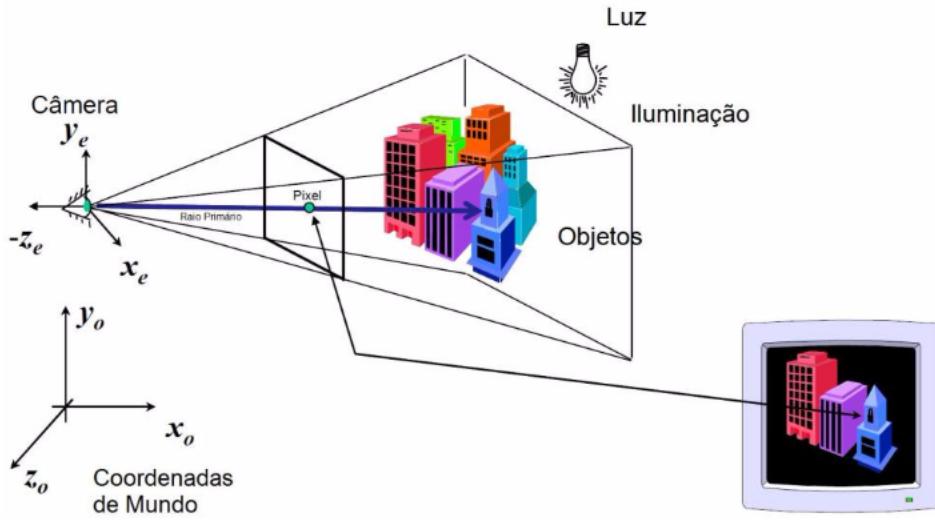


Figura 2. Processo de síntese de imagens por *ray tracing*.

Ao interseccar com um objeto na cena, novos raios poderão ser emitidos, sendo estes raios refletidos, transmitidos, de sombra ou de iluminação. Tais raios são conhecidos como raios secundários. Os raios refletidos são gerados a partir da lei de reflexão [1], enquanto que os transmitidos obedecem às regras de refração de Snell [1]. Estas regras são demonstradas na Figura 3.

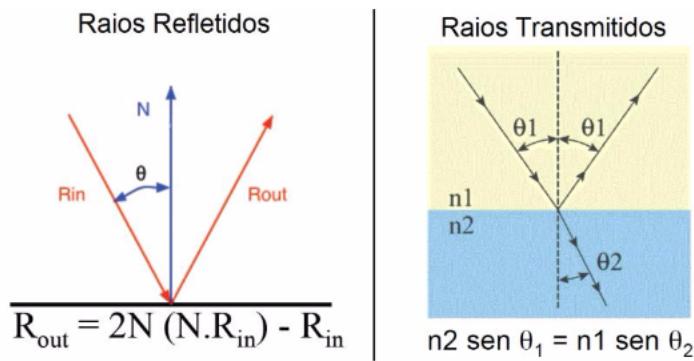


Figura 3. Cálculo do raio refletido e do raio transmitido (refratado).

É importante observar que estes casos de reflexão e transmissão podem ocorrer fora do campo de visão da câmera. Um exemplo disto é o caso de termos um espelho em frente à câmera, que irá exibir objetos atrás do olho (câmera)

virtual. Neste caso, técnicas de *frustum culling* [2], utilizadas em rasterização, necessitam de algumas modificações para funcionar corretamente em *ray tracing*. Isto significa que, ao invés de uma das faces do *frustum* representar o “*screen space*”, temos que estender o *frustum* para regiões que seguem o eixo negativo do vetor de direção da câmera. Assim, objetos fora do campo de visão poderão ser refletidos em objetos visíveis e reflexivos. Uma abordagem interessante, neste caso, é de, ao invés de definir um *frustum*, utilizar uma esfera ou um elipsóide para realizar a operação de *culling*. Isto pode melhorar a performance, já que testes de intersecção entre a esfera e outras primitivas são um dos testes mais baratos computacionalmente.

Cada raio gerado contribui com uma parcela da cor final do *pixel*. Esta contribuição depende principalmente dos materiais (texturas/cores) dos objetos intersectados e dos ângulos de incidência sobre tais objetos. Além disso, depende do modelo de iluminação local em cada ponto de intersecção raio-objeto. Os modelos mais utilizados são os de *Phong* e *Blinn Shading* [8], pois oferecem um bom resultado visual com relativo baixo custo computacional. Observe que, diferentemente do que ocorre em rasterização, no *ray tracing* o processo de *shading* pode ocorrer mais de uma vez por *pixel* para o cálculo final da cor, sendo esta quantidade de ocorrências definida pelo número de raios de iluminação que este *pixel* irá gerar. Entretanto, esta etapa envolve cálculos relativamente simples se comparado com todo o *pipeline* do *ray tracing*, não tendo assim um grande impacto na performance.

2.3. Estruturas de Aceleração

O alto custo computacional de um *ray tracer* se deve ao problema de, para cada raio \mathbf{r} emitido na cena, encontrar a primitiva geométrica \mathbf{p} intersectada por \mathbf{r} mais próxima e posterior à origem de \mathbf{r} . Uma forma simples de resolver tal

problema é a realização de testes de intersecção entre r e todas as primitivas da cena [1], como demonstra o Algoritmo 1.

Algoritmo 1. Busca simples da intersecção mais próxima entre raio e objeto.

```
Para cada primitiva geométrica  $p$  da cena{  
    Se há intersecção entre  $p$  e  $r$ {  
        Se é a intersecção mais próxima encontrada até o momento{  
            Guarde  $p$  como a primitiva de retorno;  
            Guarde a distância paramétrica de  $r$  até  $p$ .  
        }  
    }  
}
```

No entanto, por percorrer todos os objetos, o uso deste algoritmo torna a complexidade assintótica da busca em linear [8]. Como o custo da intersecção raio-objeto é relativamente alto na maioria das arquiteturas e tipos primitivos, envolvendo dezenas de operações aritméticas em ponto flutuante, tal abordagem se torna proibitiva em *ray tracing* de tempo real para cenas com milhões de primitivas geométricas. Utilizando esta abordagem, mesmo com algumas centenas de primitivas, cerca de 95% do tempo de execução do *ray tracing* se deve ao testes de intersecção raio-objeto [3].

Este problema é amenizado através do uso de estruturas de dados que organizam espacialmente ou agrupam os objetos da cena, de forma a reduzir consideravelmente o número de testes de intersecção raio-objeto. Deste modo, ao invés de seguir uma abordagem de busca “cega” compreendendo todos os objetos, apenas aqueles com alguma probabilidade de intersecção com o raio são testados. Com isto, são descartados da busca os objetos garantidos de não intersectar, como objetos distantes ou anteriores à origem em relação à direção do raio, sendo geralmente tais objetos a maior parte do conjunto de cada busca.

A maioria destas estruturas são adaptações de árvores de busca balanceadas [2]. Elas reduzem a complexidade assintótica da busca para sub-linear [8] e, por aumentarem consideravelmente a performance de um *ray tracer*, são chamadas de estruturas de aceleração.

As estruturas de aceleração podem ser obtidas através do conceito de subdivisão espacial da cena (*Partitioning Space*), por subdivisão em conjuntos de objetos (*Partitioning Object List*) ou utilizando uma abordagem híbrida, tirando proveito tanto do conceito de partição do espaço como de agrupamento de objetos [21].

Apesar da grande diversidade de estruturas de aceleração propostas na literatura, o tipo mais utilizado em *ray tracing* se refere à *Binary Space Partitioning Tree (BSP-Tree)* [2]. Tal árvore sub-divide recursivamente a cena a cada ramo, de forma a separar em sub-grupos espaciais todos os objetos da cena. Assim, um *ray tracer* com *BSP-Tree* tira proveito de algoritmos de busca que seguem a ideia de percorrer o raio em seu sentido até encontrar o primeiro grupo de objetos que o intersecta, ou até sair dos limites da cena sem intersectar com nenhuma primitiva.

A subdivisão recursiva ocorre a partir da escolha de um plano de corte (*splitting plane position*) do espaço em todas as ramificações da árvore. Entretanto, no contexto de *ray tracing* em tempo real, visando reduzir custos computacionais de construção da estrutura e simplificar o algoritmo de travessia, a escolha do plano é restrita a um dos eixos coordenados da cena, sendo esta BSP chamada de *kD-Tree (k-Dimensions Tree)* [17]. Em uma *kD-Tree*, todo nó interno armazena a dimensão e posição do plano de corte que irá dividir a cena em dois sub-espacos, sendo os filhos do nó os representantes desses sub-espacos. Assim, as folhas representam os menores sub-espacos da árvore e armazenam uma lista dos objetos contidos em tais espaços.

Como os planos de corte de uma *kD-Tree* têm suas normais como vetores canônicos das coordenadas de mundo, todo sub-espaco da cena é representado por

caixas de eixos alinhados ou *Axis-Aligned Bounding Boxes* (AABBs). Neste caso, o nó raiz da árvore representa o “*bounding box*” da cena. Deste modo, todo algoritmo de travessia da árvore se reduz a testes de intersecção raio-AABB até encontrar uma intersecção com algum objeto contido no AABB ou o raio sair do *bounding box* da cena, sem encontrar nenhum objeto de intersecção. Para cada nó interno, como apenas um dos eixos é escolhido como suporte do plano de corte, o teste de intersecção raio-AABB é novamente simplificado apenas para a dimensão do plano de corte, reduzindo consideravelmente o número de operações aritméticas por nó visitado.

A Figura 4 demonstra o *Stanford Dragon* [14] estruturado em uma *kD-Tree*, onde as caixas verdes representam os AABBs das folhas. Observe que, neste caso, toda a região esverdeada representa o AABB do dragão, que por sua vez representa o nó raiz da árvore.

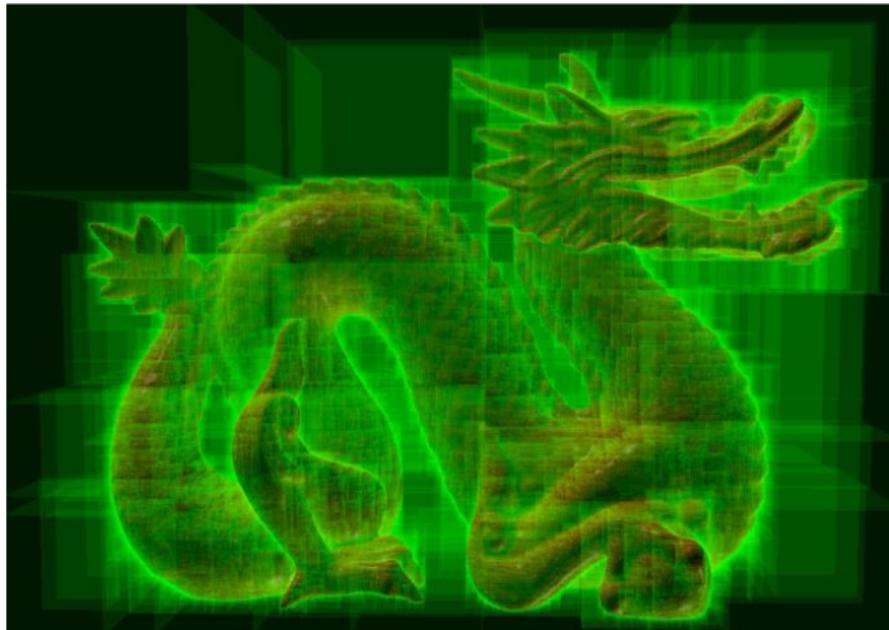


Figura 4. Representação da *kD-Tree* (regiões esverdeadas) sobre o modelo do *Stanford Dragon*.

