UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

PARALELIZAÇÃO DE ALGORITMOS SEQUENCIAIS DE RAY-TRACING UTILIZANDO TÉCNICAS DE DIVISÃO E CONQUISTA

DISSERTAÇÃO DE MESTRADO

Cícero Augusto de Lara Pahins

Santa Maria, RS, Brasil

2015

PARALELIZAÇÃO DE ALGORITMOS SEQUENCIAIS DE RAY-TRACING UTILIZANDO TÉCNICAS DE DIVISÃO E CONQUISTA

Cícero Augusto de Lara Pahins

Dissertação apresentada ao Curso de Mestrado Programa de Pós-Graduação em Informática (PPGI), Área de Concentração em Computação, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de
Mestre em Ciência da Computação

Orientador: Prof. Dr. Cesar Tadeu Pozzer

Dissertação de Mestrado Nº. 0368/2015
Santa Maria, RS, Brasil

2015

de Lara Pahins, Cícero Augusto

Paralelização de Algoritmos Sequenciais de Ray-Tracing Utilizando Técnicas de Divisão e Conquista / por Cícero Augusto de Lara Pahins. – 2015.
69 f.: il.; 30 cm.

Orientador: Cesar Tadeu Pozzer
Dissertação (Mestrado) - Universidade Federal de Santa Maria, Centro de Tecnologia, Programa de Pós-Graduação em Informática, RS, 2015.

1. Algoritmo. 2. Divisão e Conquista. 3. Paralelização. 4. Ray-Tracing. I. Pozzer, Cesar Tadeu. II. Título.

E-mail: cicerolp@gmail.com

Universidade Federal de Santa Maria
Centro de Tecnologia
Programa de Pós-Graduação em Informática

A Comissão Examinadora, abaixo assinada,
aprova a Dissertação de Mestrado

PARALELIZAÇÃO DE ALGORITMOS SEQUENCIAIS DE RAY-TRACING UTILIZANDO TÉCNICAS DE DIVISÃO E CONQUISTA

elaborada por
Cícero Augusto de Lara Pahins

como requisito parcial para obtenção do grau de
Mestre em Ciência da Computação

COMISSÃO EXAMINADORA:

Cesar Tadeu Pozzer, Dr.
(Presidente/Orientador)

Benhur de Oliveira Stein, Dr. (UFSM)

João Luiz Dihl Comba, Dr. (UFRGS)

Santa Maria, 27 de Fevereiro de 2015.

AGRADECIMENTOS

Aos meus pais, **Elenir** e **Hermógenes**, e irmãs, **Liana** e **Sílvia**, por estarem sempre ao meu lado.

Ao meu orientador e amigo, **Prof. Cesar Tadeu Pozzer**, pelo auxílio e confiança desde o início de minha graduação.

A minha namorada, **Luísa**, pelo carinho e atenção.

A **CAPES**, pela oportunidade de desenvolver este trabalho.

E por fim, agradeço a todos de minha família e aqueles que de alguma forma me auxiliaram ou contribuíram para a realização deste trabalho.

”Mesmo que o futuro lhe pareça distante, ele está começando neste exato momento”

— MATTIE J.T. STEPANEK

RESUMO

Dissertação de Mestrado
Programa de Pós-Graduação em Informática
Universidade Federal de Santa Maria

PARALELIZAÇÃO DE ALGORITMOS SEQUENCIAIS DE RAY-TRACING UTILIZANDO TÉCNICAS DE DIVISÃO E CONQUISTA

AUTOR: CÍCERO AUGUSTO DE LARA PAHINS
ORIENTADOR: CESAR TADEU POZZER
Local da Defesa e Data: Santa Maria, 27 de Fevereiro de 2015.

Ray-tracing é uma importante técnica para a obtenção de imagens foto-realísticas. O algoritmo básico é bem conhecido e consiste no teste de colisão entre todos os raios e primitivas de uma cena, o que restringe sua aplicabilidade devido à alta complexidade. Portanto, métodos de aceleração são necessários. Recentemente, foram propostas soluções que não utilizam nenhuma estrutura de dados para a subdivisão espacial, fato inédito até então. Estas soluções subdividem a cena de maneira implícita através de técnicas de dividir para conquistar. Deste modo, este trabalho apresenta um novo algoritmo paralelo de ray-tracing baseado no paradigma de divisão e conquista que é capaz de executar concorrentemente instâncias individuais de algoritmos sequencias e unir os resultados a fim de obter a imagem final. O algoritmo introduz um esquema paralelo que, sem a utilização de nenhuma estrutura de dados para a subdivisão espacial, mantém o gerenciamento de memória mínimo e determinístico. Inicialmente, a cena é dividida em sub-cenas e os dados uniformemente distribuídos no hardware paralelo. Após, é executado um processo iterativo de três etapas até a conclusão do ray-tracing. Resultados mostram que a solução torna a execução de um algoritmo do estado-da-arte cerca de 2.42 vezes mais rápida em uma configuração de quatro threads.

Palavras-chave: Algoritmo. Divisão e Conquista. Paralelização. Ray-Tracing.

ABSTRACT

Master's Dissertation
Post-Graduate Program in Informatics
Federal University of Santa Maria

PARALLELIZATION OF SEQUENTIAL RAY-TRACING ALGORITHMS USING DIVIDE-AND-CONQUER TECHNIQUES

AUTHOR: CÍCERO AUGUSTO DE LARA PAHINS
ADVISOR: CESAR TADEU POZZER
Defense Place and Date: Santa Maria, February 27th, 2015.

Ray-tracing is an important technique to obtain photo-realistic images. The basic algorithm is well-known. Its applicability is restricted by the high demands on processing required to check collision between all rays and primitives of a scene. Therefore, acceleration methods are needed. Recently, solutions that do not use any data structure for spatial subdivision have been proposed. These solutions implicitly subdivide the scene by using divide-and-conquer techniques. Thus, this work presents a new parallel algorithm for ray-tracing based on the divide and conquer paradigm, which allows to run individual instances of sequential algorithms concurrently and then combine the results in order to get the final image. The algorithm introduces a parallel scheme that, without using any data structure for spatial division, maintains memory management minimum and deterministic. Initially, the scene is divided into sub-scenes and data uniformly distributed in the parallel hardware. After, an iterative three-step process is performed until the ray-tracing is completed. Results show that our solution speeds up a sequential state-of-the-art algorithm by about 2.42 times when running in a four-thread configuration.

Keywords: Algorithm, Divide-and-Conquer, Multithread, Ray-Tracing.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS

LISTA DE APÊNDICES

LISTA DE ABREVIATURAS E SIGLAS

AABB	*Axis-Aligned Bounding Box*
AASS	*Axis-Aligned Spatial Subdivisions*
BHV	*Bounding Volume Hierarchy*
BSP-Tree	*Binary Space Partitioning Trees*
CPU	*Central Processing Unit*
DACRT	*Divide-and-Conquer Ray-Tracing*
GPU	*Graphics Processing Unit*
IA	*Inteligência Artificial*
OSP	*Object Space Partitioning*
RT	*Ray-Tracing*
SIMD	*Single Instruction, Multiple Data*
SSE	*Streaming SIMD Extensions*
Kd-Tree	*K-dimensional Tree*

SUMÁRIO

1 INTRODUÇÃO

A técnica de Ray-Tracing é um algoritmo de síntese de imagens que pode produzir cenas com alta fidelidade e simplicidade. O algoritmo básico, apresentado por Whitted (1980), é bem conhecido e consiste no teste de colisão entre todos os raios e primitivas de uma cena, dessa forma, apresentando uma complexidade quadrática (*O(raios x primitivas)*). Tal complexidade não é adequada, já que as aplicações gráficas comumente trabalham com modelos de alta densidade e resolução (FRIEDRICH et al., 2006), o que se traduz em uma grande quantidade de raios e primitivas.

Como demostrado por Whitted (1980), é comum que a renderização de uma cena gaste cerca de 75% a 95% do tempo de processamento com os testes de colisões entre raios e primitivas, desta forma, rapidamente buscou-se melhorar o desempenho do algoritmo e diversos métodos de otimização foram propostos, como novas estruturas de dados (COMBA; PERSIANO, 1991; RESHETOV; SOUPIKOV; HURLEY, 2005; WäCHTER; KELLER, 2006; WALD et al., 2006; LAGAE; DUTRÉ, 2008), métodos numéricos e estatísticos (LEE; REDNER; USELTON, 1985; KOK; JANSEN, 1992), novas técnicas de geometria computacional (BRöNNIMANN; GLISSE, 2006; LAGAE; DUTRÉ, 2008), soluções paralelas (CARR et al., 2006; SHEVTSOV; SOUPIKOV; KAPUSTIN, 2007; CHOI et al., 2010), entre outras.

Métodos que se utilizam de estruturas de dados para a otimização do algoritmo básico de ray-tracing comumente subdividem a cena, composta por raios e primitivas, de maneira hierárquica. Embora muitas vezes técnicas complexas de subdivisão apresentem ganhos de desempenho relevantes, a exigência da implementação de esquemas robustos de gerenciamento da memória e a dificuldade na adaptação da técnica em diferentes hardware, como em GPUs, podem ser vistos como desvantagens pelos desenvolvedores, já que tais dificuldades resultam em um projeto de software mais custoso e potencialmente tornam impraticável a aplicação do método em cenas dinâmicas ou cenas que exijam taxas de quadros por segundo interativas.

Recentemente, Mora (2011) apresentou uma técnica chamada *Divide-and-Conquer Ray-Tracing* (DACRT) que, pela primeira vez desde o algoritmo de ray-tracing básico (WHITTED, 1980), não utiliza uma estrutura de dados para a subdivisão espacial da cena. Esta abordagem demostrou ser uma significativa contribuição como um novo paradigma de ray-tracing, já que a velocidade de renderização é comparável a outras soluções no estado-da-arte e o gerenciamento de memória é mínimo e determinístico, criando um novo tópico de pesquisa em uma área que

já era bem conhecida.

A proposta de Mora (2011) utiliza o conceito de divisão e conquista através de chamadas recursivas para, implicitamente, subdividir raios e primitivas em uma estrutura semelhante à kd-Tree (BENTLEY, 1975). A travessia, diferentemente do que normalmente ocorre em outros métodos de ray-tracing, ocorre em largura e é implementada de forma semelhante ao algoritmo de ordenação *QuickSort*. Aliado ao modo de travessia, o autor propõe e combina uma série de otimizações de baixo e alto níveis já conhecidas, como o empacotamento de raios em cones (AMANATIDES, 1984), que garantem ao algoritmo eficiência no tratamento de cenas dinâmicas. Os requisitos de memória podem ser calculados *a priori* como uma função linear do número de raios e primitivas envolvidas, assim, o algoritmo é capaz resolver problemas de grande magnitude e tornar prático a utilização de hardware com memória limitada.

1.1 Problema

Embora o algoritmo de Mora (2011) ofereça desempenho comparável a outras soluções em estado-da-arte, e faça isso com um gerenciamento simples e mínimo de memória, o núcleo de seu método foi projetado de maneira que não é possível a utilização de mais de uma linha de execução (*thread*) concorrentemente. Diferentemente de métodos em que a paralelização é possível, ou ainda trivial, o algoritmo de Mora (2011) tem como premissa e lógica a execução sequencial de diversas etapas recursivas. Em seu trabalho, o autor apresenta todos os resultados sendo executados por um único núcleo de processamento da CPU, contrariando o atual desenvolvimento do aumento de desempenho em hardware e dificultando, ou ainda tornando impraticável, a utilização de hardware largamente paralelos, como as GPUs.

A resolução de subdivisões atuais, no processo de execução do algoritmo de *Divide-and-Conquer Ray-Tracing*, depende necessariamente da resolução de subdivisões anteriores. Além disso, a utilização de uma série de variáveis globais para a construção da lógica do método, e outros requerimentos como o armazenamento linear dos raios e primitivas da cena, computações locais (*computations in place*), e a travessia de frente para trás associada ao término antecipado de raios (*front-to-back traversal associated with early-ray termination*), são estritamente relacionadas ao fato de o núcleo da abordagem ser executado sequencialmente.

Deste modo, as ideias propostas por Mora (2011) apresentam-se como uma relevante oportunidade para o desenvolvimento de métodos que otimizam o ray-tracing através do uso da técnica de divisão e conquista (inédita no tópico), mas, que ainda assim, possam extrair o

potencial paralelo do hardware atual.

1.2 Objetivo

Este trabalho tem como objetivo principal desenvolver *(i)* um novo método de ray-tracing paralelo que utilize o paradigma introduzido em (MORA, 2011), isto é, o conceito de divisão e conquista para a renderização da cena, *(ii)* a não utilização de uma estrutura de dados para a subdivisão espacial, e *(iii)* o gerenciamento mínimo e determinístico de memória.

Como objetivos adicionais, pretende-se que o método permita a paralelização de diferentes soluções de ray-tracing, de maneira que um algoritmo sequencial possa ser encaixado e executado concorrentemente, ao estilo "plug-and-play". Para tal, deve-se apresentar um esquema de divisão e agrupamento da cena em partes independentes.

1.3 Estrutura do Trabalho

Este trabalho busca utilizar a abordagem de divisão e conquista para a proposta de um novo algoritmo paralelo de ray-tracing. No Capítulo 2, Divide-and-Conquer Ray-Tracing, são apresentados os conceitos introduzidos pelo algoritmo DACRT, fundamentais para uma melhor compreensão de nossa solução. A Seção 2.2, Implementação, aborda aspectos técnicos como o término antecipado de raios, o *streaming* rápido de triângulos através de métodos de compactação, entre outros.

A revisão bibliográfica é realizada no Capítulo 3. Inicialmente são discutidos trabalhos que propõem métodos de aceleração de ray-tracing baseados em estruturas de subdivisão espacial já conhecidas. A utilização de hardware paralelos, especialmente placas gráficas, e técnicas híbridas são os focos destes trabalhos. Posteriormente, são avaliados métodos que buscam aprimorar o algoritmo DACRT de Mora 2011. É discutida a aplicabilidade das otimizações à proposta deste trabalho.

O Capítulo 4, Desenvolvimento, apresenta a arquitetura geral da solução desenvolvida. É introduzido um modelo de implementação para processadores com múltiplos núcleos. As seções desse capítulo abordam diferentes etapas de otimizações, bem como os processos necessários para a obtenção da imagem final.

No Capítulo 5 são analisados os resultados obtidos com a execução do algoritmo proposto. São utilizados modelos de teste conhecidos para as medições de desempenho. A eficiên-

cia da solução é investigada em uma série de configurações.

Finalmente, o Capítulo 6 aborda as conclusões do trabalho e objetivos alcançados. São apresentadas contribuições, limitações e sugestões para trabalhos futuros.

2 DIVIDE-AND-CONQUER RAY-TRACING

O *Divide-and-Conquer Ray-Tracing* (DACRT) (MORA, 2011) é um algoritmo que, pela primeira vez desde Whitted (1980), o método básico de ray-tracing, não armazena nenhuma estrutura de dados quando realiza subdivisões espaciais. A computação dos testes de colisão entre raios e primitivas de uma cena ocorre localmente, no conceito de computação local (*computations in place*). O DACRT contribui significativamente na área de ray-tracing, de forma que apresenta um novo paradigma de subdivisão espacial, possui desempenho comparável a outras soluções do estado-da-arte, e por fim, possui um gerenciamento mínimo e determinístico do uso de memória.

Diferentemente das técnicas de aceleração convencionais de ray-tracing, como por exemplo, aquelas que visam reduzir o custo médio de colisão raio/primitiva (KAY; KAJIYA, 1986; GOLDSMITH; SALMON, 1987; GLASSNER, 1988; MACDONALD; BOOTH, 1990), visam reduzir o número total de raios testados (JOY; BHETANABHOTLA, 1986), ou ainda, substituem raios individuais por entidades gerais (HECKBERT; HANRAHAN, 1984; AMANATIDES, 1984; SHINYA; TAKAHASHI; NAITO, 1987), o algoritmo DACRT é uma adaptação do *conceito de divisão e conquista*, reduzindo a complexidade do algoritmo de ray-tracing básico através da subdivisão implícita da cena.

Quando comparado contra outros ray-tracers no estado-da-arte, o algoritmo DACRT implementado de maneira eficiente, consegue ser tão rápido, e em alguns casos melhor, para cenas dinâmicas e semelhante para cenas estáticas (MORA, 2011). Tal fato se torna relevante quando analisamos a simplicidade do método em contraste com as soluções mais complexas, as quais comumente utilizam hardwares largamente paralelos ou especializados para obter o nível de desempenho esperado. O DACRT foi projetado a fim de ser executado por um único núcleo de processamento, sendo esta a característica que o torna promissor, uma vez que seu potencial paralelo pode ser considerado inexplorado. Apesar disso, sua lógica de funcionamento não permite o uso de tarefas concorrentes.