Ao interceptar um nó interior, um raio pode atravessar apenas um dos dois filhos ou ambos, como demonstra a Figura 5. Para o último caso, no algoritmo de travessia padrão [15] é preciso armazenar em uma pilha o nó-filho mais distante (*far child*) para uma busca posterior, já que a travessia segue recursivamente o primeiro nó no ramo do filho mais próximo (*near child*), para depois seguir para o segundo. A pilha garante então que a travessia ocorra na ordem em que os *bounding boxes* dos nós são intersectados pelo raio, visitando apenas uma vez cada nó. O pseudo-código da travessia padrão é demonstrado no Algoritmo 2.

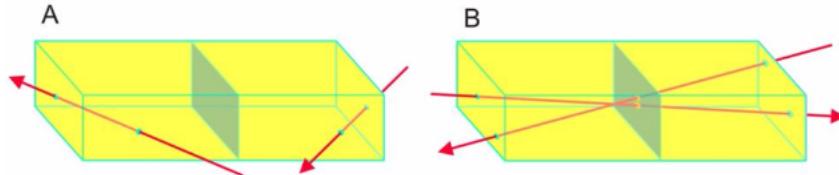


Figura 5. Casos de intersecção raio-AABB do nó. Caso A: Intersecta apenas um dos filhos. Caso B: Intersecta ambos os filhos.

Algoritmo 2. Travessia padrão em *kD-Tree*.

```

root → nó raiz da kD-Tree
stack → pilha que armazena os nós
sceneMin,sceneMax → distância paramétrica da origem do raio até o ponto de
    entrada e de saída do bounding box da cena
tHit → distância do raio ao objeto intersectado mais próximo
intersected → objeto mais próximo encontrado

stack.push(root,sceneMin,sceneMax);
tHit=infinity;
intersected = null;

while (!stack.empty()): //enquanto existir nós para visitar...

    (node,tMin,tMax)=stack.pop();
    while (!node.isLeaf()): //procura pelo próximo nó folha a ser visitado

        axis = node.axis; // eixo de corte do nó interior
        // ponto de corte em valor parametrizado em relação ao raio

```

```
tSplit = ( node.splitPosition - ray.origin[axis] ) / ray.direction[axis];
// define o primeiro e o segundo nó cortado, caso o raio passe pelos dois nós
(near, far) = order(ray.direction[axis], node.left, node.right);
if( tSplit >= tMax or tSplit < 0):
    node=near; // O raio passa somente pelo primeiro nó
else if( tSplit < tMin):
    node=far; // O raio passa somente pelo segundo nó
else:
    // guarda o nó mais distante para processar depois
    stack.push( far, tSplit, tMax );
    node=near; // escolhe o nó mais próximo para processar
    tMax=tSplit;

//Alcançou um nó-folha. Checar se intersecta com algum objeto do nó
for object in node.objects():
    (tHit,intersected)=min(tHit, object.Intersect(ray));
if tHit<tMax:
    //encontrou um objeto que intersecta, já é possível então realizar early exit
    return (tHit, intersected);

return null; //raio intersectou nenhum objeto da cena
```

2.4. Ray Tracing em GPU

Durante os últimos anos, o rápido crescimento do poder computacional dos processadores gráficos atraiu o interesse de pesquisadores para a busca por implementações eficientes de algoritmos de *ray tracing* em GPU, principalmente algoritmos de busca em estruturas de aceleração como as *kD-Trees*. Foley *et al.* [16] adaptaram o algoritmo de travessia padrão (descrito na seção anterior), criando dois novos algoritmos de busca que não necessitam de uma pilha para a travessia, reduzindo consequentemente o número de acessos à memória primária da GPU.

O primeiro algoritmo, chamado *kd-restart*, reinicia a busca para a raiz da árvore toda vez que atinge uma folha da *kD-Tree*. Esta operação de reinicialização (*restart*) ocorre até que se encontre uma intersecção com alguma primitiva ou o raio saia do *bounding box* da cena. Ao reiniciar para a raiz, não é mais necessário

armazenar os casos de intersecção com ambos os filhos de um nó interior, removendo assim a pilha presente no algoritmo padrão. Entretanto, o custo médio da busca se torna consideravelmente maior que o da travessia padrão, pois ao reiniciar para a raiz, muitos nós serão revisitados. O pseudo-código do *kd-restart* é demonstrado no Algoritmo 3.

Algoritmo 3. Travessia utilizando a técnica de *kd-restart*.

```

root → nó raiz da kD-Tree
stack → pilha que armazena os nós
sceneMin,sceneMax → distância paramétrica da origem do raio até o ponto de
    entrada e de saída do bounding box da cena
tHit → distância do raio ao objeto intersectado mais próximo
intersected → objeto mais próximo encontrado

tMin=tMax=sceneMin;
tHit=infinity;
intersected = null;

//enquanto a travessia não cruzar o sceneMax...
while (tMax<sceneMax):
    node=root; // ocorrencia do evento de restart
    tMin=tMax;
    tMax=sceneMax;

    //procura pelo próximo nó folha a ser visitado
    while (!node.isLeaf()):
        axis = node.axis; // eixo de corte do nó interior

        // ponto de corte em valor parametrizado em relação ao raio
        tSplit = ( node.splitPosition - ray.origin[axis] ) / ray.direction[axis];

        // define o primeiro e o segundo nó cortado, caso o raio passe pelos dois nós
        (near, far) = order(ray.direction[axis], node.left, node.right);
        if( tSplit >= tMax or tSplit < 0):
            node=near; // O raio passa somente pelo primeiro nó
        else if( tSplit < tMin):
            node=far; // O raio passa somente pelo segundo nó
        else:
            node=near; // escolhe o nó mais próximo para processar

```

RT² - Real Time Ray Tracer

```
tMax=tSplit;  
  
//Alcançou um nó-folha. Checar se intersecta com algum objeto do nó  
for object in node.objects():  
    (tHit,intersected)=min(tHit, object.Intersect(ray));  
    if tHit<tMax:  
        //encontrou um objeto que intersecta, já é possível então realizar early exit  
        return (tHit, intersected);  
  
return null; //raio intersectou nenhum objeto da cena
```

Para reduzir a quantidade de nós revisitados, o algoritmo *kd-backtrack* (segundo algoritmo) armazena em cada nó o seu AABB e um ponteiro para o pai. Desta forma, não se torna mais necessário retornar à raiz ao chegar numa folha e sim voltar para um nó que possibilite a busca em outros ramos ainda não visitados. Apesar de reduzir o número de nós visitados, o *kd-backtrack* demanda cerca de uma ordem de magnitude de memória a mais que o *kd-restart* para armazenar a árvore, por causa do requisito de armazenamento por nó do ponteiro do pai e dos AABBs nos nós internos. Esta característica torna este algoritmo inviável para cenas com mais de 10 milhões de polígonos, por exemplo.

Buscando manter a performance do algoritmo de travessia padrão, Horn *et al.* [17] propuseram algumas modificações no *kd-restart* para reduzir o número total de nós visitados, com os algoritmos *push-down* e *short-stack*. A técnica *push-down* mapeia o último nó interior **n** visitado que pode ser considerado como raiz da busca, sendo este um nó que o raio atravessa apenas um dos dois filhos. Desta forma, ao ocorrer um evento de *restart*, ao invés de voltar para a raiz, a busca começa em **n**, reduzindo o número de nós repetidamente visitados. O *push-down* é demonstrado no Algoritmo 4.

Algoritmo 4. Travessia utilizando a técnica de *push-down*.

```

root → nó raiz da kD-Tree ou subÁrvore do pushDown
stack → pilha que armazena os nós
sceneMin,sceneMax → distância paramétrica da origem do raio até o ponto de
    entrada e de saída do bounding box da cena
tHit → distância do raio ao objeto intersectado mais próximo
intersected → objeto mais próximo encontrado
pushDown → booleano que informa se é para realizar ou não o evento de pushDown

tMin=tMax=sceneMin; tHit=infinity; intersected = null;
//enquanto a travessia não cruzar o sceneMax...
while (tMax<sceneMax):
    node=root; //ocorrência do evento de restart
    tMin=tMax;
    tMax=sceneMax;
    pushDown = true;
    //procura pelo próximo nó folha a ser visitado
    while (!node.isLeaf()):
        axis = node.axis; //eixo de corte do nó interior
        //ponto de corte em valor parametrizado em relação ao raio
        tSplit = ( node.splitPosition - ray.origin[axis] ) / ray.direction[axis];
        //define o primeiro e o segundo nó cortado, caso o raio passe pelos dois nós
        (near, far) = order(ray.direction[axis], node.left, node.right);
        if( tSplit >= tMax or tSplit < 0):
            node=near; //O raio passa somente pelo primeiro nó
        else if( tSplit < tMin):
            node=far; //O raio passa somente pelo segundo nó
        else:
            node=near; //escolhe o nó mais próximo para processar
            tMax=tSplit;
            //como é preciso ir ao segundo nó, pushdown é bloqueado
            pushDown = false;
        if pushDown:
            root = node;

//Alcançou um nó-folha. Checar se intersecta com algum objeto do nó
for object in node.objects():
    (tHit,intersected)=min(tHit, object.Intersect(ray));
if tHit<tMax:
    //encontrou um objeto que intersecta, já é possível então realizar early exit
    return (tHit, intersected);
return null; //raio intersectou nenhum objeto da cena

```

Além do *push-down*, Horn *et al.* [17] propuseram o algoritmo *short-stack*, como mostra o Algoritmo 5. Esta técnica adiciona uma pilha do tipo circular e com tamanho em torno de 10% da pilha utilizada no algoritmo de travessia padrão. Assim, mesmo GPUs com pouca memória são capazes de executar uma travessia com pilha. Esta “*short stack*”, ou pilha pequena é utilizada na busca da mesma forma que no algoritmo de travessia padrão. Entretanto, se a pilha esvaziar e a busca não atingir o critério de término (raio sair do *bounding box* da cena), o evento de *restart* é executado da mesma forma que no *kd-restart*, retornando a busca para a raiz.

Algoritmo 5. Travessia utilizando a técnica de *short-stack*.

```

root → nó raiz da kD-Tree
stack → pilha que armazena os nós
sceneMin,sceneMax → distância paramétrica da origem do raio até o ponto de
    entrada e de saída do bounding box da cena
tHit → distância do raio ao objeto intersectado mais próximo
intersected → objeto mais próximo encontrado

tMin=tMax=sceneMin;
tHit=infinity;
intersected = null;

//enquanto a travessia não cruzar o sceneMax...
while (tMax<sceneMax):

    //Se pilha estiver vazia, é lançado um evento de restart
    if stack.empty():
        node=root; //ocorrência do evento de restart
        tMin=tMax;
        tMax=sceneMax;
    else:
        (node,tMin,tMax)=stack.pop();

    //procura pelo próximo nó folha a ser visitado
    while (!node.isLeaf()):
        axis = node.axis; // eixo de corte do nó interior

```

```
// ponto de corte em valor parametrizado em relação ao raio
tSplit = ( node.splitPosition - ray.origin[axis] ) / ray.direction[axis];

// define o primeiro e o segundo nó cortado, caso o raio passe pelos dois nós
(near, far) = order(ray.direction[axis], node.left, node.right);
if( tSplit >= tMax or tSplit < 0):
    node=near; // O raio passa somente pelo primeiro nó
else if( tSplit < tMin):
    node=far; // O raio passa somente pelo segundo nó
else:
    stack.push(sec,tSplit,tMax)
    node=near; // escolhe o nó mais próximo para processar
    tMax=tSplit;

// Alcançou um nó-folha. Checar se intersecta com algum objeto do nó
for object in node.objects():
    (tHit,intersected)=min(tHit, object.Intersect(ray));
if tHit<tMax:
    //encontrou um objeto que intersecta, já é possível então realizar early exit
    return (tHit, intersected);

return null; //raio intersectou nenhum objeto da cena
```

Pelo fato de *push-down* e *short-stack* serem algoritmos que otimizam situações diferentes do *kd-restart*, ambos podem ser utilizados de maneira complementar. Neste caso, se ocorrer um evento de *restart* no *short-stack*, ao invés de retornar para a raiz, a busca volta para o nó interior que o algoritmo de *push-down* armazenou. A utilização híbrida dessas duas técnicas é demonstrada no Algoritmo 6.

Algoritmo 6. Travessia híbrida utilizando *short-stack* e *push-down*.

root → nó raiz da kD-Tree
stack → pilha que armazena os nós
sceneMin, sceneMax → distância paramétrica da origem do raio até o ponto de entrada e de saída do bounding box da cena
tHit → distância do raio ao objeto intersectado mais próximo
intersected → objeto mais próximo encontrado

pushDown → booleano que informa se é para realizar ou não o evento de pushDown

```
tMin=tMax=sceneMin;  
tHit=infinity;  
intersected = null;  
  
//enquanto a travessia não cruzar o sceneMax...  
while (tMax<sceneMax):  
  
    //Se pilha estiver vazia, é lançado um evento de restart  
    if stack.empty():  
        node=root; //ocorrência do evento de restart  
        tMin=tMax;  
        tMax=sceneMax;  
        pushDown = true;  
    else:  
        (node,tMin,tMax)=stack.pop();  
        pushDown = false;  
  
    //procura pelo próximo nó folha a ser visitado  
    while (!node.isLeaf()):  
        axis = node.axis; // eixo de corte do nó interior  
  
        // ponto de corte em valor parametrizado em relação ao raio  
        tSplit = ( node.splitPosition - ray.origin[axis] ) / ray.direction[axis];  
  
        // define o primeiro e o segundo nó cortado, caso o raio passe pelos dois nós  
        (near, far) = order(ray.direction[axis], node.left, node.right);  
        if( tSplit >= tMax or tSplit < 0):  
            node=near; // O raio passa somente pelo primeiro nó  
        else if( tSplit < tMin):  
            node=far; // O raio passa somente pelo segundo nó  
        else:  
            stack.push(sec,tSplit,tMax)  
            node=near; // escolhe o nó mais próximo para processar  
            tMax=tSplit;  
            pushDown = false;  
  
        if pushDown:  
            root = node;  
  
    //Alcançou um nó-folha. Checar se intersecta com algum objeto do nó  
    for object in node.objects():  
        (tHit,intersected)=min(tHit, object.Intersect(ray));  
    if tHit<tMax:
```

```
//encontrou um objeto que intersecta, já é possível realizar early exit  
return (tHit, intersected);  
  
return null; //raio intersectou nenhum objeto da cena

---


```

Popov *et al.* [18] demonstraram uma implementação em CUDA de uma travessia sem uso de pilhas (*stackless*) utilizando o conceito de cordas ou *ropes* [25], em que cada folha armazena seu *bounding box* e seis ponteiros para os nós vizinhos de cada uma das faces do *bounding box* da folha. Desta forma, ao chegar numa folha, ao invés de retornar para um nó armazenado na pilha, é utilizado o ponteiro de uma das faces da folha. Assim, a travessia com *ropes* visita menos nós que todos os algoritmos citados anteriormente, mas com mais acessos à memória para leitura do *bounding box* da folha e dos *ropes*. O pseudo-código é demonstrado no Algoritmo 7.

Algoritmo 7. Travessia utilizando o conceito de *ropes*.

```
root → nó raiz da kD-Tree  
sceneMin, sceneMax → distância paramétrica da origem do raio até o ponto de  
    entrada e de saída do bounding box da cena  
tHit → distância do raio ao objeto intersectado mais próximo  
intersected → objeto mais próximo encontrado  
pEntry → ponto de entrada do raio no bounding box do nó-folha visitado  
  
tMin = sceneMin; tMax = sceneMin;  
tHit = infinity; intersected = null;  
  
//enquanto a travessia não cruzar o sceneMax...  
while (tMin < sceneMax):  
  
    pEntry = ray.origin + tMin * ray.direction;  
  
    //procura pelo próximo nó folha a ser visitado  
    while (!node.isLeaf()):  
        axis = node.axis; // eixo de corte do nó interior  
        if (pEntry[axis] <= node.splitPosition):  
            node = node.left;
```

```
else:  
    node = node.right;  
  
//exitFace indica qual das faces o raio sai, informando qual rope utilizar  
(tMax, exitFace) = ray.getExitIntersection(node.aabb);  
  
//Alcançou um nó-folha. Checar se intersecta com algum objeto do nó  
  
for object in node.objects():  
    (tHit,intersected)=min(tHit, object.Intersect(ray));  
if tHit<tMax:  
    //encontrou um objeto que intersecta, já é possível então realizar early exit  
    return (tHit, intersected);  
  
//utiliza o ropes para encontrar o nó vizinho à face de saída  
node = node.getRope(exitFace);  
if(node == null) // a face faz parte da fronteira do bounding box da cena  
    return null;  
  
return null; //raio intersectou nenhum objeto da cena
```

Uma melhoria deste algoritmo foi realizada neste Trabalho de Graduação, sendo descrita na seção 3.4 deste documento. A alteração realizada busca otimizar o conceito de travessia com *ropes* para a arquitetura GPGPU (descrita nas próximas seções), ao reduzir a demanda por registradores e o número de acessos à memória global, obtendo assim ganhos de performance através de um melhor uso da arquitetura de CUDA.

2.5. Programação em GPU

GPU é um tipo de microprocessador especializado no processamento de gráficos. Este dispositivo é encontrado em *videogames*, computadores pessoais e estações de trabalho, situado na placa de vídeo ou integrado diretamente à placa-mãe.

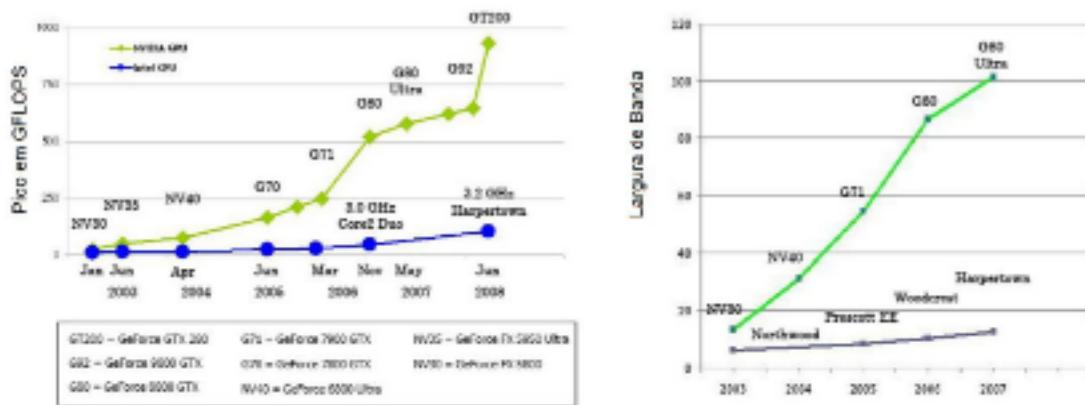


Figura 6. Evolução das GPUs e CPUs. Na figura à esquerda, gráfico demonstrativo em relação à performance, enquanto que na direita, em relação à largura de memória. Fonte: NVIDIA [5].

As recentes arquiteturas das GPUs proporcionam, além de um vasto poder computacional, demonstrado no gráfico da Figura 6, uma grande velocidade no barramento de dados, sendo superiores às CPUs comuns nestes aspectos. Apenas para exemplificar, podemos comparar o Intel Core i7 Quad Core 3.2GHz, que possui picos de 51.2 GFLOPS e 64 GB/s em seu desempenho, enquanto que a placa de vídeo GeForce GTX 295 possui picos de 1788 GFLOPS e 223.8 GB/s. É importante lembrar que estes desempenhos foram obtidos em situações ideais. Existem casos em que a performance de uma CPU pode ser superior à de uma GPU. Isto está relacionado principalmente ao grau de paralelismo que a aplicação oferece e seu modelo de acessos à memória.

Originalmente, como o próprio nome sugere, uma GPU foi concebida para realizar processamento gráfico, incluindo rasterização, e outras tarefas de sintetização de imagens. Este processo é possível através de uma sequência de estágios bem definidos chamada de *pipeline* gráfico. Cada estágio é responsável por uma tarefa como: transformação de coordenadas; cálculos de câmera e iluminação.

GPUs são processadores orientados para a execução paralela de instruções, sendo otimizados para processar operações sobre vetores no modo SIMD (*Simple Instruction, Multiple Data*). Sua arquitetura foi projetada de modo a incluir mais transistores dedicados ao processamento de dados, em detrimento da realização do *caching* de dados e do fluxo de controle, quando comparado a uma CPU comum, como ilustrado na Figura 7.



Figura 7. Arquitetura de uma CPU vs de uma GPU. Fonte: NVIDIA [5].

Desta forma, uma GPU consegue lidar melhor com problemas que demandam uma grande quantidade de operações sobre dados. Já que uma mesma operação é realizada sobre cada elemento dos dados, não existe a necessidade de um fluxo de controle sofisticado. Estas características permitem que latências no acesso à memória sejam disfarçadas através da grande vazão de cálculos. Como mencionado anteriormente, a arquitetura de uma GPU é bastante relacionada ao *pipeline* gráfico. Sua arquitetura foi projetada de modo a realizar as operações contidas no *pipeline* de forma simultânea, como ilustra a Figura 8.

RT² - Real Time Ray Tracer

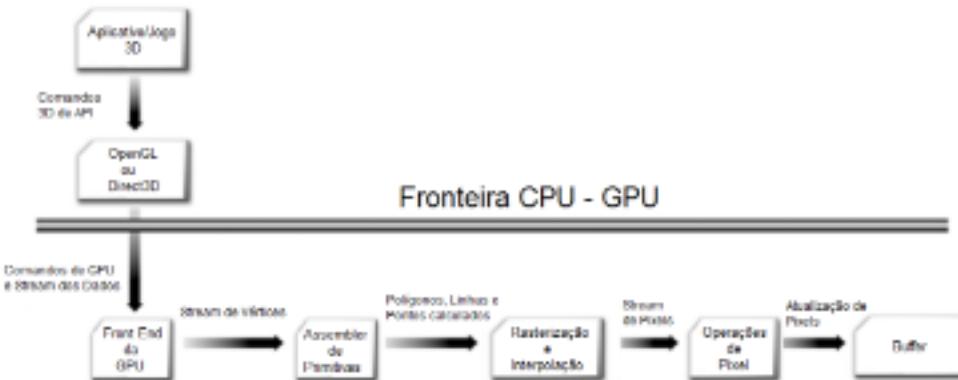


Figura 8. Pipeline gráfico 3D utilizado em rasterização. Fonte: NVIDIA[5].

Posteriormente, por volta do ano 2000, novas funcionalidades foram adicionadas às GPUs, envolvendo a habilidade de modificar o processo de renderização do *pipeline*, através dos chamados *shaders* [2]. Tais *shaders* possibilitaram a adição de efeitos gráficos customizados – como *phong shading* e *bump mapping* – e difíceis de programar utilizando apenas o antigo *pipeline* gráfico. A Figura 9 demonstra a arquitetura da imagem anterior, entretanto com a adição do *pipeline* programável.