Devido à natureza do DACRT, o uso de memória pode ser calculado linearmente em função do número de raios e primitivas, permitindo que cenas altamente complexas e grandes possam ser resolvidas. Outra característica é o fato dos testes de colisão serem computados diretamente dentro da cena e a dispensabilidade de uma etapa de ordenação, assim simplificando em muito a tarefa de engenharia de software.

2.1 Definição

O algoritmo DACRT é baseado na observação de que a complexidade para encontrar a colisão mais próxima com algum raio não necessariamente implica em *O(raios x primitivas)*, como apresentada no ray-tracing básico, uma vez que o problema pode ser simplificado a partir de um esquema de divisão e conquista. Este esquema subdivide implicitamente a cena, reduzindo o espaço do problema e limitando o número de testes entre raios e primitivas.

O esquema de divisão e conquista é implementado através de funções recursivas que, a cada chamada, geram subconjuntos de raios e primitivas mais estreitos. A subdivisão é dita implícita por não utilizar nenhuma estrutura de dados para a sua realização, de modo que a própria pilha de recursão indica os limites de fronteira de cada conjunto.

É definido como subconjunto inicial o universo de raios e primitivas. Assim, o algoritmo compara estes subconjuntos com duas constantes arbitrariamente fixadas, uma referente ao número de raios e outra ao número de primitivas. Caso alguma das comparações for verdadeira, o problema é solucionado por um algoritmo de ray-tracing básico, ou seja, é admitido que o espaço é pequeno o suficiente e não são mais necessárias subdivisões. Caso contrário, o espaço corrente é subdivido em dois e, para cada subconjunto, uma chamada recursiva é feita contendo apenas os raios e primitivas que dela fazem parte, como mostrado no Código 2.1.

```
1  procedure DACRT(Space E, SetOfRays R, SetOfPrimitives P)
2  begin
3    if R.size() < rLimit or P.size() < pLimit then
4      NaiveRT(R, T)
5    else begin
6      {E_i} = SubdivideSpace(E)
7      for each E_i do
8        SetOfRays R' = R ∩ E_i
9        SetOfPrimitives P' = P ∩ E_i
10       DACRT(E_i, R', P')
11     end do
12   end
13 end
```

Código 2.1 – Algoritmo do DACRT básico. Adaptado de Mora (2011).

Note que a travessia, diferentemente do que normalmente ocorre em outros métodos de ray-tracing que subdividem o espaço, ocorre em largura e é implementada à forma do algoritmo de ordenação *QuickSort*.

2.2 Implementação

A implementação do algoritmo DACRT proposta por Mora (2011) é baseada em Subdivisões Espaciais com Eixos Alinhados (*Axis-Aligned Spatial Subdivisions*) tridimensionais. Semelhante ao processo de construção de uma *kd-Tree*, diferencia-se pelo fato do algoritmo não armazenar a estrutura de dados para a representação da árvore, fazendo-a de maneira implícita.

São características do método a realização de computações locais, a travessia de frente para trás associada ao término antecipado do raio (*front-to-back traversal associated with early-ray termination*), o empacotamento de raios em cones (*conic packet tracing*), o *streaming* rápido de triângulos através de métodos de compactação (*fast triangle streaming*) e a determinação simplificada do ponto de subdivisão (*simplified split determination*). A combinação de tais características, juntamente com a restrição a primitivas de triângulos, garante desempenho comparável a outras soluções no estado-da-arte.

2.2.1 Computações Locais

Com o objetivo de manter o uso de memória mínimo e determinístico, Mora (2011) propôs um esquema no qual o armazenamento simplificado dos dados da cena, juntamente com uso de computações locais, permite que os requisitos de memória sejam calculados no início do algoritmo e que, além disto, não se alterem durante a execução do método. Deste modo, o algoritmo armazena raios e triângulos linear e separadamente em listas sequenciais ou vetores.

Dois pivôs são utilizados a fim de demarcar os limites dos subconjuntos de raios e triângulos, o *RayPivot* e o *TrianglePivot*, únicos para cada chamada recursiva. A demarcação é necessária pois a cada chamada recursiva do algoritmo diminui-se o espaço do problema. Ao retornar de uma recursão, o algoritmo utiliza os valores de pivôs armazenados na pilha de execução. Um pivô é definido como uma variável que aponta para um posição de memória ou endereço do vetor.

A fim de reduzir a transferência de memória na manipulação dos raios e triângulos, ao início do algoritmo são criados dois vetores de índices que correspondem aos dados reais. Deste modo, somente variáveis simples (inteiros) são manipuladas no decorrer do algoritmo. O armazenamento e ordenação dos dados reais, uma vez criados, não são mais alterados, como mostrado na Figura 2.1.

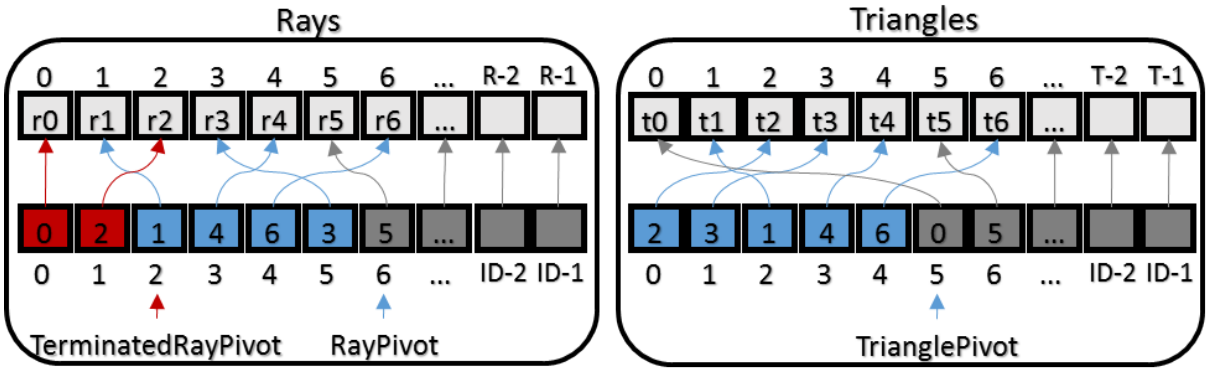

Figura 2.1 – Relação entre o vetor de índices (abaixo) e o vetor de dados (acima). Os pivôs *RayPivot* e *TrianglePivot* representam a subdivisão corrente (quadrados azuis), e são únicos para cada chamada recursiva ao DACRT, enquanto que o pivô *TerminatedRayPivot* é global para todo algoritmo e representa os raios já colididos com algum triângulo (quadrados vermelhos).

Note que, mesmo incluindo a pilha de execução resultante das chamadas recursivas, a memória que será utilizada pelo algoritmo pode ser previamente calculada utilizando a técnica de computação local apresentada.

2.2.2 Travessia de Frente para Trás Associada ao Término Antecipado do Raio

A cada chamada recursiva do DACRT, é executado um processo de filtragem que delimita os índices de raios e triângulos pertencentes à subdivisão corrente, como mostrado na Figura 2.2.

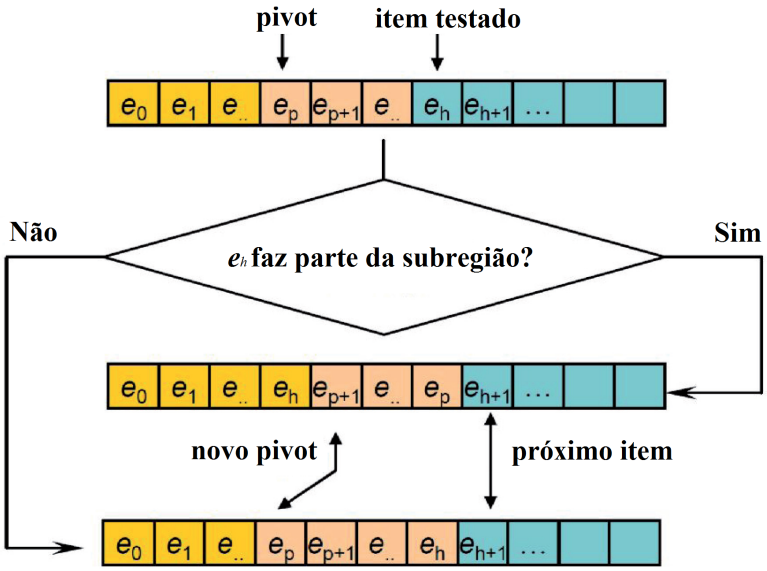

Figura 2.2 – Processo de filtragem executado a cada chamada recursiva ao DACRT. Adaptado de Mora (2011).

No processo de filtragem dos triângulos, os índices são analisados entre o primeiro item e o atual *TrianglePivot*, desta forma gerando um novo *TrianglePivot*. Somente os triângulos que permanecerem à esquerda do novo *TrianglePivot* fazem parte da subdivisão atual. Triângulos à direita não devem ser considerados. Note que cada conjunto é dividido em dois subconjuntos, esquerda e direita, assim o processo de filtragem sempre é executado para ambas as partes em um mesmo nível da recursão.

Semelhantemente, o processo de filtragem é aplicado aos raios, entretanto os limites da filtragem compreendem somente os raios pertencentes ao intervalo [*TerminatedRayPivot*, *RayPivot*]. O pivô *TerminatedRayPivot* demarca raios em que o teste de colisão contra algum triângulo foi positivo e que, deste modo, não são mais significativos à execução do ray-tracing. Esta demarcação é definida pelo autor como o término antecipado dos raios (*Early-ray Termination*) e é executa no momento de retorno da função do ray-tracing básico.

No momento da execução do ray-tracing básico, todos raios entre o intervalo [*TerminatedRayPivot*, *RayPivot*] e triângulos entre o intervalo [0, *TrianglePivot*] são testados utilizando o algoritmo de colisão descrito em (MORA, 2011). Os raios que colidem com triângulos são movidos para a posição do *TerminatedRayPivot* e este incrementado, de modo similar ao mostrado na Figura 2.2. Com este processo, triângulos oclusos são rapidamente descartados e têm seu impacto no tempo total de renderização significativamente diminuído.

Como visto anteriormente, o processo de filtragem é executado duas vezes para cada chamada recursiva ao DACRT, ou seja, uma vez para cada filho da subdivisão corrente (esquerdo e direito, como em uma kd-tree). A análise dos filhos gerados deve ser feita respeitando a posição da câmera em relação ao centro de cada subdivisão, representada por uma caixa delimitadora alinhada aos eixos (*Axis-Aligned Bounding Box*, AABB). A subdivisão a ser primeiramente analisada, e os pivôs gerados, é aquela que se encontra a uma distância menor se comparada à irmã. Tal processo se deve ao fato dos raios exigirem ser testados contra os triângulos na ordem em que estes aparecem na cena e, caso não colidam com nenhum triângulo, serem analisados pela subdivisão irmã através do processo de filtragem. Esta etapa do processo é definida pelo autor como a travessia de frente para trás (*Front-to-back Traversal*).

A fim de estabelecer o menor uso de memória possível, o autor recomenda os algoritmos de colisão raio/AABB de Williams (2005) e de colisão triângulo/AABB como uma modificação (descrita na Subseção 2.2.4) de Akenine-Moller (2005). Ambos apresentam baixos requisitos de memória e custos computacionais.

2.2.3 Empacotamento de Raios em Cones

A substituição de raios individuais por entidades gerais em problemas de ray-tracing é uma técnica bem conhecida e utilizada tanto para otimização, como Wald (2001) que propõe o uso de pirâmides para diminuir o número de travessias executadas em uma estrutura de aceleração, tanto na criação de efeitos visuais, como Amanatides (1984), que propõe o uso de cones para fins de remoção de serrilhados (*antialiasing*).

A implementação do algoritmo DACRT proposta por Mora (2011) utiliza raios empacotados através de cones. Este empacotamento busca diminuir o número de testes de colisão entre raio/AABB que, segundo o autor, é cerca de 20 à 60 vezes o número original de raios. Assim, caso nenhum método de empacotamento fosse utilizado, a renderização dos raios primários poderia apresentar uma grande perda de desempenho.

Técnicas de empacotamento são comumente utilizadas no processamento de raios primários, uma vez que se baseiam na emissão de raios de um único ponto em comum do espaço, como em fontes de luz pontuais. Esta característica permite a simplificação dos testes de colisão e a redução da largura de banda necessária. Raios secundários ou randômicos devem utilizar adaptações ou outras técnicas.

Os cones são definidos por um vetor direção, um ângulo α, delimitado de maneira que nenhum dos ângulos formados entres os raios empacotados e o vetor direção seja maior do que ele, e uma variável de 64-bit que armazena os estados de término de cada raio dentro do pacote. O número de raios empacotados é limitado a 8x8, assim cada cone pode no máximo substituir 64 testes de colisão raio/AABB, como mostrado na Figura 2.3. Note que em uma cena de, por exemplo, resolução 640x480 pixels existe um total de 307.200 raios ou, caso utilizada a técnica de empacotamento, 4800 cones.

A inclusão da técnica de empacotamento no algoritmo DACRT é simples, bastando apenas a troca dos raios do Código 2.1 por cones, como mostrado no Código 2.2. Deve-se criar um novo vetor de índices e um novo pivô de término, chamado *TerminatedConePivot*.

Anteriormente à chamada da função *DACRT_Packet* (Código 2.2), os pivôs *TerminatedRayPivot* e *TerminatedConePivot*, e a variável de 64-bit de cada cone, devem ser inicializados em zero. O vetor de índices dos raios é considerado vazio, de modo contrário ao vetor de índices dos cones, que é totalmente preenchido. Os futuros índices dos raios serão gerados pela função *FlushCones* (Código 2.2) e deduzidos implicitamente do índice de cada cone.

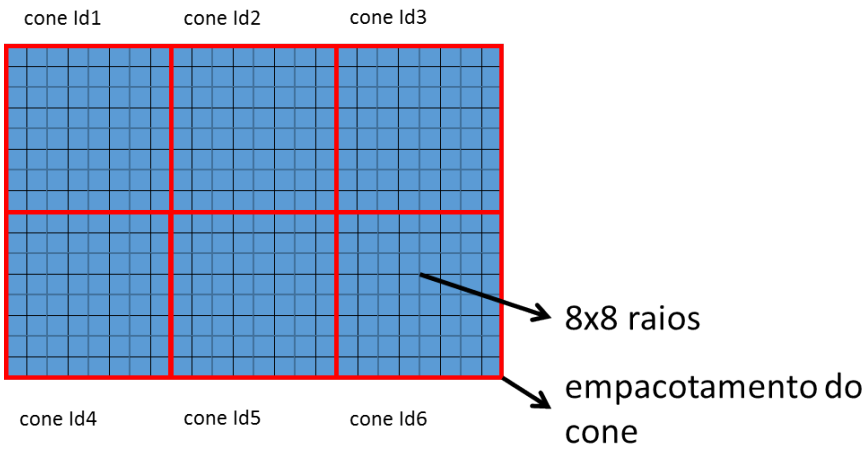

Figura 2.3 – Esquema de empacotamento dos raios.

```
1  procedure DACRT_Packet(Space E, SetOfCOnes C, SetOfPrimitives P)
2  begin
3    if C.size() < cLimit or P.size() < pLimit then
4      begin
5        SetOfRays R = FlushCones(C);
6        DACRT(E, R, P)
7      end
8    else begin
9      {E_i} = SubdivideSpace(E)
10     for each E_i do
11       SetOfCones C' = C ∩ E_i
12       SetOfPrimitives P' = P ∩ E_i
13       DACRT_Packet(E_i, C', P')
14     end do
15   end
16 end
```

Código 2.2 – Algoritmo do DACRT empacotado. Adaptado de (MORA, 2011).

Diferentemente do que ocorre no Código 2.1, onde o ray-tracing básico (*NaiveRT*) é executado quando o número de raios e triângulos for menor que uma das duas constantes arbitrariamente fixadas, o Código 2.2 chama a função *DACRT* definida anteriormente, de modo que o número de triângulos e cones, e não mais raios, são menores que as constantes. A detecção da colisão ente raios e triângulos ocorre em duas etapas, uma antes e outra depois da chamada da função *DACRT*.

Antes da chamada à *DACRT*, é necessária a transferência dos índices dos raios não terminados, relativo ao índice dos cones, para o vetor de índices de raios e após o pivô *TerminatedRayPivot*, o que determina o novo *RayPivot*.