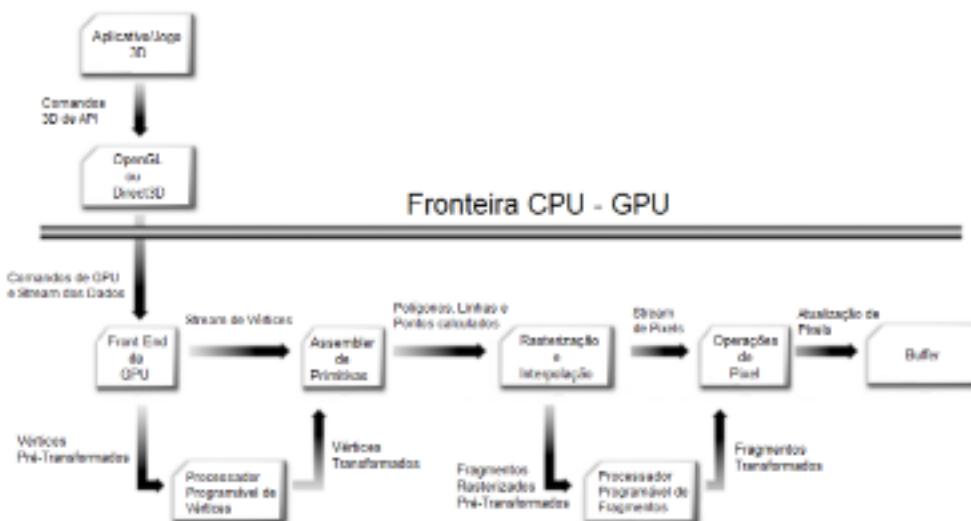


Figura 9. Pipeline gráfico 3D rasterizado com a adição de componentes programáveis. Fonte: NVIDIA [5].

Recentemente, surgiu outra forma de programar sobre a arquitetura de uma placa de vídeo. Chamado de GPGPU, este modelo propõe o uso de uma placa de vídeo não apenas para a implementação de aplicativos gráficos, ampliando a gama para programação de propósito geral abordando praticamente todas as áreas da computação. Algoritmos de segurança e de busca, por exemplo, caso ofereçam um bom grau de paralelismo, podem ser implementados em GPGPU, tirando proveito do poder computacional paralelo da GPU.

2.6. CUDA Framework

CUDA [9] se refere a um *framework* para GPGPU nas placas gráficas da NVIDIA, tendo sido lançado em 2007. Os primeiros processadores compatíveis com CUDA são os processadores G80 da NVIDIA. Tal *framework* é composto por três principais componentes de *software*:

- CUDA SDK – fornece exemplos de código e bibliotecas necessárias para a compilação de código escrito para CUDA. Além disso, o SDK dispõe de toda a documentação necessária para a programação em CUDA.
- CUDA Toolkit – contém o compilador e bibliotecas adicionais como o CUBLAS (*CUDA port of Basic Linear Algebra Subprograms*) e o CUFFT (*CUDA implementation of Fast Fourier Transform*). Além disso, o Toolkit contém o CUDA Profiler, ferramenta para análise de desempenho e uso de memória de funções escritas para CUDA.
- CUDA Driver – este componente vem instalado em conjunto com o próprio *driver* da placa de vídeo, necessário para o sistema operacional. Realiza então toda a comunicação em baixo nível entre o sistema operacional e a GPU.

O código CUDA de uma aplicação é armazenado em arquivos com a extensão “.cu”. Nestes arquivos são permitidos códigos similares a linguagem C, com algumas extensões, que são necessárias para lidar com detalhes relacionados à forma como uma GPU opera. Porém, a sintaxe básica de C e algumas partes de C++ também é permitida. Assim, aplicações CUDA podem ser facilmente integradas a aplicações já existentes, já que funções que sigam a sintaxe padrão podem ser invocadas por outras aplicações.

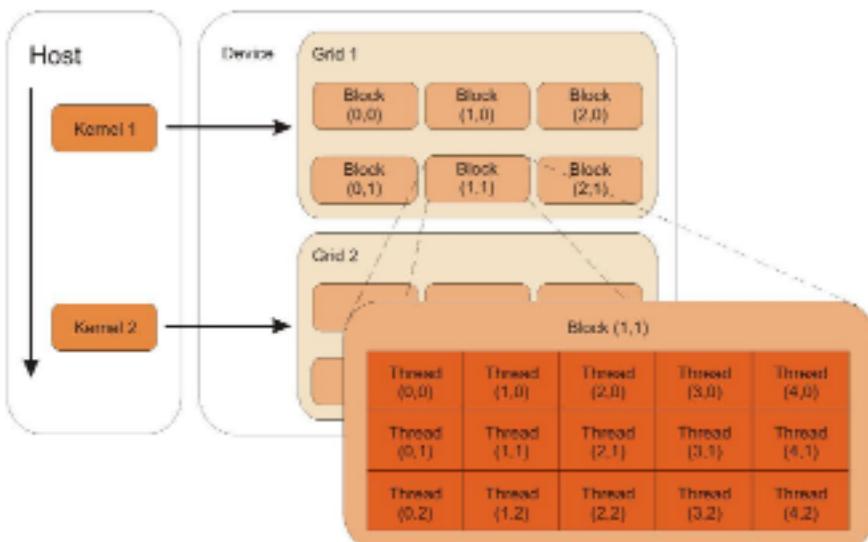


Figura 10. Organização de *threads*, blocos e *grids* em CUDA. Fonte: NVIDIA [5].

Alguns conceitos importantes relacionados à CUDA são *threads*, *blocks* (blocos), *grids* (grade) e *kernels* (núcleos), como mostra a Figura 10. *Threads* são as unidades de execução paralela em uma GPU. As *threads* são agrupadas em *blocks* ou bloco, onde *threads* pertencentes a um mesmo bloco podem sincronizar sua execução e compartilhar um mesmo espaço de memória (*shared memory*). Um conjunto de blocos representa um *grid*. Um núcleo ou *kernel* consiste no código que é executado por uma *thread*. Cada chamada a um *kernel* precisa especificar uma configuração contendo o número de blocos em cada *grid*, o número de *threads* em

cada bloco e, opcionalmente, a quantidade de memória compartilhada (*shared memory*) a ser alocada.

Em CUDA, algumas palavras chaves são adicionadas à linguagem C. O conjunto dessas palavras chaves representa o escopo de uma função ou variável: `_host_`, `_device_` e `_global_`. O `_global_` é utilizado apenas para as declarações de *kernels*, porque o *kernel* é uma função que reside na GPU, mas é chamada a partir da CPU. Os modificadores `_host_` e `_device_` são aplicados tanto a funções como a variáveis, para especificar sua localização, e de que ponto podem ser chamadas (CPU ou GPU). Outra modificação na linguagem é a invocação dos *kernels*; a chamada de um *kernel* é seguida pela sua configuração entre a sequência de caracteres “`<<<`” e “`>>>`”, como em:

```
kernel_name<<<gridDim, blockDim>>>(params);
```

A palavra-chave `_shared_` foi introduzida para indicar que uma variável será alocada no espaço da memória compartilhada, e estará acessível apenas a *threads* de um mesmo *block*.

Assim, o modelo de *software* proposto por CUDA visualiza a GPU como um co-processador de dados paralelo. Neste contexto, a GPU é considerada como o dispositivo/*device* e a CPU como o “anfitrião” (ou *host*).

A plataforma oferece três bibliotecas que podem ser utilizadas no desenvolvimento das aplicações. O *Common Runtime Library*, que provê alguns tipos de dados como vetores com duas, três ou quatro dimensões, e algumas funções matemáticas. O *Host Runtime Library*, que provê gerenciamento dos dispositivos e da memória, incluindo funções para alocar e liberar memória, e funções para invocar *kernels*. E o *Device Runtime Library*, que pode apenas ser utilizado pelos dispositivos, e que provê funções específicas como sincronização de *threads*.

3. RT² - Real Time Ray Tracer

Todo aplicativo 3D interativo depende de uma API (*Application Programming Interface*) gráfica para renderizar a sua cena. Quando se utiliza um *pipeline* gráfico em *hardware*, como hoje ocorre com o uso das GPUs, é preciso de uma ponte de comunicação entre a camada na qual o programador escreve seu aplicativo e o que a biblioteca gráfica executa. Esta divisão traz a vantagem de desacoplar a camada de aplicação de todo o *pipeline* gráfico, deixando o aplicativo do usuário independente da tecnologia utilizada para renderizar. Assim, com o surgimento de futuras tecnologias de renderização, o programador terá pouca ou nenhuma dificuldade em atualizar seu aplicativo para funcionar corretamente em novos dispositivos. Seguindo este conceito, foi planejado e desenvolvido o RT², biblioteca que oferece uma API gráfica abstrata para a renderização de *ray tracing* em tempo real, tirando proveito tanto de uma CPU como de uma ou mais GPUs.

Além da capacidade de evoluir com o surgimento de novos tipos de processadores, por trocar o *pipeline* de rasterização em *hardware* por um *ray tracer* em *software*, o RT² é também capaz de evoluir em termos algorítmicos com maior grau de liberdade do que em APIs gráficas comuns, como OpenGL [26] e Direct3D [27]. Estas bibliotecas são fortemente dependentes de uma implementação em *hardware*, sendo passíveis de melhorias apenas quando novas placas de vídeo com circuitos atualizados para as novas funcionalidades surgem, o que nos dias de hoje ocorre em um ciclo de seis meses a um ano [5]. Uma biblioteca como o RT², por ter sua base em *software*, pode ser atualizada a qualquer momento. Mais importante que a freqüência de atualização, é o fato de que novas funcionalidades poderão ser utilizadas em equipamentos antigos e compatíveis com o RT², o que não ocorre em uma API diretamente dependente de uma única e exclusiva arquitetura de *hardware*.

É importante observar que, apesar do RT² ser capaz de executar *ray tracing* em tempo real (como será demonstrado nos resultados descritos no capítulo 4) e em resoluções de alta definição, o mesmo não é capaz de superar a performance de uma implementação de rasterização otimizada em *hardware*. Entretanto, o verdadeiro ganho que o RT² oferece está na troca de desempenho por maior qualidade gráfica. Por seu núcleo se basear em *ray tracing*, o RT² oferece nativamente alguns efeitos gráficos de alta qualidade que são conseguidos em rasterização apenas através de *shaders* ou utilizando um custoso *pipeline* de múltiplos estágios. Para exemplificar, seguem abaixo algumas características presentes no RT²:

- Compatível com o uso de entidades algébricas como esferas, elipses, caixas e planos, além de triângulos. Em rasterização, a única primitiva permitida no *pipeline* são triângulos, fazendo com que toda entidade algébrica seja transformada em polígonos, reduzindo a qualidade visual da imagem. O volume de informação para representar e renderizar o modelo 3D no RT² também é reduzido. Para o caso da esfera, por exemplo, sua representação utiliza ao invés de algumas centenas ou milhares de pontos 3D dos seus polígonos, apenas quatro valores de ponto flutuante representando a sua equação espacial, sendo três desses valores para o ponto central, mais um para o tamanho do raio da esfera.
- Reflexões de luz com alta precisão. Em rasterização e com o auxílio de *shaders*, técnicas baseadas em *environment mapping* [2] são capazes de simular reflexões no ambiente, mas com baixíssima qualidade se comparado com o *ray tracing*, além de oferecerem altíssimo custo computacional.
- Transformações para coordenadas de câmera não são necessárias no RT². Não há sentido no uso de coordenadas de câmera para raios

secundários, por exemplo. O teste de visibilidade não se baseia em posições e sim se algum raio primário vai interceptar ou não um objeto. Além disso, no RT² o teste de profundidade por *z-buffer* não ocorre, sendo substituído pela própria travessia do raio na estrutura de aceleração, que ordena parcialmente os objetos da cena, sendo esta ordenação completamente independente da posição e orientação da câmera.