Depois da chamada à *DACRT*, deve-se atualizar a variável de 64-bit de cada cone com a informação de quais raios que colidiram com algum triângulo e verificar se algum desses cones

possui todos os bits da variável igual à 1. Caso positivo, deve-se transferi-lo para antes do pivô *TerminatedConePivot* através do processo de filtragem mostrado na Figura 2.2. Esta etapa pode ser realizada avaliando-se o valor do pivô *TerminatedRayPivot* antes e depois da chamada à *DACRT*.

O algoritmo de colisão entre cone/AABB é simples e se baseia na análise, primeiramente, do vetor direção em relação ao espaço da AABB (como em um teste raio/AABB) e, caso negativo, na análise das 12 arestas da caixa conforme o algoritmo descrito por Eberly (2000).

2.2.4 *Streaming* Rápido de Triângulos Através de Métodos de Compactação

Aplicações de alto desempenho comumente exigem o *streaming* rápido de blocos da memória. Além disto, é benéfico ao sistema de memória cache que os dados processados sejam relativamente pequenos, desta forma aumentando a chance de permanecerem em níveis hierárquicos mais próximos ao processador. O *streaming* de triângulos é uma das operações mais realizadas durante testes de colisão em ray-tracers atuais, especialmente naqueles que são capazes de processar cenas altamente complexas.

O gerenciamento da memória e armazenamento de triângulos do DACRT é baseado na utilização do conjunto de instruções *Streaming SIMD Extensions* (SSE). Tal conjunto de instruções de baixo nível tem como objetivo a aceleração de operações e tarefas SIMD (*Single Instruction, Multiple Data*). Mora (2011) observa que todos os algoritmos de colisão (raio/-triângulo, raio/AABB, triângulo/AABB, cone/AABB) podem ser otimizados com o uso dessas instruções. Desta forma, é proposto um esquema de representação de triângulos a fim de maximizar os ganhos de desempenho.

Uma representação padrão de 36 bytes foi escolhida para o armazenamento dos triângulos, de modo que os valores mínimo e máximo de cada eixo estão em listas separadas. Este esquema resulta na geração de três listas de dados decompostos, como na Figura 2.4. A reconstrução das coordenadas originais é feita em tempo real com o auxílio de uma quarta lista de dados, a qual armazena os pontos médios de cada coordenada e mais 4 bytes para alinhamento de memória. O processo de reconstrução, segundo Mora (2011), não influencia em uma parcela significativa do tempo total do algoritmo e é considerado válido.

A fim de acelerar o teste de colisão triângulo/AABB, o autor descreve uma série de modificações e simplificações ao bem conhecido algoritmo de Akenine-Möller (2005). São retirados alguns testes específicos do algoritmo, como aqueles que tratam de casos excepcionais,

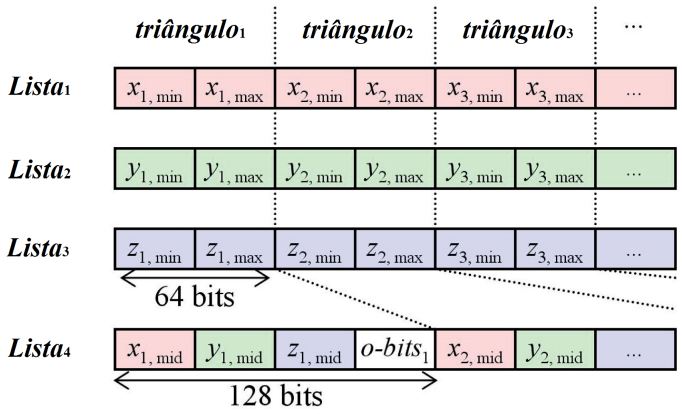

Figura 2.4 – Esquema de armazenamento de triângulos. Adaptado de Mora (2011).

e é adicionada uma etapa de escolha do refinamento dos testes conforme o número de triângulos (fixado em 100) da subdivisão corrente. Desta forma, são admitidos resultados imprecisos em porções pequenas do resultado final.

Como decorrência das modificações e simplificações do algoritmo de Akenine-Möller (2005), juntamente com o uso de instruções SSE, é possível realizar quatro testes de colisão triângulo/AABB concorrentemente, gerando um significativo aumento de desempenho ao algoritmo. Note que os testes concorrentes ocorrem a nível de instruções SIMD, e não com o uso de dois ou mais processadores.

2.2.5 Determinação Simplificada do Ponto de Subdivisão

O algoritmo DACRT utiliza duas técnicas para a definição da posição de corte das subdivisões, as quais são definidas conforme o número de triângulos a serem processados na subdivisão corrente. O uso destas duas técnicas, uma mais exata que outra, tem como base a noção de que não são necessários cálculos precisos para porções pequenas do resultado final.

Subconjuntos pequenos de triângulos, aqueles com um número inferior à 10000, executam a escolha do corte analisando todos os objetos, sempre retornando as coordenadas referente ao corte mediano do espaço.

O método utilizado em subconjuntos pequenos não é suficientemente robusto quando aplicado a subconjuntos grandes, uma vez que a análise de todos os objetos acarreta em uma relevante perda de desempenho do algoritmo. Desta forma, subconjuntos grandes, aqueles com um número superior à 10000 objetos, são analisados em um intervalo de 50 (por exemplo, [0,50,100,150,...]), de modo que é calculada a média da posição dos centros de cada triângulo no eixo escolhido. O cálculo da média é delimitado pelas coordenadas da AABB, como é

mostrado na Figura 2.5.

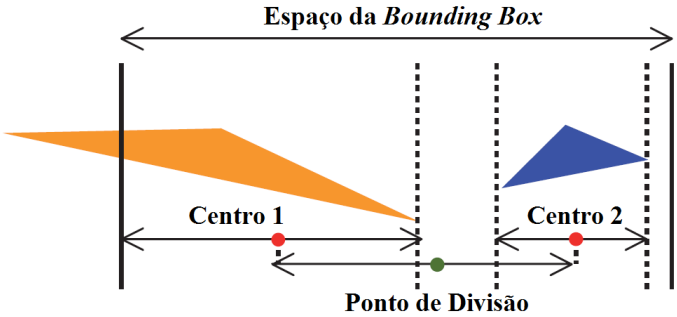

Figura 2.5 – Esquema de divisão para conjuntos grandes de triângulos. Adaptado de Mora (2011).

Mora (2011) argumenta que o uso de duas técnicas, uma mais exata que outra, para a escolha da posição do corte das subdivisões é válido, já que não há prejuízo visual aparente e o ganho de desempenho para subconjuntos grandes é de cerca 20% caso fosse utilizado a técnica de divisão para conjuntos pequenos.

3 REVISÃO BIBLIOGRÁFICA

Na literatura, existe um vasto número de pesquisas realizadas na área de aceleração de ray-tracing. Segundo Arvo (1989), pode-se classificar os métodos naqueles que (i) buscam a simplificação dos testes de colisão raio/primitiva, (ii) a redução do custo médio da colisão, (iii) a redução do número total de raios testados, ou ainda, (iv) aqueles que substituem raios individuais por entidades gerais, assim como é mostrado na Figura 3.1. Embora antiga, esta classificação é pertinente, já que é comum que soluções mais modernas proponham modificações de técnicas conhecidas e possam ser encaixadas nas categorias propostas.

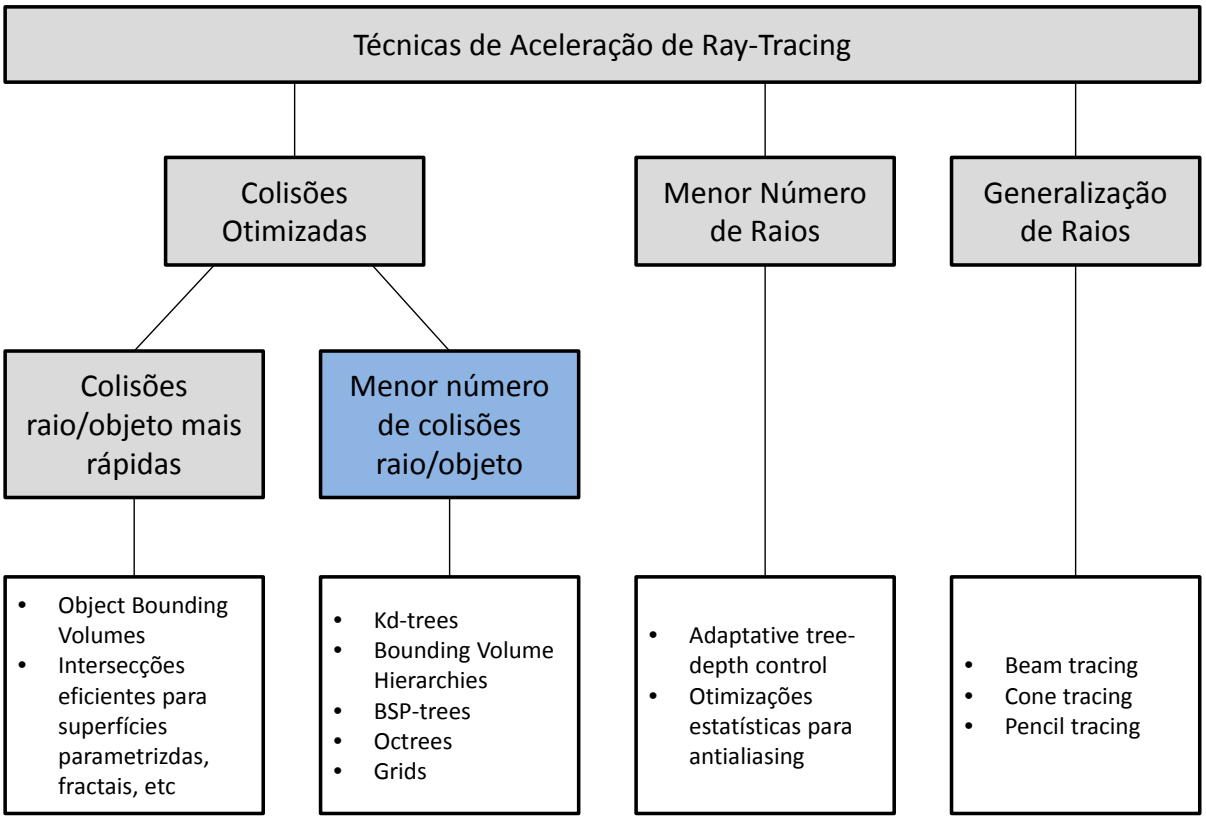

Figura 3.1 – Classificação geral das técnicas de aceleração de ray-tracing. Adaptado de Arvo (1989).

Com exceção da primeira implementação de ray-tracing (WHITTED, 1980), todos os demais métodos que buscam reduzir o custo médio das colisões adotam estruturas de dados para a subdivisão espacial da cena. Inicialmente, foram propostas estruturas como a Kd-tree (BENTLEY, 1975), a Bounding Volume Hierarchy (BHV) (RUBIN; WHITTED, 1980; KAY; KAJIYA, 1986), as BSP-trees (FUCHS; KEDEM; NAYLOR, 1980), as Octrees (GLASSNER, 1988) e as Grids (FUJIMOTO; TANAKA; IWATA, 1988). É notório que diversos trabalhos

recentes propõem soluções baseadas nestas estruturas.

Barboza (2011) propõe uma implementação de ray-tracing que utiliza uma octree como estrutura de aceleração. Neste trabalho, são comparados dois métodos de representação da octree em GPU, através de uma hash table e de uma lista linear. Cada raio dispara a execução de uma thread independente e todos os dados são armazenados na placa gráfica. O uso do potencial paralelo do hardware mostra-se promissor, entretanto, a utilização de uma thread por raio é limitada pela capacidade de gerenciar um grande número de tarefas concorrentes, o que pode esgotar rapidamente a largura de banda de memória disponível.

Um dos maiores desafios da paralelização do ray-tracing em placas gráficas é o gerenciamento da memória decorrente da travessia da estrutura de dados utilizada. Barboza (2011) não aborda este problema em seu trabalho, mas Santos (2009) defende o uso de grades uniformes (*Uniform Grids*) como forma de reduzir o impacto desta tarefa. A *uniform grid* é uma estrutura de subdivisão espacial regular, o que torna seu processo de travessia e construção simples. Em seu trabalho, Santos (2009) discute a importância do uso de hardware paralelo na execução do ray-tracing, observando que muitos métodos são beneficiados pela arquitetura de programação das placas gráficas atuais que permitem a execução de códigos de propósito geral.

Outros trabalhos, como o de Du (DU et al., 2003), buscam paralelizar a execução do ray-tracing através de hardware paralelos não convencionais. Du (DU et al., 2003) apresenta um ray-tracer baseado nas BSP-trees (FUCHS; KEDEM; NAYLOR, 1980), o qual é diretamente mapeado para um processador com arquitetura SIMD reconfigurável. Os resultados apresentados são promissores, mas a necessidade de uma tecnologia proprietária pouco difundida limita a sua aplicabilidade.

Técnicas híbridas também demonstram bons resultados. Klimaszewski (1997) introduz uma abordagem híbrida para a subdivisão do espaço através de grids adaptativas. É discutido o balanceamento entre refinamento e velocidade, de modo que grids regulares são aninhadas hierarquicamente. Segundo o autor, o método é de 5 a 50 vezes superior ao de grids regulares para cenas compactas, e ainda melhor para cenas com distribuições altamente irregulares de objetos.

Reshetov (2005) propõe o uso de uma estrutura hierárquica para a representação de uma coleção de raios (*hierarchical beam structure*). Durante a travessia de uma kd-tree, esta coleção de raios é testada a fim de descartar regiões que garantidamente não colidam com nenhum raio do conjunto. Isto permite que operações de travessia em nodos superiores sejam descartadas.

Como apresentado no Capítulo 2, o algoritmo DACRT de Mora (MORA, 2011), assim como o proposto neste trabalho, não utiliza nenhuma estrutura de dados para a subdivisão espacial da cena, e por esta razão, não pode ser diretamente comparado com os métodos anteriormente descritos. Entretanto, há diversos trabalhos recentes que possuem objetivos similares, incluindo o controle e a redução de memória e a renderização em contexto dinâmicos.

Keller e Waechter (2007) mostram como modificar os critérios de construção de uma kd-tree para permitir que as subdivisões espaciais ajustem-se a um bloco de memória restrito. Segundo os autores, o gerenciamento de memória simplificado torna o algoritmo mais rápido, eficiente e abrangente, uma vez que é possível o balanceamento automático entre o tempo de construção e de ray-tracing. Keller e Waechter também propuseram a patente de um método de ray-tracing sem a utilização de nenhuma estrutura de dados para aceleração (KELLER; WAECHTER, 2009).

A patente arquivada por Keller e Waechter (2009) baseia-se na abordagem de divisão e conquista. A cada passo da travessia é primeiramente criada a caixa delimitadora das primitivas correntes. Em seguida, os raios que colidem com esta caixa delimitadora são computados. As primitivas são divididas em dois conjuntos conforme o ponto de divisão escolhido. Desta maneira, o algoritmo é recursivamente chamado para os raios ativos e a nova divisão. Caso o número de primitivas estiver abaixo de um certo limiar, é realizado o teste de colisão dos raios ativos. Infelizmente, os autores não apresentam resultados de desempenho para esta abordagem, uma vez que seria esclarecedor a comparação deste método com o de Mora (2011).

Recentemente, Eisemann (2012) propôs uma estrutura de *Object Space Partitioning* (OSP) que pode ser representada de forma completamente implícita. Os limites de cada nodo são recriados em tempo de execução com o auxílio de um pré-ordenamento da geometria. Segundo o autor, o método é facilmente paralelizável e bem adequado a processadores com múltiplos núcleos. Uma das limitações deste método, entretanto, é a exigência da mediana do objeto como ponto de corte do esquema de particionamento, técnica que, segundo o autor, é inferior a outras alternativas.

Outros trabalhos buscam aprimorar o algoritmo DACRT de Mora (MORA, 2011). Como já discutido anteriormente, o processo de subdivisão do DACRT leva em consideração somente a distribuição das primitivas, o que pode não reduzir o número de raios nos subproblemas. Assim, Nabata (2013) propõe o traço de subconjuntos de raios para detectar a razão de colisão entre as possíveis partições de primitivas. Informações da distribuição de raios e primitivas são

utilizadas em conjunto, o que difere da solução original que assume uma distribuição de raios uniforme. Embora promissor, o método de Nabata (2013) apresenta níveis de desempenho semelhantes ao de Mora (MORA, 2011), tornando os resultados inconclusivos. Configurações de hardware diferentes foram utilizadas para os experimentos.

Ravichandran e Narayanan (2013) buscaram aprimorar o DACRT através de uma abordagem paralela. Diferentemente de nosso trabalho, os autores apresentam uma solução exclusivamente orientada a GPUs que não possui um gerenciamento de memória mínimo e determinístico. É necessário o armazenamento e controle de informações auxiliares a respeito de todos os níveis de subdivisão durante cada interação do algoritmo. Além disso, como um raio pode colidir com mais de uma subdivisão em um mesmo nível, pode ser preciso duplicar o seu id. Segundo os autores, para raios incoerentes, esta duplicação pode se tornar dispendiosa, uma vez que raios e triângulos não descartados eficientemente acarretam no aumento de memória.

Outra relevante limitação do trabalho de Ravichandran e Narayanan (2013) é a queda de desempenho quando a câmera está dentro da caixa delimitadora da cena, situação devida a não utilização da técnica de término antecipado do raio (discutida na Subseção 2.2.2). Assim, há casos em que a solução, mesmo sendo executada em um hardware largamente paralelo como uma GPU, é mais lenta que a solução sequencial. Não são discutidas a eficiência e escalabilidade da solução.

4 DESENVOLVIMENTO

Em um primeiro esforço para a obtenção de uma solução paralela do paradigma apresentado no algoritmo *Divide-And-Conquer Ray-Tracing* (DACRT) de Mora (2011), podemos paralelizar as chamadas às funções recursivas, separando os subconjuntos de raios e triângulos em dois, esquerda e direita, como é mostrado no Código 4.1.

```
1  procedure ParallelAttempt(Space S, SetOfRays R, SetOfTriangles T)
2  begin
3   if R.size() < rLimit or T.size() < tLimit
4     then LeafList.push_back(R, T); // unlike NaiveRayTracing(R, T)
5   else begin
6     {E.Left, E.Right} = SubdivideSpace(E)
7
8     E.Left.SetOfRays = R ∩ E.Left;
9     E.Left.SetOfTriangles = T ∩ E.Left;
10
11    E.Right.SetOfRays = R ∩ E.Right;
12    E.Right.SetOfTriangles = T ∩ E.Right;
13
14    do in parallel
15      ParallelAttempt(E.Left, E.Left.SetOfRays, E.Left.SetOfTriangles)
16      ParallelAttempt(E.Right, E.Right.SetOfRays, E.Right.SetOfTriangles)
17    end do
18  end
19 end
```

Código 4.1 – Tentativa de solução multithread.

Esta abordagem busca criar listas de armazenamento de subconjuntos para posterior chamada ao algoritmo de ray-tracing básico (linha 4 do Código 4.1). Deste modo, ao final do processo, cada subconjunto da lista é executado por threads concorrentes.

Dada uma subdivisão, podem existir casos em que raios acabam por não colidirem com nenhum triângulo, o que gera a necessidade de serem transferidos para outras partes da cena. Por exemplo, imagine um conjunto de raios que, dado o processo de subdivisão da caixa delimitadora alinhada aos eixos (AABB) nos filhos da esquerda e direita, foi transferido para um parte pouco povoada da cena, ou seja, com baixa concentração de triângulos. Desta forma, é possível que a maioria ou grande parte dos raios deste conjunto não colida, fazendo com que estes devam ser transferidos para o irmão da subdivisão corrente. Note que este processo é recursivo.

A utilização da abordagem do Código 4.1 para a obtenção da solução paralela, exige que cada raio seja associado à cadeia de subdivisão calculada pelo algoritmo, respeitando a hierarquia criada pelas chamadas recursivas e que, posteriormente, fossem transferidos para as subdivisões mais próximas. Esta implementação requer um algoritmo de múltiplas passadas, já

que os raios precisam ser transferidos e, além disto, é perdida a característica do paradigma introduzido pelo algoritmo DACRT de possuir gerenciamento mínimo e determinístico do uso de memória, uma vez que os requisitos para o armazenamento das lista não podem ser calculados previamente.

É notório que a implementação desta abordagem apresenta um elevado custo computacional. Testes preliminares indicaram que a solução consome um excessivo tempo no processo de transferência de raios, o que para cena complexas, domina o tempo total de execução do algoritmo. Além disto, a imprevisibilidade dos requisitos de memória é tida como uma desvantagem, já que isto dificulta a implementação em hardware com quantidade de memória fixa ou limita. Desta forma, este caminho foi descartado e outras soluções que mantivessem as características do paradigma introduzido pelo algoritmo DACRT foram pesquisadas.

Note que os resultados e experimentos preliminares desta abordagem foram obtidos através da paralelização em CPUs de múltiplos núcleos, enquanto que a investigação em arquiteturas de memória ou paralelização diferentes, como nas placas gráficas, não foi explorada. Hoedt (2012) demonstra o desenvolvimento de uma abordagem semelhante para a utilização em GPUs, mas não apresenta resultados conclusivos de desempenho ou eficiência.

4.1 Formalizando o Algoritmo Paralelo

O algoritmo paralelo a ser proposto garante o gerenciamento mínimo e determinístico do uso de memória, assim como não utiliza uma estrutura de dados para a subdivisão espacial da cena. Primeiramente a cena é dividida em um número ajustável de regiões independentes e os dados, raios e triângulos, rearranjados a fim de refletir esta nova configuração. Em seguida, um processo de três etapas é executado até que cada região independente esteja completa, ou seja, o processo de ray-tracing esteja concluído. A união das regiões independentes para a formação do resultado final é obtida através da conclusão destas etapas. A escolha do número de regiões independentes é diretamente relacionada ao número de núcleos de processamento ou threads disponíveis. Como resultado, o algoritmo busca distribuir a carga de trabalho igualmente, para isto aplicando diversos processos durante a divisão, e também procura garantir a utilização do completo potencial paralelo do hardware.

Nosso algoritmo é baseado na observação de que somente raios não terminados, aqueles delimitados pelo pivô *terminatedRayPivot*, devem ser transferidos para outras subdivisões no algoritmo DACRT. Assim, caso imaginarmos que uma cena pode ser a união e combinação de

uma coleção de outras sub-cenas, é possível a implementação do paradigma divisão e conquista para a execução do ray-tracing em regiões restritas. Nesta fundamentação, o algoritmo DACRT pode ser projetado a executar em regiões delimitadas nos vetores de dados, estes globais para todas as threads, de maneira independente e concorrentemente.

O conceito de interpretar uma cena como a composição de sub-cenas exige com que, a cada chamada recursiva à função de subdivisão do espaço, sejam especificados os limites dos subconjuntos de raios de triângulos. Isto posto, a função recursiva do Código 2.1 (Capítulo 2), a qual representa o algoritmo básico do DACRT, é alterada para compreender as novas exigências de restrição do espaço, como é mostrado no Código 4.2.

```
1 procedure ParallelDacrt(Pivot rayStart, Pivot rayEnd, Pivot triangleStart,
    Pivot triangleEnd, Space S)
```

Código 4.2 – Definição da chamada à função recursiva DACRT modificada.

A restrição do espaço de execução do algoritmo é realizada através de pivôs, os quais delimitam as posições limites de memória. Além disto, é necessária uma completa refatoração de todos os algoritmos de detecção de colisão (raios, triângulos, cones e AABB) propostos no trabalho de Mora (2011), uma vez que a utilização de um vetor global de dados não deve ocasionar condições de corrida. O gerenciamento adequado dos espaços restritos garante a dispensabilidade de processos de sincronização de threads, o que caso contrário poderia acarretar em uma expressiva perda de desempenho do algoritmo.

A divisão da cena em sub-cenas representa o número de partes que devem ser processadas pelo algoritmo de ray-tracing. Assim, a divisão pode ocorrer baseada no número de threads reais do hardware, por exemplo, ou ser definida por experimento, já que em cenas altamente complexas deve ser levado em consideração o uso que a implementação fará do sistema de memória cache. O desafio é encontrar o melhor balanço entre divisões e recursos de hardware disponíveis para cada arquitetura de implementação, como CPUs ou GPUs, tornando esta uma etapa crucial para a otimização geral do algoritmo. Cada divisão irá, de fato, representar uma porção da cena e será interpretada como um problema independente pelo nosso algoritmo, como mostrado na Figura 4.1

A interpretação de porções da cena como regiões independentes exige um cuidadoso processo de composição, já que caso fossem unificadas através de um processo ingênuo, a imagem final resultante da execução do algoritmo iria apresentar uma série de falhas e artefatos. Como cada região é independentemente gerada, a corretude do algoritmo dependeria direta-

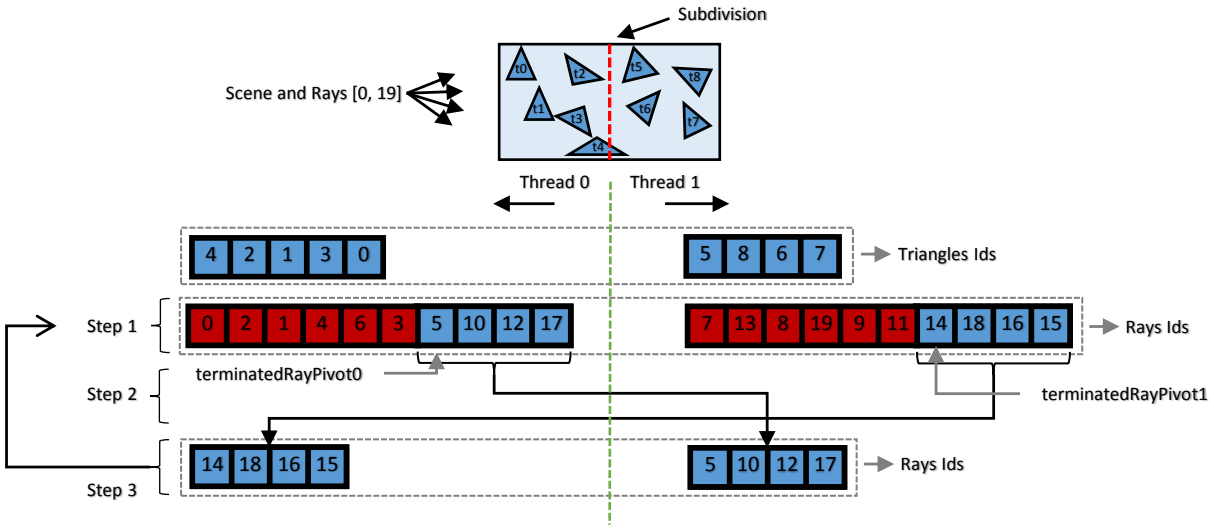

Figura 4.1 – Esquema com duas threads paralelas. Quadrados em vermelho são raios terminados delimitados pelo pivô $terminatedRayPivot_i$, local a cada região.

mente da posição inicial de emissão dos raios, ou seja, da posição da câmera. A união ingênua gera frações de vazios, uma vez que os segmentos extremos de cada região (bordas) da cena comumente dependem de raios pertencentes as regiões vizinhas, como mostrado na Figura 4.2. Deste modo, o algoritmo proposto aplica um processo consistente de **3 etapas**, como formalizado abaixo, a fim de corrigir tais falhas e, além disto, introduz um conjunto de otimizações para a lógica do método.

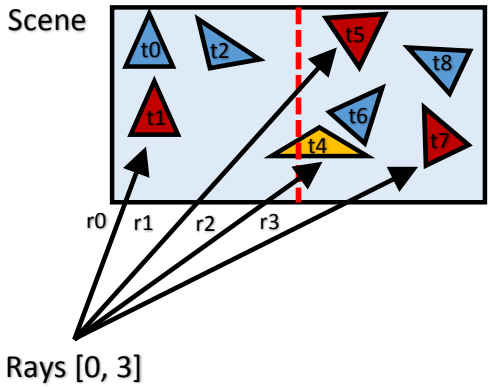

Figura 4.2 – Esquema que demonstra a dependência de raios de outras regiões em segmentos da borda. Neste caso o triângulo *t4* depende do raio *r2*, o qual é pertencente a uma região vizinha.

Após a escolha do número de regiões, cada região deve gerar conjuntos únicos dos pivôs [*rayStart*, *rayEnd*] e [*triangleStart*, *triangleEnd*], os quais demarcam as porções da cena e selecionam raios e triângulos que as colidem, utilizando a mesma lógica do DACRT de Mora (2011). O uso deste esquema de pivôs permite a execução do algoritmo em um conjunto de dados compartilhado sem a ocorrência de condições de corrida. Após a execução do

ray-tracing por cada região, devem ser demarcados os raios que colidiram com algum triângulo. Esta demarcação é realizada através do pivô *terminatedRayPivot*$_i$, único para cada região. Este processo é definido como a ***Etapa 1***, como é mostrado na Figura 4.1.

Posteriormente à execução da ***Etapa 1*** ainda pode haver a existência de raios válidos, ou seja, aqueles que não colidiram com nenhum triângulo em sua região de origem. O número de ocorrências deste caso depende diretamente de dois fatores, primeiramente do número de regiões escolhido antes da execução do algoritmo e, posteriormente, do ponto de origem dos raios. O número de regiões é relevante, pois, quanto maior o número, maior será a quantidade de bordas, estas frações críticas. Já o ponto de origem dos raios é relativo a posição e ângulo da câmera, uma vez que isto acabará por ditar a incidência de raios na cena. Como decorrência, as próximas duas etapas irão trabalhar exclusivamente com raios válidos, ou chamados de *não terminados*.

A ***Etapa 2*** é definida como a execução da transferência dos raios não terminados, resultantes da ***Etapa 1***, de uma região para outra. O processo de transferência é necessário para corrigir a dependência por raios de regiões vizinhas, como já mostrado na Figura 4.2. As transferências são realizadas através da troca dos pivôs delimitadores, de modo que o conjunto de pivôs referente aos raios [terminatedRay$_i$, rEnd$_i$] é passado para a região$_{i+1}$ e vice-versa.

A fim de garantir a independência das regiões de trabalho no conjunto de dados compartilhados, esquema utilizado pelo algoritmo, deve-se evitar que um mesmo conjunto de raios [terminatedRay$_i$, rEnd$_i$] ou conjunto de triângulos [triangleStart$_i$, triangleEnd$_i$] seja empregado em mais de uma thread. Caso isto ocorra, o desempenho do algoritmo poderá ser severamente afetado por condições de corrida.

Após o término da ***Etapa 2***, o algoritmo está apto para a execução paralela do ray-tracing em regiões que ainda apresentam falhas provenientes da dependência de raios, assim este processo é definido com a ***Etapa 3***. Note que esta etapa é semelhante à ***Etapa 1***, entretanto, além de diferir-se pelo conjunto de dados e sua interpretação lógica, comumente apresenta um tempo de execução menor, já que o número de raios envolvidos tende a ser reduzido. Assim, cada região$_i$ será formada pelos triângulos [triangleStart$_i$, triangleEnd$_i$] e os raios não terminados das regiões vizinhas. Em um cenário com duas threads de execução, são realizadas as seguintes chamadas em paralelo: Thread$_0$: ParallelDacrt (terminatedRay$_1$, rEnd$_1$, tStart$_0$, tEnd$_0$, box$_0$) e Thread$_1$: ParallelDacrt (terminatedRay$_0$, rEnd$_0$, tStart$_1$, tEnd$_1$, box$_1$), como mostrado na Figura 4.1.

4.2 Implementação Básica

O objetivo de nosso trabalho é a proposta de um algoritmo de ray-tracing paralelo baseado nos conceitos introduzidos por Mora (2011) e no paradigma de divisão e conquista, contudo, a solução apresentada também pode ser utilizada para a paralelização de diferentes técnicas além do DACRT de Mora. A paralelização de outras soluções sequenciais exige, como forma de adaptação, a simples demarcação de raios não terminados, ou seja, aqueles que, após a primeira execução de um algoritmo sequencial qualquer de ray-tracing em regiões independentes da cena, ainda não colidiram contra nenhuma primitiva. Desta forma, a fim de demonstrar e analisar uma implementação de nosso algoritmo, iremos admitir a paralelização do DACRT.