- Funcionalidades avançadas de câmera, como *zoom* e foco são facilmente definidas no RT², por serem parâmetros apenas dos raios primários. Em rasterização, envolve todo o processo de visibilidade e cálculo de perspectiva, fazendo com que algumas dessas técnicas sejam possíveis apenas através de *shaders*.
- *Shading* eficiente. No RT², a operação de coloração ocorre apenas em regiões que afetarão o resultado da imagem final. Em rasterização, o processo de *shading* vem antes do teste de visibilidade, sendo assim mais custoso neste aspecto. Este fato é verdadeiro até para os casos de se renderizar uma interface gráfica sobre a cena, como um *Head Up Display* (HUD). Neste caso, todos os *pixels* não transparentes do HUD são marcados para não executarem o *ray tracing*. Em rasterização, todos os *pixels* do *viewport* da janela são obrigatoriamente processados, mesmo quando claramente serão substituídos pela cor do HUD.
- Ausência de empilhamento de matrizes de modelo de visão e de perspectiva. Como não há a necessidade de transformar cada objeto em coordenadas de câmera, não faz sentido o empilhamento de matrizes do modelo de visão. Matrizes para a perspectiva também são desnecessárias, já que em *ray tracing* o modelo de perspectiva a ser utilizado depende apenas da origem e direção dos raios

primários. A ausência destas pilhas de matrizes facilita o trabalho do programador ao isentá-lo da necessidade de controle manual de operações de *push* e *pop* das matrizes, como ocorre em rasterização.

- Suporte nativo a *Hard Shadows* [2]. Em rasterização, são necessárias múltiplas passagens de *raster* para adquirir o mesmo resultado que se obtém numa única execução do *ray tracing*.

3.1. Arquitetura de *Software* do RT²

O RT² propõe se estabelecer como uma biblioteca *open source* de referência em *ray tracing* em tempo real. Para tal, uma parcela do tempo dedicado ao seu desenvolvimento foi destinada ao planejamento de uma arquitetura de *software* capaz de oferecer suas funcionalidades de maneira eficiente, escalável e transparente. Assim, o modelo de *software* adotado no RT² une a comunicação das duas arquiteturas de *hardware* (GPU e CPU) utilizadas no projeto em uma única interface de alto nível para o programador, chamada de RT² API.

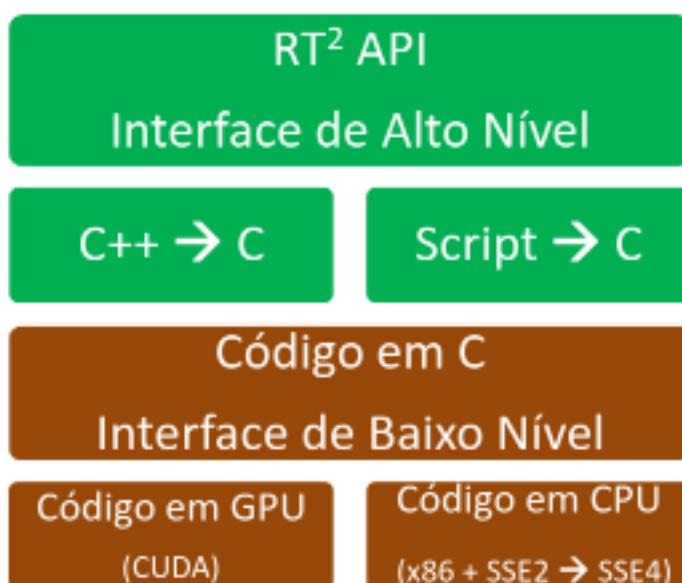


Figura 11. Arquitetura de *software* do RT².

A arquitetura do RT² em suas camadas de *software* é demonstrada na Figura 11. Utilizando a RT² API o desenvolvedor não precisa ter conhecimento sobre a implementação do RT² em baixo nível em CPU ou em GPU. Os únicos requisitos para o programador são:

- Conhecimentos básicos em Computação Gráfica, incluindo conceitos de álgebra vetorial e geometria analítica, manipulação de câmeras virtuais e manipulação de modelos 3D;
- Bons conhecimentos de programação orientada a objetos, pelo fato deste ser o paradigma de programação adotado na RT² API.

A RT² API se divide em dois módulos, um de programação em C++ e outro através do uso de arquivos de *script*. A versão atual do RT² implementa por completo o módulo em C++, faltando a implementação do módulo de *scripts*, apesar de sua especificação estar pronta. A implementação completa deste segundo módulo será adicionada em versões futuras do RT². O módulo de C++ oferece ao programador todas as funcionalidades do RT² através do uso da linguagem C++, sendo assim necessário um passo manual de compilação antes da execução. No caso do módulo de *scripts*, a utilização de uma linguagem de alto nível favorece o uso de compiladores-interpretadores do código, agilizando assim o processo de desenvolvimento do aplicativo.

Abaixo da camada de alto nível, há uma de baixo nível em C. Esta camada repassa todas as operações da camada de alto nível para um nível de decisão entre uso em CPU e GPU. Como exemplo, a operação de construção da *kD-Tree* da cena é realizada em CPU e outras, como o *ray tracing* em si são realizadas em GPU. Esta camada de controle em C escalona essas operações para serem realizadas em arquiteturas diferentes. Abaixo desta camada, existem dois módulos, um para a GPU e outro para a CPU. O módulo para a GPU é escrito em C com as extensões de CUDA enquanto que o em CPU é escrito em C e também em *assembly* x86 com

extensões SSE4 [4]. Cada um destes módulos tem a função de fazer a comunicação com a CPU e GPU no mais baixo nível possível no RT².

3.2. Ciclo de Renderização do RT²

Todo aplicativo interativo síncrono em tempo real obedece a um ciclo de operações bem definido. Para aplicativos 3D, este ciclo geralmente funciona como mostrado na Figura 12. O RT² é uma biblioteca destinada a executar única e completamente a etapa C.

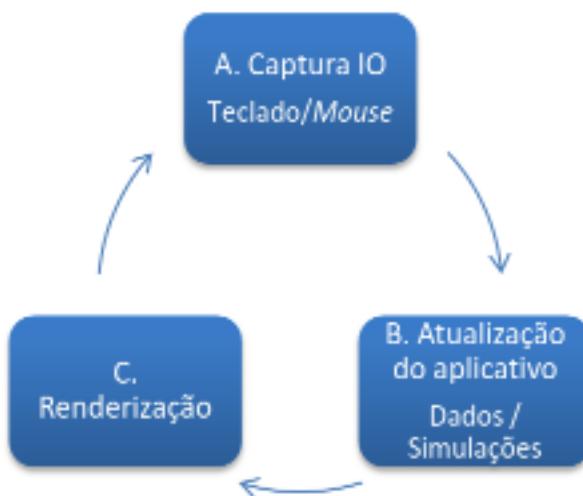


Figura 12. Ciclo de renderização.

O RT² executa dentro desta etapa de renderização uma sequência de operações, demonstradas na Figura 13 e descritas abaixo:

1. Criação/Atualização da *kD-Tree* da cena – Esta operação é completamente realizada em CPU. Reestrutura por completo a *kD-Tree* caso algum dos objetos da cena tenha se movido ou alterado a sua orientação;

2. Transferências de memória da CPU para a(s) GPU(s) – Nesta operação, ocorre a transferência de toda a *kD-Tree* atualizada da memória da CPU para a(s) da(s) GPU(s), bem como das coordenadas de objetos que se moveram ou tiveram a sua orientação alterada em relação ao ciclo anterior;
3. Execução do *ray tracing* em GPU(s) – Toda a simulação do *ray tracing* ocorre na(s) GPU(s), incluindo o processo de travessia em *kD-Tree* e *Shading*. Assim, é nesta operação que é realizado todo o cálculo computacionalmente intensivo do *ray tracing*;
4. Troca de *Buffer* – Esta operação final realiza a troca de *buffer* da imagem gerada pelo *ray tracer* na(s) GPU(s) para a tela, sendo o conceito básico da técnica de *double buffering* [1]. Esta troca pode ocorrer exclusivamente em GPU caso a execução envolva apenas um dispositivo de vídeo. Nos casos de Multi-GPU, é preciso transferir o *buffer* de cada dispositivo para a memória principal da CPU para posteriormente transferir da CPU para a placa de vídeo responsável pelo *display*.



Figura 13. Sequência de operações do RT².

3.3. Núcleo do RT²

Apesar do alto custo computacional, a técnica de *ray tracing* possui a vantajosa característica de ser altamente paralelizável. Como descrito na Seção 2.2, cada *pixel* da imagem origina um raio primário, que por sua vez gera raios secundários computados de maneira independente dos *pixels* vizinhos. Assim, o processamento de um *ray tracer* pode ser igualmente sub-dividido entre os *pixels* da imagem. Seguindo esta ideia, o núcleo do RT², implementado em CUDA, distribui o trabalho referente à cada *pixel* a diferentes *threads* do kernel. Como CUDA é uma arquitetura SIMT (*Single Instruction, Multiple-Thread*) [9], é importante que *threads* vizinhas sigam sempre que possível a mesma sequência de instruções, de forma que a instrução é executada simultaneamente em todas as *threads*. Assim, *pixels* próximos são executados em conjunto, favorecendo a execução paralela pelo fato de raios gerados de *pixels* próximos terem origens e direções semelhantes, com alta probabilidade de realizarem a mesma travessia na cena, intersectando os mesmos objetos e, consequentemente, executando o mesmo conjunto de instruções. Tais raios são conhecidos como raios coerentes, ou *coherent rays*. Implementações em CPU tiram proveito de raios coerentes ao utilizar algoritmos de travessia em pacotes de raios [20] agrupando a computação da busca de tais raios em conjuntos de instruções SIMD (*Single Instruction, Multiple-Data*). Entretanto, a arquitetura SIMT de CUDA é mais adequada para processar tais raios, pois não exige que o algoritmo de busca seja alterado, nem adiciona o significativo *overhead* necessário no caso do uso de pacotes de raios. Assim, o núcleo do RT² tira proveito da arquitetura de CUDA para oferecer uma melhor performance do que a encontrada em implementações baseadas apenas em CPU.

O núcleo do RT² oferece suporte a reflexões e sombras seguindo o modelo descrito por Turner Whitted [3]. Entretanto, por questões de otimização e

limitações de *hardware*, CUDA não suporta chamadas recursivas de funções, presentes no algoritmo original. Para resolver este problema, o núcleo do RT² tira proveito de um algoritmo iterativo adaptado do recursivo padrão: utilizando apenas a instância do raio primário, cada vez que uma intersecção é encontrada, o raio é transladado para a posição de intersecção e tem sua direção modificada para um sentido dependente do tipo de raio a ser gerado, podendo ser um raio de sombra ou um de reflexão. A iteração acaba se encontrar um raio de sombra, se o raio refletido não atingir nenhum objeto ou se o raio atingir um nível de reflexão máximo pré-determinado.

A versão atual do núcleo do RT² não suporta refração, pois demandaria uma pilha para tratar os casos de reflexão e refração, causando uma penalidade severa na performance em CUDA. Entretanto, há o suporte a elementos transparentes sem refração, ou seja, sem desvios na direção da luz, como ocorre em rasterização. O suporte total a elementos refratários será adicionado em futuras versões e quando a arquitetura de CUDA evoluir a ponto de não sofrer tanto com o impacto de uma pilha de dados a mais no *kernel*.