O Código 4.3 apresenta uma implementação básica de nosso algoritmo nos parâmetros anteriormente definidos. Esta implementação busca explorar o potencial paralelo de processadores com múltiplos núcleos através da divisão e execução da cena em um número ajustável de threads. Além disto, introduz as otimizações de escolha do melhor eixo de divisão, o método de subdivisão da cena, a expansão e sobreposição das caixas delimitadoras e, por fim, a criação e configuração de um pool de threads. No Código 4.4 é mostrada uma estrutura de dados utilizada no processo de subdivisão da cena. No caso da escolha de um número fixo de threads a estrutura se torna opcional, uma vez que os limites de cada porção da cena podem ser calculados previamente e inseridos no código de maneira fixa.

```
1  procedure NonPacketParallelDacrt(Space S, SetOfRays R, SetOfTriangles T,
       NOfThreads nThreads)
2  begin
3    // remove unused rays
4    [R, T] = [R, T] ∩ S;
5
6    // choosing the best subdivision axis
7    DivisionAxis axis = CalculateBestAxis(CamPosition, S);
8
9    // bounding box expansion
10   S.boundBox += 1.f;
11
12   // scene partition (rayStart/End, triangleStart/End)
13   DacrtPartition P = S;
14   do recursively until P.NumberOfSubRegions < (log(nThreads) / log(2))
15     DacrtPartition P = SubdivideSpace(P, axis);
16   end do
17
18   // bounding box overlap
19   for each DacrtPartition P do
20     P.boundBox += 1.f;
21   end do
22
23   // create thread pool work items
```

```
24    ThreadPool threads[nThreads];
25
26    // Step 1
27    for each DacrtPartition P do
28      threads.QueueWorkItem(P_i.rStart, P_i.rEnd, P_i.tStart, P_i.tEnd,
          P_i.boundBox);
29    end do
30    threads.WaitForAll();
31
32    // Step 2 and Step 3
33    // from left to right
34    for step = 1, step < nThreads, step+=1 do
35      for i = 0, i < nThreads - step, i+=1 do
36        if CameraIntersect(P_{i+step}, axis) do
37          threads.QueueWorkItem(P_{i+step}.terminatedRay, P_{i+step}.rEnd, P_i.tStart,
              P_i.tEnd, P_i.boundBox);
38        end do
39      end do
40      threads.WaitForAll();
41    end do
42
43    // from right to left
44    for step = 1, step < nThreads, step+=1 do
45      for i = nThreads - 1, i < step - 1, i-=1 do
46        if CameraIntersect(P_{i-step}, axis) do
47          threads.QueueWorkItem(P_{i-step}.terminatedRay, P_{i-step}.rEnd, P_i.tStart,
              P_i.tEnd, P_i.boundBox);
48        end do
49      end do
50      threads.WaitForAll();
51    end do
52  end
```

Código 4.3 – Pseudocódigo da solução multithread.

```
1  struct DacrtPartition
2  begin
3    Pivot terminatedRay;
4    Pivot rStart, rEnd; // rayStart and rayEnd pivots
5    Pivot tStart, tEnd; // triangleStart and triangleEnd pivots
6    AxisAlignedBoundingBox boundBox;
7  end
```

Código 4.4 – Estrutura de dados auxiliar utilizada no armazenamento das regiões independentes da cena no processo de divisão. Não é obrigatória.

Diferentemente do que foi introduzido na Seção 4.1, o processo de transferência de raios não terminados, quando da utilização de um número superior a duas divisões, deve ser executado de maneira semelhante a uma operação de deslocamento binário (*shift*). Esta exigência se dá pelo fato de que, mesmo após a passagem dos raios às regiões vizinhas, como no caso anteriormente definido, ainda podem existir ocorrências de colisões raio/triângulo não detectadas. Mais especificamente, a passagem de raios deve ocorrer até que sejam detectadas todas as

colisões raio/triângulo válidas ou que os limites da cena sejam alcançados, como mostrado na Figura 4.3.

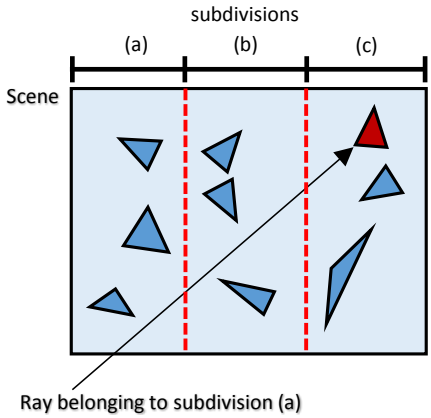

Figura 4.3 – Uma simples troca de raios entre regiões vizinhas não irá transferir os raios válidos de (a) para (c), exigindo um processo semelhante a um deslocamento binário (*shift*).

Note que a transferência de raios deve ocorrer simultaneamente em todas as regiões da cena, como mostrado no Código 4.3, o que maximiza o uso dos recursos paralelos da arquitetura de implementação.

Como decorrência da relação formada pelo vetor direção da câmera e o centro da AABB de cada região, são resultantes duas ordens de transferência de raios, como mostrado na Figura 4.4. No caso A (Figura 4.4) a câmera está posicionada em uma região externa, fazendo com que os raios sejam transferidos para o lado oposto, neste exemplo para a direita. Já no caso B (Figura 4.4) a câmera está posicionada em uma região interna, fazendo com que os raios desta região sejam transferidos para ambos os lados e os demais para somente um lado, semelhantemente ao caso A.

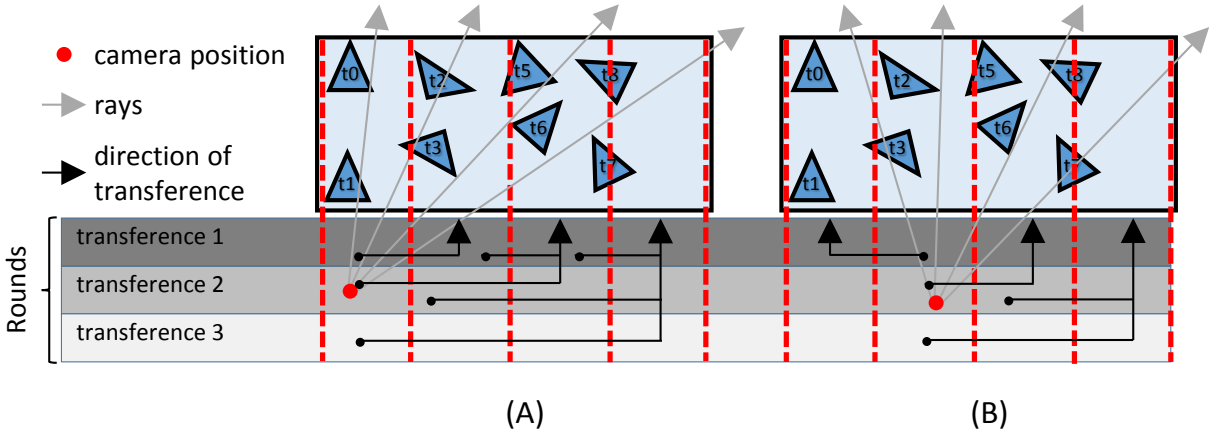

Figura 4.4 – Modelo da transferência de raios. Em (A) a câmera está em uma região externa, em (B) em uma interna.

A seleção da ordem de transferência ocorre baseada na análise individual de cada caso, como mostrado no Código 4.5, assim, as etapas 2 e 3 do nosso algoritmo são executas em dois ciclos: a passagem de raios para esquerda e a passagem para a direita. A utilização desta lógica garante a independência da execução paralela em diferentes regiões e simplifica o controle de detecção de colisões raio/triângulo válidas.

```
1  procedure CameraIntersect(DacrtPartition P, DivisionAxis axis)
2  begin
3    if camPosition[axis] <= P_i.boundBox.center[axis] do
4      return true;
5    end do
6
7    if camPosition[axis] + 1.f >= P_i.boundBox.min[axis] and camPosition[axis]
       - 1.f <= P_i.boundBox.max[axis] do
8      return true;
9    end do
10   return false;
11 end
```

Código 4.5 – Teste realizado na transferência de raios para a esquerda.

Cada ciclo de transferência é dividido por rodadas, como ilustrado na Figura 4.4. O número de rodadas depende diretamente da quantidade de divisões da cena, sendo dado por *N Divisões - 1*. Note que a troca simples ou a transferência semelhante a um deslocamento binário de raios válidos não representa um relevante aumento no custo computacional do algoritmo quando comparado ao DACRT sequencial, já que este também executa um procedimento semelhante. Em todas as rodadas o pivô $terminatedRayPivot_i$ de cada região deve ser atualizado a fim de incluir os novos raios terminados, evitando com que estes colidam novamente.

4.3 Otimizações

4.3.1 Escolhendo o Melhor Eixo de Divisão

A interpretação de uma cena como um conjunto de sub-cenas, lógica utilizada pelo algoritmo proposto, exige a escolha de um sentido de corte, o que no sistema tridimensional pode ocorrer nos eixos x, y ou z. A escolha do melhor eixo é dependente da relação entre a posição da câmera e o ponto central da cena, definindo um plano principal de intersecção.

Considerando que a cena pode ser englobada por uma caixa delimitadora alinhada aos eixos, ela possui seis lados e, consequentemente, seis faces. Sempre existe um lado de maior visibilidade. Este lado irá determinar o eixo de divisão mais apropriado.

Caso a câmera estiver contida em uma face frontal ou traseira, o melhor eixo será o x ou o y. Caso estiver contida em uma face lateral, o melhor eixo será o y ou o z. Em último caso, se a câmera estiver contida em uma face superior ou inferior, o melhor eixo será o x ou o z. Note que o correto eixo de corte evita a sobreposição de regiões quando utilizada uma projeção ortogonal, de modo que os dados da cena sejam distribuídos de maneira mais uniforme. O uso de um simples algoritmo de colisão raio/plano torna trivial a tarefa de encontrar o melhor eixo de divisão para qualquer posição da câmera.

A escolha do melhor eixo de divisão é crucial para o desempenho adequado do algoritmo, uma vez que o corte incorreto pode causar um desbalanceamento na divisão de trabalho entre as threads. O correto balanceamento ajuda a diminuir o tempo de espera para a finalização do processo de ray-tracing após a execução das etapas 2 e 3. Nestas etapas, cada ciclo de transferência de raios é realizado em rodadas, como descrito anteriormente, assim, o começo da próxima rodada depende da finalização da rodada anterior. A medida que diminui-se o tempo de espera para o início das rodadas, maximiza-se o uso do potencial paralelo da arquitetura de implementação.

4.3.2 Subdividindo o Espaço

Uma vez realizada a seleção do eixo de corte mais adequado, a próxima etapa corresponde à divisão dos dados da cena, ou seja, à criação das sub-cenas. Dada uma região chamada *Root*, são calculadas duas sub-regiões, *Left* e *Right*, assim como em uma árvore binária. Um modo simples de dividir a *Root* é a utilização de sua mediana em relação ao eixo de divisão anteriormente definido, gerando duas sub-regiões uniformes, como mostrado no Código 4.6.

```
1  procedure SubdivideSpace(Space Root, DivisionAxis Axis)
2  begin
3    DacrtPartition[Left, Right] = AverageSpaceCut(Root, Axis);
4
5    if (Length(Right.boundBox - CamPosition) < Length(Left.boundBox -
       CamPosition)) do
6      Swap(Left, Right);
7    end do
8
9    integer rPivot = SplitRays(Root.rStart, Root.rEnd, Left.boundBox);
10   integer tPivot = SplitTriangles(Root.tStart, Root.tEnd, Left.boundBox);
11
12   DacrtPartition[Left].rStart = Root.rStart;
13   DacrtPartition[Left].tStart = Root.tStart;
14   DacrtPartition[Left].rEnd = rPivot;
15   DacrtPartition[Left].tEnd   = tPivot;
16
```

```
17    DacrtPartition[Right].rStart = rPivot;
18    DacrtPartition[Right].tStart = tPivot;
19    DacrtPartition[Right].rEnd   = Root.rEnd;
20    DacrtPartition[Right].tEnd   = Root.tEnd;
21
22    return DacrtPartition[Left, Right];
23  end
```

Código 4.6 – Pseudocódigo da função *SubdivideSpace* utiliza no Código 4.3.

O uso da função *AverageSpaceCut* (Código 4.6), um método que leva em consideração somente o tamanho das regiões, pode causar um desbalanceamento na divisão de dados, uma vez que não é garantido a uniformidade de distribuição dos modelos na cena, nem levado em consideração a incidência dos raios, esta dada pela posição e ângulos da câmera. Um método mais eficiente deve analisar estes fatores e potencialmente apresentar regiões de tamanhos diferentes.

Como visto, a implementação de um método mais preciso para a subdivisão da cena é possível, entretanto, baseado em resultados obtidos através da realização de experimentos, constatou-se que o aumento da precisão exige um esforço computacional maior que a utilização da técnica simples quando da análise do tempo total de execução do algoritmo. Desta forma, optamos por utilizar a função *AverageSpaceCut*.

Nosso algoritmo utiliza o esquema de pivôs para a demarcação das regiões independentes, assim, como discutido anteriormente, os pivôs *rPivot* e *tPivot* representam, respectivamente, o ponto de separação entre duas regiões de raios e triângulos nos vetores de ids. Note que somente estes vetores são manipulados durante toda a execução do algoritmo, já que seus valores apontam para os dados reais. As funções *SplitRays* e *SplitTriangles*, utilizadas no Código 4.6, movem os objetos que pertencem a uma determinada região para o início de cada vetor de ids, retornando a última posição válida. Este processo é semelhante ao utilizado no DACRT (MORA, 2011).

O correto funcionamento das etapas 2 e 3 do algoritmo, ou seja, a definição dos pivôs [rStart, rEnd] e [tStart, tEnd], exige com que seja respeitada a ordem dada pela menor distância formada entre a câmera e o centro das regiões. Dada uma região *R*, por exemplo, que irá ser dividida em *Left* e *Right*, os pivôs da sub-região *Left* precisam conter os dados mais próximos à câmera, caso contrário, a troca *Left «» Right* é necessária. Está exigência é dada pela lógica de funcionamento das funções *SplitRays* e *SplitTriangles*, assim como demais aspectos do algoritmo.

4.3.3 Expansão e Sobreposição das Caixas Delimitadoras

O processo de subdivisão do espaço (Subseção 4.3.2) pode levar à ocorrência de artefatos nas regiões de borda das divisões. Raios que colidem em um espaço próximo às bordas podem ser impedidos a colidirem com os triângulos que normalmente fariam.

Triângulos e raios são considerados pertencentes a uma região de maneiras diferentes. Enquanto que os triângulos são separados pelo centro, os raios são separados pelo primeiro ponto de colisão com a caixa delimitadora, o que pode fazer com que o processo de transferência de raios não ocorra de maneira adequada e isto gere falhas e artefatos na imagem final, como ilustrado na Figura 4.5a e Figura 4.5b.

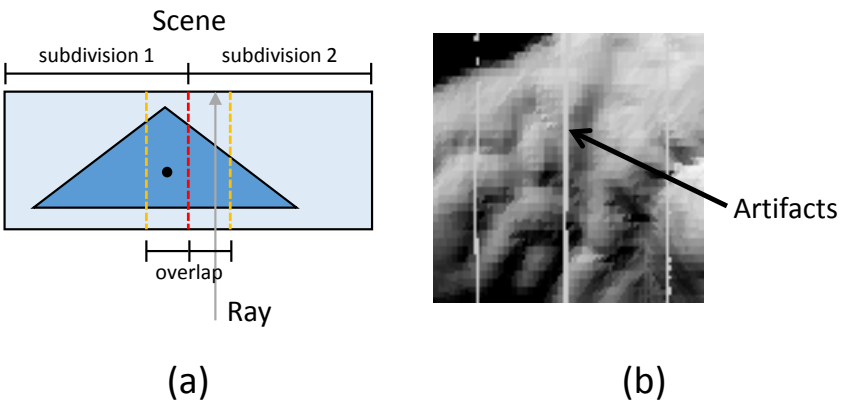

Figura 4.5 – (a) Triângulo e raio são separados em diferentes sub-regiões. (b) Artefatos na imagem final gerados por ocorrência do caso (a).

Em condições normais, a transferência de raios definida anteriormente corrige os casos de raios que devem ser passados para outras regiões, entretanto, a passagem dos raios depende da colisão com as caixas delimitadoras vizinhas. No caso da Figura 4.5a, o raio não será transferido para a primeira região, uma vez que nunca irá colidir com ela. Artefatos podem ser gerados caso haja triângulos colidindo com as seções de bordas.

A correção deste problema exige um método de expansão e sobreposição das caixas delimitadoras. Primeiramente, a caixa delimitadora que engloba toda a cena é expandida por um determinado valor em cada dimensão (Código 4.3, *bounding box expansion*). Isto irá fazer com que as bordas da cena sejam corrigidas. Apesar disso, o problema ainda ocorre nas bordas das regiões internas. Então, após a subdivisão do espaço, há o incremento das caixas delimitadoras de todas as regiões (Código 4.3, *bounding box overlap*). Este processo causa uma sobreposição múltipla, assim como é mostrado na Figura 4.5a.

4.3.4 Pool de Threads

Dado que o número de regiões pode diferir do número de núcleos de processamento da arquitetura de implementação, é necessária a utilização de um esquema que facilite a distribuição da carga de trabalho. Assim, a fim de obter um desempenho adequado nestes cenários, adotamos a utilização do padrão de projeto de software conhecido como pool de threads (*thread pool*), como é mostrado no Código 4.3

O padrão *pool de threads* consiste em threads mestres que consomem itens de trabalhos, estes normalmente organizados em uma fila, até se tornem indisponíveis. Threads mestres são permanentes entre diferentes quadros de renderização, então a sobrecarga de criação e destruição é amortizada. Além disso, este padrão previne que a troca de contexto ocasionada por um grande número de threads prejudique demasiadamente o desempenho. Itens de trabalho, por outro lado, configuram a execução das threads mestres e são as tarefas a serem realizadas.

A facilidade de distribuir threads mestres sobre os recursos do hardware é uma importante característica na implementação de um algoritmo que busca alto desempenho, como um ray-tracer. Em um processador moderno, é comum encontrar um número maior de threads virtuais do que threads físicas, desta forma, é necessária uma análise prévia do comportamento do algoritmo em diferentes configurações. Experimentos indicaram que nosso algoritmo obtém melhores resultados quando do atrelamento de uma região a um núcleo físico, entretanto, a situação pode modificar-se caso utilizado outra arquitetura ou fabricante de hardware.

Cada transferência de raios nas etapas 2 e 3 é considerada um item de trabalho, como mostrado no Código 4.3. Estas são divididas em duas configurações, transferência de raios para esquerda e para a direita, assim, a criação de itens de trabalho segue o mesmo padrão. Após cada rodada de transferência, e consequentemente de criação de itens de trabalho, o algoritmo aguarda a finalização de todas as tarefas. Pela própria natureza do esquema, as threads mestres disponíveis (em espera) irão aumentar após cada rodada. Este comportamento está presente nas duas configurações das etapas 2 e 3.

A implementação do *pool de threads* deve ser capaz de garantir a afinidade do processador, de modo que threads mestres sejam configuradas a executar em núcleos previamente designados. Esta exigência busca maximizar a eficiência do sistema de memória cache. APIs que implementam o modelo *fork/join* e, por consequência, não oferecem controle direto sobre as threads, não devem ser utilizadas. Deste modo, escolhemos utilizar a API nativa do sis-

tema operacional Windows (MICROSOFT CORPORATION, 2014), já que em experimentos demonstraram ter a menor sobrecarga quando comparada a outras alternativas.

5 RESULTADOS

Como discutido no Capítulo 4, o algoritmo proposto neste trabalho interpreta uma cena como um coleção de sub-cenas, introduzindo uma série de etapas para permitir o ray-tracing e a união de resultados independentes na formação de uma imagem final. Também foi discutido que toda solução de ray-tracing sequencial em que seja possível a demarcação de raios não terminados, ou seja, aqueles que, após o execução do ray-tracing, não colidiram com nenhuma primitiva, pode ser paralelizada utilizando nosso algoritmo. Deste modo, a fim de avaliar objetivamente a eficiência do algoritmo desenvolvido, iremos admitir a paralelização do método *Divide-And-Conquer Ray-Tracing* (DACRT) (MORA, 2011).

A escolha da paralelização do DACRT deu-se por suas características. Mora (2011) propôs e utilizou uma lógica estritamente sequencial para a concepção do algoritmo, há necessariamente somente uma linha de execução. Tal como observado no Capítulo 2, não é prática a realização de tarefas concorrentes, já que há diversas etapas que dependem da resolução de etapas anteriores, como o uso de computação local para a subdivisão recursiva da cena. Embora sequencial, o desempenho do DACRT é comparável à soluções no estado-da-arte largamente paralelas, o que torna ainda mais relevante a sua paralelização. Ademais, ambos algoritmos, o DACRT e o proposto neste trabalho, utilizam o paradigma de divisão e conquista para a aceleração de ray-tracing, compartilhando conceitos de implementação.

Neste capítulo serão apresentados resultados sob configurações e entradas diferentes, discutindo suas implicações. Comumente, em testes de algoritmos de ray-tracing, busca-se o uso de cenas já conhecidas a fim de tornar o ato de comparar diferentes soluções mais prático, evitando-se a construção ou manipulação de cenas com características atípicas. Desta forma, os experimentos realizados neste trabalho utilizam um conjunto de modelos público e bem conhecido (STANFORD COMPUTER GRAPHICS LABORATORY, 2014), como mostrado na Figura 5.1, que varia de 69 mil à 7.21 milhões de triângulos. Presume-se esta variação de modelos seja capaz de analisar a solução proposta em uma série de situações, como o impacto decorrente de diferentes níveis de esgotamento do sistema de memórias cache da arquitetura de implementação.

Os resultados das próximas seções foram conduzidos a uma resolução fixa de 640x480 pixels, em um notebook com CPU Intel Core i7-4700MQ, Turbo Boost desabilitado (todos os quatro núcleos a uma frequência de 2.4 GHz) e 8 GB de memória RAM. O processador utili-

Figura 5.1 – Modelos utilizados para os experimentos e renderizados por nosso método, nomeadamente: *Bunny*, *Armadillo*, *Dragon*, *Happy Buddha* and *Asian Dragon* (retirados do *Stanford 3D Scanning Repository* (STANFORD COMPUTER GRAPHICS LABORATORY, 2014)).

zado possui três níveis de memória cache, o último, de 6 MB, é compartilhado entre todos os núcleos (TIAN; SHIH, 2012). A implementação do algoritmo DACRT sequencial não dispõe da técnica de empacotamento de raios em cones discutida no Capítulo 2, uma vez que sua utilização pode distorcer a eficiência da união das sub-cenas independentes pelo algoritmo proposto, dificultando as análises de desempenho. Em situações reais, não há impedimentos para o uso da técnica.

5.1 Eficiência do Algoritmo de Paralelização

Em nossos experimentos, todos os modelos utilizados são considerados como cenas dinâmicas, isto é, cada frame de renderização é obtido sem nenhum conhecimento prévio e as estruturas de armazenamento, como os índices de ids de raios e triângulos, são resetadas. Como discutido no Capítulo 4, a posição e ângulos da câmera ditam a incidência de raios na cena, aspecto este relevante para o algoritmo, já que o número de rodadas de transferência de raios depende disto. Assim, os experimentos foram realizados com a câmera em movimentos circulares em torno do eixo y.

O algoritmo DACRT sequencial, propriamente implementado baseado no trabalho de Mora (2011), foi configurado para traçar somente raios primários, uma vez que buscou-se medir

diretamente a eficiência do algoritmo proposto. O traço de raios secundários e randômicos (path-tracing) não altera a lógica ou funcionamento de nosso algoritmo, desde que raios não terminados sejam identificados.

A Tabela 5.1 mostra o desempenho das abordagens sequencial e paralela com a utilização de quatro configurações de threads diferentes. A abordagem sequencial é a execução isolada do DACRT, enquanto que os demais resultados são a execução do DACRT utilizando o algoritmo de divisão da cena em sub-cenas proposto neste trabalho.

Tabela 5.1 – Comparação de desempenho entre as abordagens sequencial e paralela.

Cena	No. de Triângulos	Melhor Eixo de Divisão	Threads/Speedup			
			DACRT Sequencial	2	4	4+4
Bunny	69,451	Y	1x	1.49x	2.42x	2.42x
Armadillo	345,944	X	1x	1.52x	1.98x	2.26x
Dragon	871,414	X	1x	1.21x	1.84x	1.96x
Happy Buddha	1,087,716	Y	1x	1.29x	1.81x	2.01x
Asian Dragon	7,219,045	X	1x	1.08x	1.52x	1.68x

Primeiramente, testados todos os modelos no DACRT sequencial, os resultados obtidos foram admitidos como patamares para a comparação de desempenho. Assim, embora modelos mais complexos apresentem tempos médios de conclusão maiores, as comparações se dão pelo ganho que o algoritmo proposto é capaz de proporcionar. Posteriormente, foram executados testes com as seguintes configurações de hardware: duas threads, dois núcleos; quatro threads, quatro núcleos; e por fim, oito threads, quatro núcleos (representado na Tabela 5.1 por 4+4). Note que a última configuração faz uso de duas threads por núcleo de processamento, chamadas threads virtuais, as quais não possuem o mesmo desempenho que as threads físicas (INTEL CORPORATION, 2003, 2015).

Nos cenários testados, cada thread representa uma divisão da cena, ou seja, uma sub-cena processada independentemente, assim, o algoritmo proposto obteve um resultado de até 2.42 vezes o desempenho do DACRT executado isoladamente. A eficiência atingida por nosso algoritmo é estritamente dependente do comportamento do algoritmo de ray-tracing sequencial a ser paralelizado. Algoritmos que utilizam representações compactas dos raios, primitivas ou estruturas de dados, tendem a armazenar um conjunto de dados maior no sistema de memórias cache, o que para a tecnologia atual de processadores com múltiplos núcleos é crítica (TIAN;

SHIH, 2012).

Como observado anteriormente, nosso algoritmo é altamente dependente dos limites da largura de banda, uma vez que o ray-tracing de raios transferidos (etapas 2 e 3) será mais rápido se estes já se encontrarem na memória de acesso imediato do processador. À medida que o tamanho dos modelos do experimento aumentam, de 69 mil para 7.21 milhões de triângulos, o que representa um acréscimo de duas ordens de magnitude, a eficiência do algoritmo tende a diminuir, como é mostrado na Figura 5.2.

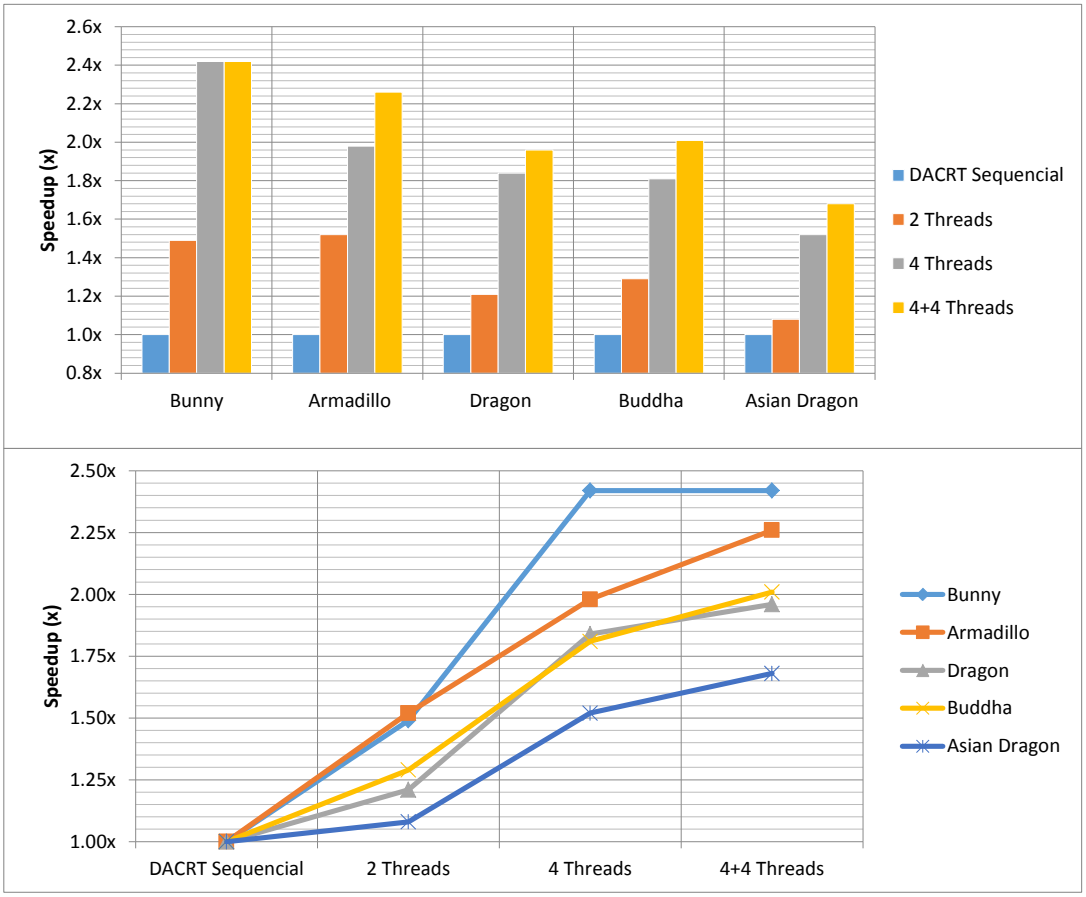

Figura 5.2 – Comparativo de speedup apresentado para cada modelo.

Analisando os modelos *Bunny* e *Armadillo* pode-se notar que, embora o modelo *Armadillo* possua um tamanho 4.9 vezes maior, o ganho de desempenho quando executados com duas threads é praticamente idêntico. Entretanto, quando executados com um número maior de threads, o modelo *Bunny* atinge desempenhos semelhantes com a utilização de quatro ou oito threads, ao passo que o *Armadillo* atinge o desempenho máximo com o uso de oito threads, comportando-se linearmente no decorrer do experimento. O desempenho no modelo *Bunny* pode ser explicado pela tendência natural de esgotamento da largura de banda quando um número maior de threads compartilha o mesmo canal de memória, mas esta análise é complexa e

depende de atributos específicos do hardware.

Os modelos *Dragon* e *Happy Buddha* possuem tamanhos similares e, consequentemente, ganhos de desempenho próximos. A paralelização da execução do modelo *Asian Dragon* apresentou a pior eficiência obtida pelo algoritmo proposto, o que é decorrente de possuir o maior tamanho dentre todos os modelos do experimento, cerca de 7 vezes maior que o tamanho do *Happy Buddha*. Caso aplicado à situações reais, espera-se que o algoritmo proposto apresente níveis de eficiência semelhantes ao obtido no modelo *Armadillo*, uma vez que cenas de jogos 3D modernos podem ser compostas pela combinação de vários personagens com cerca de 40 mil polígonos cada (VALIEN, 2013).

Além disto, se analisarmos em conjunto todos os modelos testados no experimento, observaremos que, com a exceção do modelo *Bunny*, há ganho de desempenho quando o algoritmo proposto é executado em um número de threads virtuais superior ao de núcleos de processamento. A CPU utilizada nos testes possui a habilidade de fazer com que um núcleo físico seja visto como dois núcleos virtuais em nível de software, assim posto, o uso desta característica, além de trazer outros benefícios, otimiza a utilização do sistema de memórias cache e possibilita a execução paralela de algumas instruções menos complexas (MARR et al., 2002; BONONI et al., 2006), ambos fatores críticos para o desempenho do algoritmo.

5.2 A Importância da Escolha do Melhor Eixo de Divisão

Na Subseção 4.3.1 do Capítulo 4, foi discutida a importância da correta seleção do eixo de divisão da cena, a qual acaba por implicar na distribuição de carga de trabalho entre as diversas threads. Idealmente, deve-se buscar uma distribuição completamente uniforme, já que assim o tempo de espera entre as rodadas de transferência de raios é minimizado. Os resultados da Tabela 5.1 destacam os melhores eixos de divisão quando a câmera está posicionada na face frontal de cada modelo, embora esta seleção seja realizada automaticamente pelo algoritmo naqueles experimentos.

Os resultados apresentados na Figura 5.3, diferentemente dos anteriores, fixa a câmera na face frontal de cada modelo e compara os ganhos de desempenho quando utilizados os eixos x e y. Note que, nesta situação, somente estes dois eixos podem ser candidatos à seleção, uma vez que o eixo z sempre irá apresentar os piores resultados. Isto ocorre em função de que, se escolhido o eixo z, a divisão frontal, ou seja, a sub-cena mais próxima à câmera, irá conter todos os raios da cena, exigindo um processamento mais custoso nas etapas de transferências de raios

e subutilizando o potencial paralelo do hardware.