Apesar de *ray tracing* apresentar um alto grau de paralelismo, há uma demanda por grande largura de banda de memória, devido ao requisito de muitos acessos à memória principal. No caso do RT², todas as *threads* realizam um grande número de acessos à memória ao buscar informações dos objetos na cena. Estes acessos sendo realizados aleatoriamente podem gerar resultados muito aquém do esperado, principalmente no caso de acessos de memória em GPU. Estas unidades de processamento são mais sensíveis à leitura e escrita de memória por não disporem de uma hierarquia de memória tão desenvolvida como as de uma CPU.

Com o intuito de reduzir o número de acessos à memória global, o núcleo do RT² organiza a cena 3D na memória de forma que primitivas geométricas espacialmente próximas estejam alocadas em blocos de memória vizinhos. Desta maneira, ao realizar a travessia na cena, um raio provavelmente irá intersectar

grupos de objetos vizinhos que seguem o princípio da localidade [21] tanto em relação ao ambiente virtual 3D como à organização da memória global, aumentando as taxas de acerto (*hit*) na *cache* para tais objetos. A *kD-Tree* da cena também é estruturada de maneira a aumentar as chances de *hit* na *cache*. Para tal, a árvore é estruturada seguindo o *layout* de van Emde Boas [22], demonstrado na Figura 14. Tal modelo favorece a travessia ao garantir acertos na *cache* ao descer para um nó de nível ímpar da árvore. Desta forma, uma travessia terá pelo menos 50% de *hits* na *cache*, sendo estatisticamente um excelente caso para buscas aleatórias na árvore, buscas estas bastante frequentes em *ray tracing*.

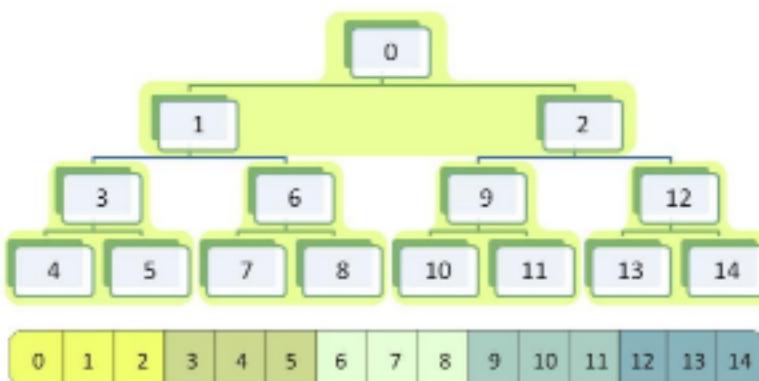


Figura 14. *Layout* de van Emde Boas.

Além de manter uma boa média de taxa de acertos na *cache*, a travessia numa árvore de van Emde Boas é considerada como um algoritmo do tipo “*cache-oblivious*” [15], ao não necessitar conhecimento prévio sobre parâmetros da *cache*, como o tamanho de linhas de *cache*. Tal fato é uma vantagem importante para o núcleo do RT², ao deixar o código tirando proveito corretamente de uma *cache* na atual e em futuras melhorias ou adaptações da arquitetura de CUDA.

Ao distribuir o trabalho de um *pixel* por *thread*, é possível utilizar apenas uma chamada de *kernel* para toda a execução de *ray tracing* em GPU, incluindo a travessia na *kD-Tree* e *shading*. Desta maneira, cada bloco de CUDA representa uma

sub-região da imagem a ser renderizada, comumente chamada de *tile*, como é demonstrado na Figura 15.

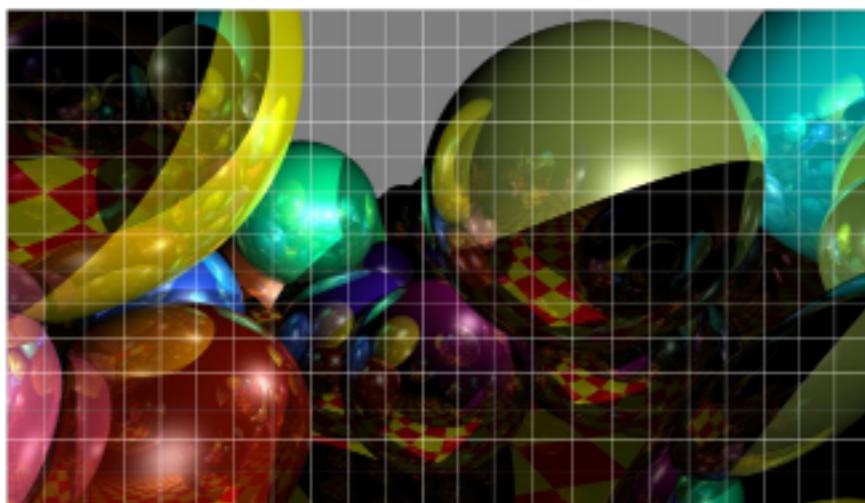


Figura 15. Renderização com *tiles*. Cada *tile* é processado em paralelo.

Esta abordagem oferece menor *overhead* que implementações com várias chamadas de *kernel*, já que no último caso, para cada mudança de estágio no *pipeline* é necessário armazenar na memória global todo o estado do processamento a ser repassado para a próxima etapa. Entretanto, a abordagem de um único *kernel* é falha para cenas com muitos objetos reflexivos, que geram muitos raios secundários com pouca coerência. Este problema é comum em CPU e agravado na GPU, por reduzir o grau de paralelismo da execução devido às muitas instruções divergentes e acessos randômicos à memória. Além disso, são frequentes os casos onde apenas uma pequena parte do bloco gera raios secundários, de tal maneira que as *threads* com poucos raios secundários que já terminaram sua execução ficarão em modo de espera, liberando recursos para outras *threads* apenas quando todo processamento do bloco for finalizado; isto não ocorre em CPU, onde um novo *pixel* é processado assim que o anterior for finalizado. Diante destes problemas, para o caso de renderização com apenas raios primários e poucos

pontos de reflexão, o RT² utiliza a abordagem de um único *kernel*. Nos casos de cenas com muitos objetos reflexivos, para reduzir a ociosidade de *threads*, é utilizada uma nova abordagem para tratamento de raios secundários, baseada em compactação do “*screen space*”. A próxima sub-seção trata deste tema.

3.3.1. Tratamento de Raios Secundários

Para resolver o problema relacionado com os raios secundários, a execução de cada *pixel* é sub-dividida em etapas, de forma que cada nova etapa corresponda ao novo raio refletido a ser computado. Desta forma, inicialmente é chamado o *kernel* responsável por processar todos os raios primários. Este *kernel* executa a travessia e realiza o *shading* dos raios primários. No fim deste *kernel* é realizado um mapeamento dos *pixels* que irão gerar raios secundários, utilizando um mapa booleano do “*screen space*”, configurando o valor 1 (*true*) apenas para *pixels* em que seus raios gerarão reflexões. Este mapa de booleanos é então compactado utilizando a técnica de soma paralela de prefixos, conhecida como PPS (*Parallel Prefix Sum*) [23]. Com o mapa compactado é possível, então, a execução de um novo *kernel* de *ray tracing* em que todos os *pixels* ativos no mapa de booleanos serão agrupados em blocos contíguos, garantindo que há demanda de processamento em todas as *threads* de um bloco. Agrupar estes *pixels* aumenta a “ocupância” ao reduzir significativamente o número de *threads* ociosas. Apesar da existência de *overhead* de algumas chamadas de *kernel* e maior uso da memória global, esta técnica possibilita ganhos de performance de até 30% se comparada à abordagem de um único *kernel*, como é demonstrado nos resultados descritos no capítulo 4.

3.3.2. Espaços de Memória

Toda a cena 3D, representada pela sua *kD-Tree*, é armazenada tanto em CPU como em GPU. Isto se deve ao fato de que a construção da *kD-Tree* da cena é feita em CPU e posteriormente a estrutura é enviada para a GPU. No RT², a construção da *kD-Tree* é feita em CPU utilizando a técnica de sub-divisão através da heurística baseada em área de superfícies com cortes perfeitos (*Surface Area Heuristic with Perfect Splits*) [24], resultando em construções de *kD-Trees* de alta qualidade.

Na GPU, tanto a *kD-Tree* como as propriedades geométricas e materiais dos objetos são armazenadas na memória global. Para reduzir o custo de acesso a tal memória, são utilizados *bindings* de textura na memória global, que oferece um sistema de *cache* de tamanho de 16KB e tamanho de linha de *cache* de 256 *bytes* nas placas gráficas atuais.

Pelo fato do *ray tracing* ser de alto custo computacional, alguns trechos do algoritmo demandam cálculos envolvendo um grande número de variáveis, aumentando significativamente a demanda por registradores para armazenar todas as variáveis do cálculo. Quando o compilador de CUDA não é mais capaz de reduzir o número de variáveis necessárias num escopo de bloco de forma a ter menos ou o mesmo número de registradores disponíveis que o de variáveis, todas as variáveis extras são destinadas à memória local, sendo seu acesso cerca de 600 vezes mais lento que um acesso ao registrador nas placas de vídeo atuais. Para amenizar este problema, o RT² tira proveito do espaço de memória compartilhada (*shared memory*) para utilizá-la como armazenadores de variáveis. Isto é vantajoso pelo fato de que o tempo de transferência entre a memória compartilhada e o registrador é de no máximo 4 ciclos de *clock* para acessos sem conflitos de banco [9], tendo um custo equivalente à uma simples operação de soma em ponto flutuante. Assim, a estrutura do raio, representada por um ponto de origem e um vetor de direção são armazenados na memória compartilhada. Como o raio é

utilizado em todas as etapas do algoritmo, o uso de registradores é significativamente reduzido.

Outra vantagem de colocar o raio no espaço de memória compartilhada é a possibilidade de realizar operações de indexação de ponteiros, necessárias em testes de interseção raio-triângulo e na travessia da *kD-Tree*. Como exemplo, na travessia em cada nó interno, não se sabe qual eixo vai ser utilizado em tempo de compilação, significando que é preciso realizar um acesso indexado entre o eixo x e z da origem e direção do raio, tendo tais vetores representados no intervalo [0, 1, 2]. Se o código de CUDA utilizar indexação de uma variável vetorial, a mesma será automaticamente armazenada na memória local para possibilitar a indexação, já que registradores não suportam indexação. Ao armazenar tais variáveis na memória compartilhada, este problema é contornado ao manter o uso de indexação de ponteiros de forma rápida.

O núcleo do RT² utiliza o espaço de memória constante de CUDA para armazenar a maioria dos parâmetros de função de *kernel*, como configurações de câmera e profundidade máxima do *ray tracing*. Ao definir parâmetros diretamente na função do *kernel*, tais parâmetros são automaticamente passados para o espaço de memória compartilhada. Como este espaço já está sendo utilizado para armazenar os raios, o armazenamento é alterado manualmente ao transferir os parâmetros para a memória constante, sendo este um espaço de leitura rápida de memória, com *cache*, não oferecendo perdas de performance visíveis em relação à memória compartilhada.

3.4. Ropes++

A nova abordagem para a travessia em *kD-Tree*, nomeada de Ropes++, proposta neste Trabalho de Graduação, reduz o número de leituras à memória global e de operações aritméticas necessárias na travessia de um nó-folha se

comparada ao algoritmo de Ropes (descrito na seção 2.4). Para tal, o algoritmo de intersecção raio-AABB é adaptado de forma que o teste seja reduzido apenas para localizar o ponto de saída do AABB. Isto é possível a partir da observação de que a face de saída de uma intersecção raio-AABB depende apenas da direção do raio, sendo sempre três faces de intersecção possíveis. Assim, o algoritmo reduz o número de leituras da memória global ao apenas acessar os três valores de ponto flutuante necessários para o cálculo de intersecção raio-AABB, ao invés dos seis que representam o *bounding box* da cena.