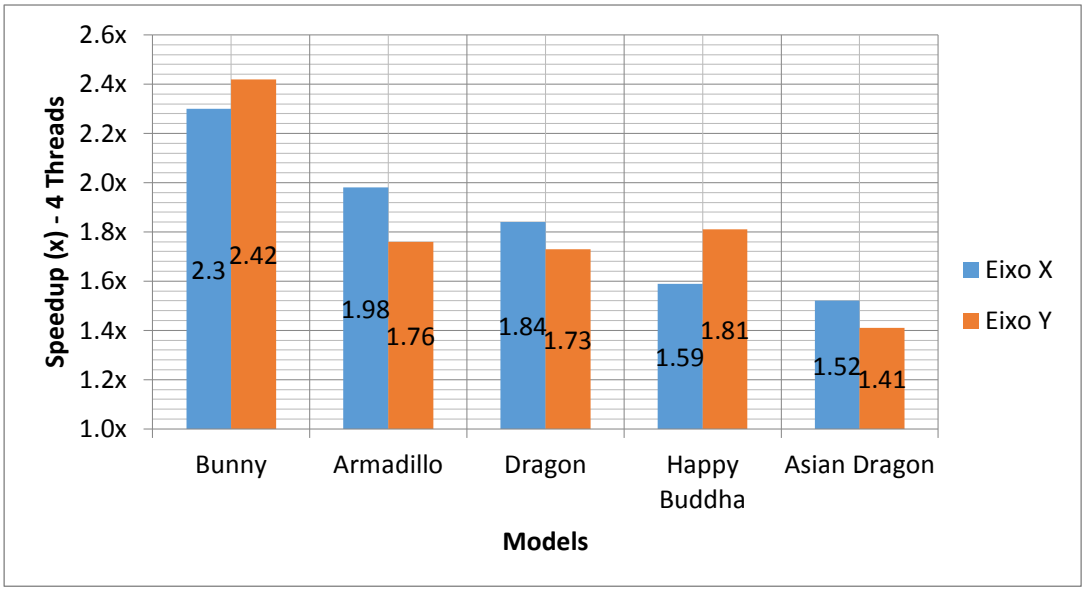

Figura 5.3 – Speedup entre diferentes escolhas de eixos de divisão.

Deste modo, a seleção do eixo de divisão mais apropriado traz resultados até 14% melhores em cenários com quatro threads. Quando é comparada execução do DACRT isolado e com o uso do nosso algoritmo, a escolha do eixo correto para o modelo *Happy Buddha* gera um ganho de desempenho que aumenta de 1.59 vezes para 1.81 vezes. Esta diferença dá-se pela assimetria dos modelos, uma vez que regiões altamente densas, como por exemplo, a parte superior do *Armadillo*, prejudicam a uniformidade de distribuição da carga de trabalho.

6 CONCLUSÃO

Este trabalho teve como objetivo principal a concepção de um método de ray-tracing paralelo baseado na utilização do paradigma de divisão e conquista. Para este fim, foi realizada uma extensa pesquisa sobre soluções de aceleração de ray-tracing e o algoritmo de *Divide-and-Conquer Ray-Tracing* (DACRT) (MORA, 2011). Como já discutido no Capítulo 2, o DACRT foi o primeiro algoritmo desde Whitted (1980) a não utilizar nenhuma estrutura de dados para realizar a subdivisão espacial da cena, característica esta que representa uma relevante vantagem quando analisada sob o aspecto de engenharia de software. Deste modo, a abordagem proposta visou manter o gerenciamento mínimo e determinístico de memória, assim como possibilitar a paralelização do DACRT e, adicionalmente, outros algoritmos sequenciais.

Os resultados obtidos demonstram o potencial da solução desenvolvida. Quando executado com o esquema paralelo proposto, o DACRT apresentou taxas de quadros por segundo até 2.42 vezes maiores que quando executado isoladamente. Além disso, a paralelização de outros algoritmos sequencias de ray-tracing é simples, exigindo somente a demarcação de raios não terminados em cada sub-cena. O esquema é adaptável a um número arbitrário de threads, facilitando a exploração do hardware paralelo.

Pesquisas recentes, como em (BORKAR, 2007) e (HILL; MARTY, 2008), têm demonstrado que, atualmente, o principal meio para o ganho de desempenho em CPUs é através do aumento de núcleos físicos em um mesmo empacotamento. Desta forma, este trabalho apresentou um algoritmo para ray-tracing que vai ao encontro desta tendência, mostrando-se mais adequado do que soluções sequenciais.

Deste modo, a principal contribuição deste trabalho foi a proposta de que uma cena pode ser interpretada como uma coleção de sub-cenas e que, esta interpretação, possibilita a execução de algoritmos sequencias de ray-tracing em regiões independentes de modo paralelo. O método apresentado gerencia o ray-tracing dessas sub-cenas e aplica um conjunto de etapas para tornar possível a formação da imagem final através da união dos resultados avulsos.

6.1 Limitações e Trabalhos Futuros

Apesar deste trabalho apresentar resultados satisfatórios quando visto sobre o aspecto da paralelização de algoritmos sequencias de ray-tracing, torna-se relevante a busca por técnicas que aumentem a eficácia paralela da solução. Resultados mostraram que o algoritmo é capaz

de obter um ganho de até 2.42 vezes quando executado em uma configuração com quatro núcleos de processamento. Embora esta eficácia também dependa do tamanho das representações de raios e triângulos na memória, os processos de escolha do melhor eixo de divisão (Subseção 4.3.1) e subdivisão do espaço (Subseção 4.3.2) desempenham fatores de maior impacto. Deste modo, estes são os principais aspectos a serem melhorados em trabalhos futuros.

Na atual implementação, a escolha do melhor eixo de divisão é fixa. Uma vez analisada a posição da câmera em relação à face de maior visibilidade da cena, é selecionado o eixo que maximiza a uniformidade na distribuição dos dados, sendo este utilizado para todas as divisões seguintes. A análise individual de cada uma dessas divisões pode trazer um aumento de eficiência, uma vez que, definido o lado de maior visibilidade, sempre existem dois eixos candidatos. Caso a câmera esteja posicionada na área de visibilidade do lado frontal, por exemplo, os eixos x e y devem ser considerados.

Outro fator relevante para a eficácia do algoritmo é o método utilizado na subdivisão do espaço. Neste trabalho, a geração de cada sub-cena é baseada na divisão da cena no ponto médio do eixo de divisão escolhido, levando-se em consideração somente o tamanho da região. Buscou-se a implementação de métodos mais precisos, os quais analisavam a distribuição de triângulos e incidência de raios, entretanto, não foram atingidos ganhos de desempenho satisfatórios. Como trabalho futuro, pode-se pesquisar a inclusão de técnicas que levem à criação de sub-cenas de tamanhos desiguais, diminuindo-se a diferença das cargas de trabalho entre as threads de execução.

As implementações precisas da escolha dinâmica do melhor eixo de divisão e do processo de subdivisão da cena são promissoras, posto que estas etapas irão ser executados somente uma vez por renderização. De maneira geral, deve-se buscar o equilíbrio entre precisão e velocidade para atingir uma escalabilidade mais eficiente a medida que o número de raios e triângulos cresce.

Outra possibilidade para a realização de trabalhos futuros, é a investigação de métodos como o de Nabata (2013), o qual propõe um algoritmo semelhante ao DACRT de Mora (2011), mas explora a distribuição de raios para construir uma estrutura de dados e deriva uma nova métrica que busca evitar a subdivisão ineficiente quando o número de raios não é suficientemente reduzido. A combinação de nosso algoritmo paralelo em conjunto com o método de Nabata (2013) demonstra-se promissor, já que ambas as técnicas de aceleração de ray-tracing independem uma da outra.

6.1.0.0.1 Implementação em GPUs

O método proposto neste trabalho permite um alto grau de paralelismo, no entanto, não foi examinada a implementação em arquiteturas largamente paralelas, como GPUs ou clusters. Recentemente, foi proposto um algoritmo semelhante ao DACRT de Mora (2011), chamado de *Parallel Divide and Conquer Ray Tracing* (RAVICHANDRAN; NARAYANAN, 2013), que executa exclusivamente em GPUs. Este algoritmo explora o uso de primitivas básicas como a ordenação e a redução, mas difere substancialmente de nosso método de paralelização.

Assim, como trabalho futuro, pode-se investigar a combinação dos métodos para a concepção de um algoritmo paralelo de ray-tracing mais eficiente, o que, juntamente com o aumento do poder de processamento do hardware, poderia possibilitar o ray-tracing em tempo real de cenas em alta resolução.

REFERÊNCIAS

AKENINE-MöLLER, T. Fast 3D triangle-box overlap testing. In: ACM SIGGRAPH 2005 COURSES, New York, NY, USA. **Anais...** ACM, 2005. (SIGGRAPH '05).

AMANATIDES, J. Ray Tracing with Cones. **SIGGRAPH Comput. Graph.**, New York, NY, USA, v.18, n.3, p.129–135, Jan. 1984.

ARVO, J.; KIRK, D. An introduction to ray tracing. In: GLASSNER, A. S. (Ed.). . London, UK, UK: Academic Press Ltd., 1989. p.201–262.

BARBOZA, D.; CLUA, E. A GPU-Based Data Structure for a Parallel Ray Tracing Illumination Algorithm. In: SBGAMES 2011. **Proceedings...** SBC, 2011.

BENTLEY, J. L. Multidimensional Binary Search Trees Used for Associative Searching. **Commun. ACM**, New York, NY, USA, v.18, n.9, p.509–517, Sept. 1975.

BONONI, L. et al. Exploring the Effects of Hyper-Threading on Parallel Simulation. **Proceedings Of The 10-Th Acm/Ieee International Symposium On Distributed Simulation And Real Time Applications**, New York, NY, USA, 2006.

BORKAR, S. Thousand Core Chips: a technology perspective. In: ANNUAL DESIGN AUTOMATION CONFERENCE, 44., New York, NY, USA. **Proceedings...** ACM, 2007. p.746–749. (DAC '07).

BRöNNIMANN, H.; GLISSE, M. Octrees with Near Optimal Cost for Ray-shooting. **Comput. Geom. Theory Appl.**, Amsterdam, The Netherlands, The Netherlands, v.34, n.3, p.182–194, July 2006.

CARR, N. A. et al. Fast GPU Ray Tracing of Dynamic Meshes Using Geometry Images. In: GRAPHICS INTERFACE 2006, Toronto, Ont., Canada, Canada. **Proceedings...** Canadian Information Processing Society, 2006. p.203–209. (GI '06).

CHOI, B. et al. Parallel SAH k-D Tree Construction. In: CONFERENCE ON HIGH PERFORMANCE GRAPHICS, Aire-la-Ville, Switzerland, Switzerland. **Proceedings...** Eurographics Association, 2010. p.77–86. (HPG '10).

COMBA, J. L. D.; PERSIANO, R. C. M. Ray Tracing of CSG Solids using a Multi-Resolution Bresenham. In: COMPUGRAPHICS 91, VOLUME 2. **Proceedings...** [S.l.: s.n.], 1991. p.210–219. Sesimbra, Portugal. September 1991.

DU, H. et al. Interactive ray tracing on reconfigurable SIMD MorphoSys. In: ASIA AND SOUTH PACIFIC DESIGN AUTOMATION CONFERENCE, 2003., New York, NY, USA. **Proceedings...** ACM, 2003. p.471–476. (ASP-DAC '03).

EBERLY, D. **Intersection of a line and a cone**. Online. Acessado em 25 de Julho de 2013, http://www.geometrictools.com/Documentation/IntersectionLineCone.pdf.

EISEMANN, M. et al. Geometry Presorting for Implicit Object Space Partitioning. **Comp. Graph. Forum**, New York, NY, USA, v.31, n.4, p.1445–1454, June 2012.

FRIEDRICH, H. et al. Exploring the use of ray tracing for future games. In: ACM SIGGRAPH SYMPOSIUM ON VIDEOGAMES, 2006., New York, NY, USA. **Proceedings...** ACM, 2006. p.41–50. (Sandbox '06).

FUCHS, H.; KEDEM, Z. M.; NAYLOR, B. F. On Visible Surface Generation by a Priori Tree Structures. In: ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, 7., New York, NY, USA. **Proceedings...** ACM, 1980. p.124–133. (SIGGRAPH '80).

FUJIMOTO, A.; TANAKA, T.; IWATA, K. Tutorial: computer graphics; image synthesis. In: JOY, K. I. et al. (Ed.). . New York, NY, USA: Computer Science Press, Inc., 1988. p.148–159.

GLASSNER, A. S. Tutorial: computer graphics; image synthesis. In: JOY, K. I. et al. (Ed.). . New York, NY, USA: Computer Science Press, Inc., 1988. p.160–167.

GOLDSMITH, J.; SALMON, J. Automatic Creation of Object Hierarchies for Ray Tracing. **IEEE Comput. Graph. Appl.**, Los Alamitos, CA, USA, v.7, n.5, p.14–20, May 1987.

HECKBERT, P. S.; HANRAHAN, P. Beam tracing polygonal objects. **SIGGRAPH Comput. Graph.**, New York, NY, USA, v.18, n.3, p.119–127, Jan. 1984.

HILL, M. D.; MARTY, M. R. Amdahl's Law in the Multicore Era. **Computer**, Los Alamitos, CA, USA, v.41, n.7, p.33–38, 2008.

HOEDT, A. **Divide-and-Conquer Ray Tracing on the GPU**. Online. Accessado em 4 de Julho de 2013, http://asgerhoedt.dk/?page_id=240.

INTEL CORPORATION. **Intel Hyper-Threading Technology Technical - User's Guide**. Online. Acessado em 10 de Janeiro de 2015, https://www.utdallas.edu/~edsha/parallel/2010S/Intel-HyperThreads.pdf.

INTEL CORPORATION. **Get faster performance for many demanding business applications**. Online. Acessado em 10 de Janeiro de 2015, http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html.

JOY, K. I.; BHETANABHOTLA, M. N. Ray tracing parametric surface patches utilizing numerical techniques and ray coherence. **SIGGRAPH Comput. Graph.**, New York, NY, USA, v.20, n.4, p.279–285, Aug. 1986.

KAY, T. L.; KAJIYA, J. T. Ray tracing complex scenes. **SIGGRAPH Comput. Graph.**, New York, NY, USA, v.20, n.4, p.269–278, Aug. 1986.

KELLER, A.; WAECHTER, C. **Efficient Ray Tracing Without Acceleration Data Structure**. US 2009/0225081 A1, Patent Application.

KLIMASZEWSKI, K. S.; SEDERBERG, T. W. Faster Ray Tracing Using Adaptive Grids. **IEEE Comput. Graph. Appl.**, Los Alamitos, CA, USA, v.17, n.1, p.42–51, Jan. 1997.

KOK, A. J. F.; JANSEN, F. W. Adaptive Sampling of Area Light Sources in Ray Tracing Including Diffuse Interreflection. **Computer Graphics Forum**, [S.l.], v.11, n.3, p.289–298, 1992.

LAGAE, A.; DUTRÉ, P. Compact, Fast and Robust Grids for Ray Tracing. In: ACM SIGGRAPH 2008 TALKS, New York, NY, USA. **Anais...** ACM, 2008. p.20:1–20:1. (SIGGRAPH '08).

LAGAE, A.; DUTRÉ, P. **Accelerating Ray Tracing using Constrained Tetrahedralizations**. 2008. 1p.

LEE, M. E.; REDNER, R. A.; USELTON, S. P. Statistically Optimized Sampling for Distributed Ray Tracing. In: ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTE-

RACTIVE TECHNIQUES, 12., New York, NY, USA. **Proceedings...** ACM, 1985. p.61–68. (SIGGRAPH '85).

MACDONALD, D. J.; BOOTH, K. S. Heuristics for ray tracing using space subdivision. **Vis. Comput.**, Secaucus, NJ, USA, v.6, n.3, p.153–166, May 1990.

MARR, D. T. et al. Hyper-Threading Technology Architecture and Microarchitecture. **Intel Technology Journal**, [S.l.], v.6, n.1, p.4–15, Feb. 2002.

MICROSOFT CORPORATION. **Thread Pool API (Windows)**. Online. Acessado em 04 de Março de 2014, http://msdn.microsoft.com/en-us/library/windows/desktop/ms686766%28v=vs.85%29.aspx.

MORA, B. Naive ray-tracing: a divide-and-conquer approach. **ACM Trans. Graph.**, New York, NY, USA, v.30, n.5, p.117:1–117:12, Oct. 2011.

NABATA, K. et al. Efficient Divide-and-conquer Ray Tracing Using Ray Sampling. In: HIGH-PERFORMANCE GRAPHICS CONFERENCE, 5., New York, NY, USA. **Proceedings...** ACM, 2013. p.129–135. (HPG '13).