Para reduzir o número de registradores necessários, o Ropes++ também realiza uma simplificação do teste prévio de intersecção do raio com a cena, ao inicialmente localizar apenas a distância da origem do raio ao ponto de entrada. Observando o fato de que um raio que não intersecta o Δ AABB da cena também não intersecta qualquer sub-AABB da cena, o teste inicial entre raio e cena pode utilizar como distância do ponto de saída o primeiro nó-folha visitado. Sendo assim, o teste inicial de intersecção raio-cena agora é realizado em conjunto com o cálculo de intersecção com o primeiro nó-folha encontrado, cálculo este sempre necessário. Apesar de aumentar o tempo de travessia para o caso do raio não intersectar o *bounding box* da cena, este aumento é compensado em casos de intersecção, que ocorrem com maior frequência. Ao não precisar armazenar o ponto de saída da cena, a necessidade de alocar variáveis ou registradores é então amenizada, possibilitando um maior número de *threads* a serem processadas em paralelo. A técnica é demonstrada no Algoritmo 8

Algoritmo 8. Ropes++.

```
Ray r = (org, dir); // origem + direção do raio.  
Node curNode = root; // nó atual sendo visitado. Inicia na raiz da árvore.
```

RT² - Real Time Ray Tracer

```
// Em ropes++, apenas a distância ao ponto de entrada é necessário neste momento
tEntry = distância ao ponto de entrada do AABB da cena;
Object intersected; // objeto intersectado de retorno
tDistance; // distância entre raio e objeto intersectado
do{
    Point p = r.org + tEntry * r.dir; // ponto de entrada atual
    while(!curNode.isLeaf()){ // enquanto para nó interior...
        if(pEntry[curNode.axis] <= curNode.splitPosition) curNode = curNode.leftChild;
        else curNode = curNode.rightChild;
    }
    // O AABB está sendo representado por um array de 6 flutuantes (Min Point,Max Point)
    // é preciso de apenas 3 desses valores, que dependem apenas da direção do raio
    localFarX = (curNode.AABB[(r.dir.x >=0)*3] - ray.org.x) / ray.dir.x;
    localFarY = (curNode.AABB[(r.dir.y >=0)*3 + 1] - ray.org.y) / ray.dir.y;
    localFarZ = (curNode.AABB[(r.dir.z >=0)*3 + 2] - ray.org.z) / ray.dir.z;
    tExit = min(localFarX, localFarY, localFarZ); // a distância de saída é a menor de todas as 3.
    if(tExit <= tEntry) return MISS; // aqui é o caso do raio não intersectar com o AABB da cena.
    // busca por intersecção
    for (all object T in currentNode){
        I = Intersection(r,T,tEntry,tExit);
        if(I != null){
            Atualize tDistance e intersected, utilizando o melhor resultado atual;
        }
    }
    if(tDistance >= tEntry && tDistance <= tExit) return HIT(intersected ,tDistance) ;
    // Pega o próximo nó a ser visitado a partir dos ropes.
    curNode = curNode.ropes[ (r.dir[minLocalFarAxis] >=0)*3 +minLocalFarAxis];
}while(curNode != null);
return MISS;
```

4. Resultados

Os algoritmos de travessia descritos na seção 2.4 são analisados neste capítulo, sob a perspectiva de performance e consumo de memória. Esta análise gerou resultados significativos para a escolha do algoritmo de travessia utilizado no núcleo do RT². Esta escolha foi importante para ganhos de performance no RT², pelo fato do algoritmo de travessia ser responsável pela maior parte do custo computacional do *ray tracing*.

Todos os resultados encontrados e descritos neste capítulo são provenientes de testes realizados em um processador Intel Core2 Quad 2.66GHz com 4GB de memória RAM, uma placa de vídeo NVIDIA GeForce GTX 295, utilizando apenas um dos dois dispositivos (*Single GPU Mode*). O sistema operacional utilizado foi o Microsoft Windows XP 64-bit Service Pack 2.

Para os testes de uso de memória e performance, foram utilizadas seis cenas virtuais diferentes, descritas a seguir:

- Dragon – Cena que contém o modelo do *Stanford Dragon* [14], com 1.1 milhões de triângulos e duas luzes pontuais. Nesta cena foi realizado apenas *ray casting*;
- Bunny – Cena que contém o modelo do *Stanford Bunny* [14], com 69 mil triângulos e duas luzes pontuais. Nesta cena foi realizado apenas *ray casting*;
- Alien – Cena que contém o modelo do busto de um *Alien*, com 32 mil triângulos e duas luzes pontuais. Nesta cena foi realizado apenas *ray casting*;
- Spheres 1 – Cena com 1.2 milhões de esferas organizadas em um grande *bounding box*, com duas luzes pontuais. Nesta cena foi realizado apenas *ray casting*;

- Spheres 2 – Cena com 5 mil esferas e duas luzes pontuais. Nesta cena foi realizado um *ray tracing* com profundidade máxima de valor 6;
- Spheres 3 – Cena com mil esferas, um plano xadrez e duas luzes pontuais. Nesta cena foi realizado um *ray tracing* completo com suporte à sombras e profundidade máxima de valor 6.

Todas as cenas foram renderizadas utilizando *Phong shading* [8]. Estas cenas são exibidas na Tabela 2.

4.1. Análise Comparativa de Travessias em *kD-Tree*

Apesar de cada algoritmo de travessia seguir um conceito de busca diferente, há uma certa semelhança na estrutura dos mesmos. Inicialmente, todos realizam, antes da busca, um teste prévio de intersecção raio-ΑΑΒΒ da cena. Isto é feito de maneira que, caso o raio não atravesse a cena, não há sentido em realizar a travessia na *kD-Tree*. Outro motivo para este teste prévio se deve ao fato de que a maioria dos algoritmos de travessia necessita armazenar durante toda a busca duas variáveis que representam o ponto de intersecção de entrada e de saída, sendo utilizados como condição de término da travessia.

A implementação no RT² do algoritmo de travessia padrão armazena na memória local a pilha necessária para a travessia. Tal memória, apesar de não tirar proveito de uma *cache* e ter a mesma alta latência que o acesso manual à memória global, garante uma única leitura de variáveis locais em *threads* vizinhas, obtendo um melhor uso da arquitetura SIMT.

O algoritmo *kd-restart* não necessita de uma pilha, tornando o código do RT² mais simples e com menor demanda de variáveis, reduzindo portanto o número de registradores por *thread*, o que favorece o paralelismo. Entretanto, ao reiniciar a busca para a raiz quando a travessia encontrou uma folha, muitos nós são visitados

novamente, como é demonstrado na Tabela 1, resultando em um número maior de leituras da memória global.

Tabela 1. Uso de memória da *kd-Tree* vs número de nós visitados.

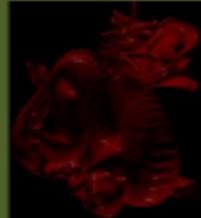
	Uso de Memória			# de Nós Visitados		
	DRAGON	BUNNY	ALIEN	DRAGON	BUNNY	ALIEN
Padrão	17.2 MB	7.7 MB	6.3 MB	38.4	18.3	13.1
<i>kd-restart</i>	15.3 MB	5.8 MB	4.4 MB	140.6	63.3	30.2
<i>push-down</i> (PD)	15.3 MB	5.8 MB	4.4 MB	120.2	55.2	27
<i>short-stack</i> (SS)	15.3 MB	5.8 MB	4.4 MB	80.3	43.3	24.3
PD + SS	15.3 MB	5.8 MB	4.4 MB	60.5	38.5	18.2
<i>Mips</i> [18]	47.1 MB	19 MB	13 MB	38.4	18.6	13.1
<i>Mips+I</i>	47.1 MB	19 MB	13 MB	38.4	18.6	13.1

O algoritmo *push-down* reduz o número de nós visitados no *kd-restart* adicionando ao código estruturas de controle para evitar o retorno da busca à raiz da árvore. Entretanto, há um decréscimo na performance, como é demonstrado na Tabela 2. Isto se deve à adição de estruturas de controle, gerando casos de divergências e consequentemente menor grau de paralelismo.

Em relação ao *short-stack*, sua implementação no RT² aloca a pilha na memória compartilhada, com um tamanho máximo possível de três nós, sendo esta uma pilha circular, sem ocorrências de estouro de pilha. Por estar na memória compartilhada, acessos a esta pilha são realizados mais rapidamente que no algoritmo padrão. Este algoritmo supera a performance do *kd-restart*, como demonstra a Tabela 2. Entretanto, da mesma forma que o *kd-restart*, no *short-stack* o pequeno tamanho da pilha significa retorno à raiz e consequente visitas à nós anteriormente visitados. Esta repetição de leituras torna este algoritmo mais lento que o de travessia padrão.

RT² - Real Time Ray Tracer

Tabela 2. Performance em diferentes cenários (1376x768 pixels - GTX 295).

DRAGON		BUNNY		ALIEN	
	Ray Casting		Ray Casting		Ray Casting
	2 Pontos de Luz		2 Pontos de Luz		2 Pontos de Luz
	Phong Shading		Phong Shading		Phong Shading
					
1.1 milhões de triângulos		69k triângulos		32k triângulos	
padrão	25.03 ms		13.91 ms		12.32 ms
kd-restart	36.91 ms		19.66 ms		16.34 ms
push-down(PD)	36.92 ms		19.76 ms		16.58 ms
short-stack(SS)	32.58 ms		17.58 ms		14.77 ms
PD + SS	29.73 ms		16.88 ms		13.58 ms
ropes[18]	-		78.74 ms (8800 GTX)		-
ropes++	22.19 ms		12.18 ms		10.82 ms
SPHERES 1		SPHERES 2		SPHERES 3	
	Ray Casting		Ray Tracing		Ray Tracing
	1 ponto de luz		Profundidade máxima: 6		Profundidade máxima: 6
	Phong Shading		2 pontos de Luz		2 pontos de Luz
					
1.2 milhões de esferas		5k esferas		1k esferas	
				1 plano xadrez	
padrão	6.39 ms		59.35 ms		66.06 ms
kd-restart	9.42 ms		84.59 ms		95.05 ms
push-down(PD)	9.64 ms		85.36 ms		96.97 ms
short-stack(SS)	8.22 ms		68.28 ms		77.36 ms
PD + SS	7.63 ms		61.82 ms		70.92 ms
ropes++	5.61 ms		34.50 ms		35.93 ms

O algoritmo híbrido de *push-down* e *short-stack* oferece um ganho de performance significativo se comparado ao *short-stack* em separado, mesmo com a adição de estruturas de controle divergentes do *push-down* e operações de pilha presentes no *short-stack*. Isto se deve ao fato do *push-down* ajudar a limitar o uso da pilha ao transferir o evento de *restart* a uma sub-árvore da cena, reduzindo o número de nós a serem armazenados, sendo uma situação conveniente devido ao pequeno tamanho da pilha.

O algoritmo de travessia com *ropes* para a arquitetura de CUDA resulta num menor número de nós visitados que todos os outros citados anteriormente, com exceção da travessia padrão. Apesar de aumentar o número de operações aritméticas, é importante observar que, ao contrário dos outros algoritmos, a travessia com *ropes* não necessita realizar a custosa operação de divisão (com um custo mínimo de 20 ciclos de *clock* [9]) a cada travessia de um nó interior, precisando realizar em troca três operações de multiplicação-adição, com custo de 4 ciclos de *clock* cada. Entretanto, é necessário um teste de intersecção raio-AABB a cada folha visitada para a escolha do *rope* a ser utilizado.

Como pode ser observado na Tabela 2, o novo algoritmo de travessia com *ropes*, *Ropes++*, proposto neste Trabalho de Graduação, obtém um melhor resultado em termos de performance para todas as cenas. O principal motivo desta melhoria está relacionado a um menor número de nós visitados e melhor teste de intersecção raio-AABB do nó-folha da kD-Tree, reduzindo de maneira significativa o total de acessos à memória global. Uma grande e exclusiva vantagem da abordagem com *ropes* está relacionada aos casos de travessias em raios de sombra e reflexivos, em que a busca já inicia na última folha intersectada, que contém a origem do raio. Nos outros algoritmos, a travessia sempre é iniciada no nó raiz da kD-Tree. Entretanto, como pode ser visto na Tabela 1, um algoritmo baseado em *ropes* demanda cerca de três vezes mais espaço de memória para armazenar a cena,

podendo ser um fator proibitivo para cenas de alta complexidade em placas com pouca memória.

Surpreendentemente, o segundo algoritmo com melhor performance foi o de travessia padrão. Apesar de alocar uma pilha na memória local, o mesmo tem a vantagem de ser um código simples, com poucas estruturas de controle e consequentemente de baixa divergência, sendo estes bons casos para uma implementação em CUDA. Por terem um maior número de nós visitados por travessia, todos os algoritmos propostos com adaptações da travessia padrão apresentaram resultado inferior. Tais algoritmos foram adaptações por questões de várias limitações de arquitetura de antigas GPUs, não sendo mais condizentes com modelos atuais. Entretanto, é importante observar que todas as técnicas implementadas atingiram o patamar de tempo real mesmo em *frames* de alta definição.

4.2. Análise Comparativa do Ropes++ em CPU e GPU

Para complementar a análise, foi realizada uma comparação de performance entre a implementação do Ropes++ em CPU e em GPU. A Figura 16 mostra que a implementação em GPU alcança ganhos de performance de até 21 vezes a mais que a implementação paralela em CPU, podendo chegar à 40 vezes mais com o uso de duas GPUs, como acontece na GTX 295 em modo multi-GPU. Entretanto, a travessia em CPU não tirou proveito de SSE4, extensão que provavelmente daria ganhos em torno de duas vezes no desempenho da CPU. Cabe ressaltar que mesmo com o uso dessas extensões, a performance em GPU teria um amplo ganho à frente da CPU.

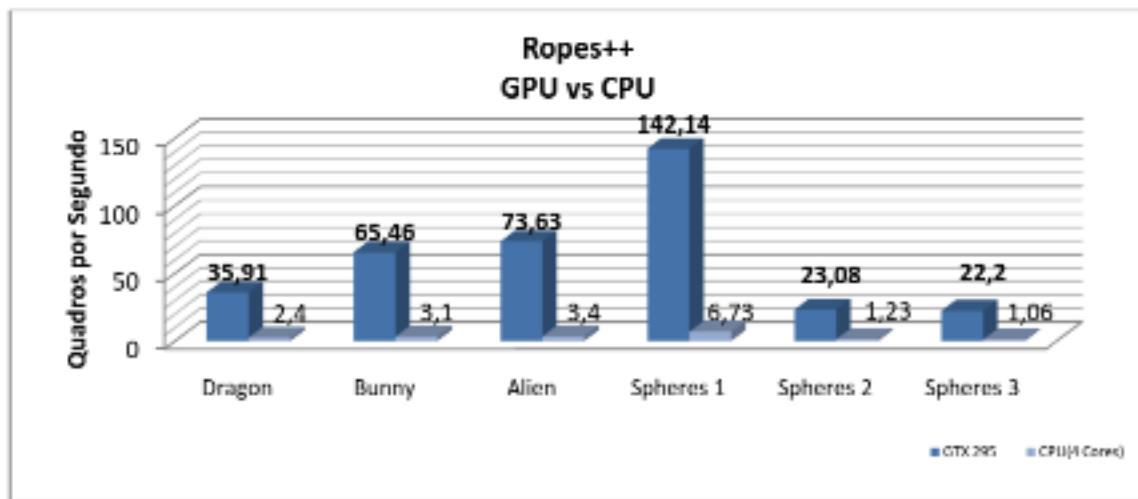


Figura 16. Gráfico comparativo de performance GPU vs CPU.

5. Conclusões

Este trabalho apresentou um vasto estudo sobre técnicas aplicadas ao desenvolvimento de *ray tracing*, tendo em vista a sua aplicação direta em renderização em tempo real. O principal objetivo desta pesquisa foi o de desenvolver um sistema de *ray tracing* em tempo real, nomeado de RT² - *Real Time Ray Tracer*, que une o potencial paralelo dos processadores gráficos com a eficiência de controle de uma CPU. Para tal, foram comparadas técnicas de travessia em *kD-Tree* em GPU, com o objetivo de obter um algoritmo que oferecesse o menor tempo de execução possível. Como resultado, surgiu neste trabalho uma nova abordagem para a travessia de *kD-Tree* com *ropes*, a *ropes++*, semelhante à proposta por Popov et. al. [18], trazendo ganhos entre 20% à 30% em performance se comparado às técnicas existentes, devido também às otimizações específicas para a arquitetura de CUDA. Esta arquitetura traz maior flexibilidade na programação do que em outros modelos de programação (como *shaders*), favorecendo implementações antes eficientes somente em CPU. Este é o caso da travessia padrão, que utiliza uma pilha, estrutura de dados difícil de implementar eficientemente em *shaders*.

Foi concluído também que, apesar das diferenças entre todas as travessias implementadas, todas foram capazes de renderizar imagens de alta definição em tempo real e de maneira escalável em relação ao número de processadores e a funcionalidades, fato que não acontecia na maioria das implementações originais de *ray tracing* em GPU, mencionadas neste trabalho.

Como fruto recente deste trabalho, foi também realizado uma submissão de um artigo para o SIBGRAPI 2009, sendo este artigo relacionado ao algoritmo Ropes++. No momento de escrita deste documento, o artigo se encontra em processo de revisão.

Assim, o RT², apesar de estar em sua primeira versão, já alcança o patamar de *ray tracing* em tempo real, provando que com os processadores gráficos atuais já é possível o uso desta técnica em um contexto de aplicativos interativos, como jogos, por exemplo.

5.1. Trabalhos Futuros

Como trabalhos futuros, são listadas algumas melhorias e alterações a serem realizadas:

- Suporte completo à refração;
- Implementação do módulo de *scripts*, descrito na seção 3.1;
- Suporte nativo a um maior número de primitivas geométricas, como cilindros, cones, superfícies de revolução e superfícies de Bézier;
- Suporte completo das funcionalidades presentes em versões atuais do OpenGL, como texturas e *MipMapping* [2].

Por fim, como há a intenção do autor em tornar o RT² uma biblioteca *open source* de referência em *ray tracing* em tempo real, outros desenvolvedores poderão observar a necessidade de muitas outras melhorias não percebidas pelo autor. Tais desenvolvedores terão a oportunidade de não somente opinar, como também fazer parte da contínua adição de melhorias neste projeto.

6. Referências Bibliográficas

- [1] Glassner, A., *An Introduction to Ray Tracing*, Academic Press, London, 1989.
- [2] Watt, A., *3D Computer Graphics*, Pearson – Addison Wesley, New York, 2000.
- [3] T. Whitted, "An improved illumination model for shaded display", *Communications of ACM*, v. 23, no. 6, ACM, New York, USA, 1980, pp. 343-349.
- [4] Intel. *Intel Corporation homepage*. Disponível em: <http://www.intel.com>. Acessado em 1 de Junho de 2009.
- [5] NVIDIA. *Nvidia Corporation homepage*. Disponível em: <http://www.nvidia.com>. Acessado em 1 de Junho de 2009.
- [6] Interactive Ray Tracing. *Quake 4: Ray Traced*. Disponível em: http://download.intel.com/pressroom/kits/research/poster_interactive_ray_tracing_quake_4_ray_traced.pdf. Acessado em 1 de Junho de 2009.
- [7] NVIRT – *Nvidia Interactive Ray Tracing API*. Disponível em: <http://realtimerendering.com/downloads/NVIRT-Overview.pdf>. Acessado em 20 de Junho de 2009.
- [8] V. Havran, "Heuristic Ray Shooting Algorithms", Phd thesis, Faculty of Electrical Engineering, Czech Technical University, Prague, 2001.
- [9] NVIDIA CUDA Programming Guide 2.1. Disponível em: http://developer.download.nvidia.com/compute/cuda/2_1/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.1.pdf. Acessado em 22 de Maio de 2009.
- [10] Appel, A. *Some techniques for shading machine renderings of solids*. AFIPS. SJCC, 37-45, 1968.
- [11] *Grand Theft Auto IV*. Disponível em <http://www.gta4.com/>. Acessado em 20 de Junho de 2009.
- [12] *AutoDesk Maya*. Disponível em <http://usa.autodesk.com/>. Acessado em 1

de Junho de 2009.

- [13] AutoDesk AutoCAD. Disponível em <http://usa.autodesk.com/>. Acessado em 1 de Junho de 2009.
- [14] The Stanford 3D Scanning Repository. Disponível em <http://graphics.stanford.edu/data/3Dscanrep/>. Acessado em 1 de Junho de 2009.
- [15] F. Jansen, "Data structures for ray tracing", *Eurographics Seminars on Data Structures For Raster Graphics*, Springer-Verlag, New York, USA, 1986, pp. 57-73.
- [16] T. Foley and J. Sugerman, "Kd-tree acceleration structures for a GPU Raytracer", *Proceedings of the ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*, ACM, New York, USA, 2005, pp. 15-22.
- [17] D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan, "Interactive k-d tree GPU raytracing", *Proceedings of the Symposium on 3D Graphics and Games*, ACM, New York, USA, 2007, pp 167-174.
- [18] S. Popov, J. Günther, H.-P. Seidel, and P. Slusallek, "Stackless kd-tree traversal for high performance gpu ray tracing", *Computer Graphics Forum*, v. 26, no. 3, Blackwell Publishing, 2007, pp. 415-424.
- [19] J. Günther, S. Popov, H.-P. Seidel, and P. Slusallek, "Realtime ray tracing on GPU with BVH-based packet traversal", *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing*, IEEE Computer Society, Los Alamitos, USA, 2007, pp. 113-118.
- [20] I. Wald, V. Havran, "On building fast kd-trees for Ray Tracing, and on doing that in O(N log N)", *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, 2006, pp. 61-70.
- [21] C. Wächter, A. Keller. "Instant ray tracing: the bounding interval hierarchy", *Proceedings of the Eurographics Symposium on Rendering*, 2006, pp. 139-149.
- [22] G. Brodal, R. Fagerberg, R. Jacob, "Cache Oblivious Search Trees via Binary

- Trees of Small Height", *Symposium on Discrete Algorithms*, 2002, pp.39-48.
- [23] Harris M., Sengupta S., Owens J. D., "Parallel prefix sum (scan) with CUDA", *GPU Gems 3*, Addison Wesley, 2007.
- [24] I. Wald, P. Slusallek, C. Benthin, M. Wagner, "Interactive Rendering with Coherent Ray Tracing", *Computer Graphics Forum*, v. 20, no. 3, 2001, pp. 153-164.
- [25] V. Havran., J. Bittner, J. Zára, "Ray tracing with rope trees", *Spring Conference on Computer Graphics*, Budmerice, Slovakia, 1998, pp. 130-140.
- [26] OpenGL API. Disponível em <http://www.opengl.org/>. Acessado em 1 de Junho de 2009.
- [27] DirectX SDK. Disponível em <http://msdn.microsoft.com/en-us/directx/default.aspx>. Acessado em 1 de Junho de 2009.

Assinaturas

Artur Lira dos Santos

Aluno

Veronica Teichrieb

Orientadora