RAVICHANDRAN, S.; NARAYANAN, P. J. Parallel Divide and Conquer Ray Tracing. In: SIGGRAPH ASIA 2013 TECHNICAL BRIEFS, New York, NY, USA. **Anais...** ACM, 2013. p.30:1–30:4. (SA '13).

RESHETOV, A.; SOUPIKOV, A.; HURLEY, J. Multi-level Ray Tracing Algorithm. In: ACM SIGGRAPH 2005 PAPERS, New York, NY, USA. **Anais...** ACM, 2005. p.1176–1185. (SIGGRAPH '05).

RUBIN, S. M.; WHITTED, T. A 3-dimensional Representation for Fast Rendering of Complex Scenes. In: ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, 7., New York, NY, USA. **Proceedings...** ACM, 1980. p.110–116. (SIGGRAPH '80).

SANTOS, P. **Ray Tracing Dynamic Scenes on the GPU**. 2009. Dissertação (Mestrado em Ciência da Computação) — Pontífica Universidade Católica do Rio de Janeiro.

SHEVTSOV, M.; SOUPIKOV, A.; KAPUSTIN, A. Highly Parallel Fast KD-tree Construction for Interactive Ray Tracing of Dynamic Scenes. **Computer Graphics Forum**, [S.l.], v.26, n.3, p.395–404, 2007.

SHINYA, M.; TAKAHASHI, T.; NAITO, S. Principles and applications of pencil tracing. **SIGGRAPH Comput. Graph.**, New York, NY, USA, v.21, n.4, p.45–54, Aug. 1987.

STANFORD COMPUTER GRAPHICS LABORATORY. **The Stanford 3D Scanning Repository**. Online. Acessado em 04 de Março de 2014, http://www-graphics.stanford.edu/data/3Dscanrep/.

TIAN, T.; SHIH, C.-P. **Software Techniques for Shared-Cache Multi-Core Systems**. Online. Acessado em 20 de Dezembro de 2014, https://software.intel.com/en-us/articles/software-techniques-for-shared-cache-multi-core-systems/?wapkw=smart+cache.

VALIEN, M. **Killzone Shadow Fall Demo Postmortem**. Sony Devcon 2013. Acessado em 20 de Dezembro de 2014, http://www.guerrilla-games.com/presentations/Valient_Killzone_Shadow_Fall_Demo_Postmortem.pdf.

WäCHTER, C.; KELLER, A. Instant Ray Tracing: the bounding interval hierarchy. In: EUROGRAPHICS CONFERENCE ON RENDERING TECHNIQUES, 17., Aire-la-Ville, Switzerland, Switzerland. **Proceedings...** Eurographics Association, 2006. p.139–149. (EGSR'06).

WACHTER, C.; KELLER, A. Terminating Spatial Hierarchies by A Priori Bounding Memory. In: INTERACTIVE RAY TRACING, 2007. RT '07. IEEE SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2007. p.41–46.

WALD, I. et al. Interactive Rendering with Coherent Ray Tracing. In: COMPUTER GRAPHICS FORUM. **Anais...** [S.l.: s.n.], 2001. p.153–164.

WALD, I. et al. Ray Tracing Animated Scenes Using Coherent Grid Traversal. In: ACM SIGGRAPH 2006 PAPERS, New York, NY, USA. **Anais...** ACM, 2006. p.485–493. (SIGGRAPH '06).

WHITTED, T. An improved illumination model for shaded display. **Commun. ACM**, New York, NY, USA, v.23, n.6, p.343–349, June 1980.

WILLIAMS, A. et al. An efficient and robust ray-box intersection algorithm. In: ACM SIGGRAPH 2005 COURSES, New York, NY, USA. **Anais...** ACM, 2005. (SIGGRAPH '05).

APÊNDICES

APÊNDICE A – Artigo 1 - Improving Divide-And-Conquer Ray-Tracing Using a Parallel Approach

Artigo publicado na *27th SIBGRAPI Conference on Graphics, Patterns and Images* (SIBGRAPI 2014). Classificado como B1 no Qualis da CAPES.

Improving Divide-And-Conquer Ray-Tracing Using a Parallel Approach

Cícero Augusto de Lara Pahins, Cesar Tadeu Pozzer
Department of Applied Computing (DCOM)
Graduate Program in Informatics (PPGI)
Federal University of Santa Maria
Santa Maria, Brazil
Email: {cicerolp,pozzer}@inf.ufsm.br

Fig. 1: Models used for experiments and rendered by our algorithm, namely: Bunny, Armadillo, Dragon, Happy Buddha and Asian Dragon [1].

***Abstract*—This paper presents a new Divide-and-Conquer Ray-Tracing (DACRT) algorithm that is designed to perform on multi-core processors. This new algorithm proposes a parallel and generic scheme that, without the use of any data structure for spatial subdivision, maintains memory management minimal and deterministic. Initially, the scene is divided into sub-scenes and those uniformly distributed across available hardware resources, processing each sub-scene individually. After, an iterative step to ensure the correct results is performed until the final frame is obtained. Results show that our algorithm is up to 2.4x times faster than the original DACRT in a common quad-core processor setup, allowing very high interactive frame rates in well-known benchmark scenes.**

***Keywords*-divide-and-conquer; ray-tracing; parallelization;**

I. Introduction

Ray-Tracing (RT) is a fundamental and much investigated image synthesis algorithm that aims to produce high fidelity images. The naive RT algorithm is well known and consists in the intersection test of each ray with all primitives within the scene. However, the complexity of this algorithm (θ*(primitives x rays)*) is not suitable for most current applications that deal with high density models and large output resolutions [2]. Thus, the naive algorithm was quickly discarded for use in real time applications and many optimization methods were proposed and widely investigated, such as novel data structures [3]–[6], numerical and statistical methods [7] [8], new computational geometry techniques [9] [10], parallel solutions [11]–[13], among others.

Methods that use data structures to reduce the complexity of the naive RT algorithm by subdividing the scene (triangles and rays) in a hierarchical fashion, as spatial subdivision structures, are very common. However, the requirement of robust memory management and the difficulty in adapting it to specific hardware (e.g., GPUs), are seen as a burden by developers, often making the software engineering more complex and implying additional restrictions in the graphics pipeline.

Recently, it was presented a technique called Divide-and-Conquer Ray-Tracing (DACRT) [14] that, for the first time since the original RT algorithm [15], does not use any data structure for spatial subdivision. Furthermore, this new approach showed significant contributions as a new paradigm of RT, such as the rendering speed comparable and even faster than many state of the art solutions that use sophisticated and complex methods of optimizations and also a minimal and deterministic memory usage, creating a new field of research in the area. One of the biggest drawbacks of the method, however, is that the core algorithm of DACRT was designed in a way that it can be performed only on single-core setups, contradicting the current multi-core development of CPUs and the massive parallel nature of GPUs.

In this work we introduce a new RT algorithm that uses the fundamentals ideas proposed by [14], meaning no use of data structure for spatial subdivision and ensuring minimal and deterministic memory usage, yet well suitable to multi-core setups. To make this possible, firstly the scene is subdivided into an arbitrary number of independent regions and data (triangles and rays) rearranged to reflect this new configuration. Following, a three-step process is performed until the processing of each independent region is completed. The choice of how many independent regions will be created depends on the number of available processor cores or threads. As a result, the algorithm will seek to distribute the workload

evenly, applying various processes to ensure this and to utilize the full potential of multi-core hardwares.

The main contribution of this paper is the proposition of a new divide-and-conquer ray-tracing algorithm that exploits parallel hardware. We introduce a generic scheme that can be adapted to an arbitrary number of threads. Memory management is kept minimal and deterministic. In particular, our solution handles ray-traced sub-scenes individually by applying a set of optimizations and procedures to transform the union of these results into the final image. In our experiments, we evaluated a gain of up to 2.42x when compared to the original DACRT algorithm [14].

This article is organized as follows. Section II presents some previous works and the basic concepts of the DACRT utilized by the proposed algorithm. In Section III the proposed algorithm is formalized by describing a first implementation oriented to multi-core CPUs. In Section IV various optimizations steps, to ensure the evenly distribution of workload, are presented, and, finally, the Section V shows the results of the algorithm when applied on well-known scenes (Figure 1). A benchmark between the proposed algorithm and the original DACRT [14] is also presented.

II. Related Work

The use and adaptation of spatial subdivision structures on parallel hardware is a highly studied topic. Barboza [16] presents a data structure designed to take advantage of the high degree of parallelism of GPUs, where each thread runs only one ray. In [17] a BSP-based ray-tracing is mapped to a reconfigurable SIMD architecture, showing that ray-tracing is possible even on portable devices. Santos [18] presents a uniform grid-based GPU ray-tracer that achieves interactive frame rate in dynamic scenes.

The main drawback from these previous works is that the acceleration data structure is stored in memory. Works from Mora [14] and Keller [19] began to propose implicit acceleration structures. In [20] is presented a new data structure for object partitioning that can be represented completely implicit, in contrast to our algorithm that is based on space partitioning.

A. DACRT: The Fundamentals

The DACRT algorithm [14] is based on the observation that the complexity to find the nearest ray/triangle intersection does not necessarily imply in *θ(primitives x rays)* when proposed a divide-and-conquer approach that decreases the solution space. The method consists in a spatial subdivision of the scene through the use of pivots, which implicitly delimit regions of both rays and triangles (Figure 2). Firstly, pivots are compared with two arbitrary fixed constants that are related to the number of rays and triangles. If a comparison is true, the problem is reduced to a naive RT algorithm, otherwise the space is subdivided and a recursive call is made to the DACRT containing only rays and triangles that intersect the new space.

At the end of the naive ray-tracing there are rays that do not intersect any triangle in the current subdivision, so they should be transferred to the next subdivisions. Those that intersected

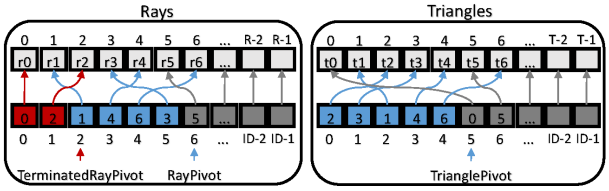

Fig. 2: Relationship between the vectors of ids (bottom) and vector of data (top). *rayPivot* and *trianglePivot* represent the current subdivision (blue squares), and must be recalculated for each recursive call to DACRT, while the *terminatedRayPivot* is global for the entire algorithm and represents the rays that intersect with a triangle (red squares).

a triangle are transferred to the region bounded by the pivot *terminatedRayPivot*, and can be ignored in future subdivisions as they are no longer part of the problem. The triangles do not need to be transferred because they are static during the whole process. In other words, they can change position only in the next frame. Scene data, rays and triangles, are stored in vectors and remain unchanged throughout the process. Thus, the algorithm uses only vectors of ids that correspond directly to the real data, making the process of handling memory much faster (Figure 2).

As the *terminatedRayPivot* is global for all calls to the DACRT and the decision of the rays that will be transferred to future subdivisions depends on the resolution of past subdivisions, the algorithm proposed by [14] can only perform in a single-core setup. In addition, other requirements for the implementation of the solution such as the linear scene data storage, in-place computations, and front-to-back transversal associated with early-ray termination are strictly related to the fact that the core of such approach will be executed by a single thread.

With the aim to benchmark the speed-up between our algorithm and the original DACRT, we develop a DACRT implementation that follows the algorithm described by [14], using the same methods for intersection tests of ray/box [21], triangle/box [22] and triangle/ray [23], and performing all the high-level optimizations proposed in that work. However, hardware dependent, low-level optimization instructions such as the use of *Streaming SIMD Extensions* (SSE) to make intersection tests faster and allow compact triangle structure representation, were not implemented.

Our algorithm was conceived to be applicable to different architectures, and so, does not utilize any specific platform instruction. This way, to make the analysis of the relative speed-up between our algorithm and the original DACRT in an equivalent and best possible form, we decided to reproduce the results obtained by Mora [14] with our own implementation.

III. A Multithreaded Solution

Our first attempt to create a multithreaded solution, focused on the parallelization of the recursive calls of the DACRT algorithm, is shown in Figure 3.

```
procedure PARALLELATTEMPT(Space S, SetOfRays R, SetOfTriangles T)
    if R.size() < rLimit or T.size() < tLimit then
        LeafList.push_back(R, T)
    else
        {E.Left, E.Right} = SubdivideSpace(E)

        E.Left.SetOfRays = R ∩ E.Left
        E.Left.SetOfTriangles = T ∩ E.Left

        E.Right.SetOfRays = R ∩ E.Right
        E.Right.SetOfTriangles = T ∩ E.Right

        do in parallel
            ParallelAttempt(E.Left, E.Left.SetOfRays, E.Left.SetOfTriangles)
            ParallelAttempt(E.Right, E.Right.SetOfRays, E.Right.SetOfTriangles)
        end do
    end if
end procedure
```

Fig. 3: A first attempt to a multithreaded solution.

In this approach the idea is to create a list that stores groups of rays and triangles and then call the naive ray-tracer. Thus, at the end of the process, each group in the list can be executed by independent threads. Note that still exist rays that do not intersect with any triangle given a subdivision. Thus, it would be necessary that each ray keeps track of the subdivision chain, respecting the hierarchy created by recursive calls. After, the recorded data would be transferred to the nearby subdivisions.

The implementation of this approach would require a multiple pass algorithm, since the rays must be transferred and, in addition, the crucial feature of the DACRT algorithm of minimum and deterministic memory usage would be lost (by creating the list of groups and associating each ray with nearby subdivisions).

Having shown tremendous overhead for its implementation, and no promising result in a first attempt, this path was quickly discarded. Hoedt [24] shows an implementation with similar idea in GPUs as proof of concept.

A. Formalizing the Multithreaded Algorithm

Our multithreaded solution is based on the observation that only unterminated rays (delimited by *terminatedRayPivot*) should be transferred, so DACRT can run independently and in parallel to restricted regions if we imagine that a scene is a composition of several sub-scenes. Thus, initially the DACRT algorithm is modified, as shown in Figure 4, to be executed in established regions (sub-scenes) in the vector of ids, as shown in Figure 5.

```
procedure NEWDACRT(Pivot rayStart, Pivot rayEnd, Pivot triangleStart, Pivot
triangleEnd, Space S)
    ...
end procedure
```

Fig. 4: Definition of the modified DACRT call.

Note that while the new call has the same logic of calling the original algorithm, it is necessary a complete refactoring in all rays, triangles and cones split algorithms of the original DACRT [14] to allow it to perform in specific regions.

The scene can be divided according to the number of threads or be defined by analyzing the triangle distribution. The key is to find the best balance between divisions and available resources for each type of hardware, such as cache size and number of physical or virtual threads, making this a critical step for the general optimization of the algorithm. Each division will, in fact, represent a part of the entire scene and is interpreted as a totally independent region by our algorithm.

The idea of treating each region of the scene (rays and triangles) as sub-scenes, or as independent regions, is based on the basic concepts of divide-and-conquer algorithms. The scene is divided in sub-scenes by the filtering process described in [14] and each of these ray traced independently with the modified DACRT. The straightforward union of all ray-traced sub-scenes gives us a very sketchy result, with many artifacts and defects in the final image, far from what would be expected. Our algorithm, thus, apply a **3 steps** process, as formalized in the following, to correct this artifacts and defects. Additionally, many optimizations on the whole process are presented.

Each division generates four pivots: *rayStart*, *rayEnd*, *triangleStart* and *triangleEnd*, which demarcate the rays and triangles colliding with a given region (sub-scene), allowing multiple instances of the modified DACRT algorithm to be executed in parallel on shared dataset (rays and triangles). Each run will result in terminated rays, delimited by $terminatedRayPivot_i$, now local to the threads. This process is defined as ***Step 1***.

After ***Step 1*** still exist valid or unterminated rays, in other words, there are rays that did not intersect any triangle in the given region. The quantity of these cases directly depends on the number of divisions chosen, defined before the execution of the algorithm, and on the point of space that originated the rays. The next two steps deals exclusively with unterminated rays.

The **Step 2** refers to the transference of unterminated rays, resulting from ***Step 1***, from a given region to another region of the scene. Note that this transference is the simple passage of pivots [terminatedRay_i, rEnd_i] to thread_{i+1} and vice-versa (in a two threads setup). One must take care to never utilize the same set of rays (terminatedRay_i, rEnd_i) or set of triangles (triangleStart_i, triangleEnd_i) in more than one subdivision, thus creating independent regions of work in a shared dataset.

Step 3 represents the parallel execution of the modified DACRT algorithm, as in ***Step 1***, where each thread_i contains the pivots [triangleStart_i, triangleEnd_i] and the unterminated rays from other regions of the scene. Figure 5 presents a rendering configuration with two threads. For this situation, two parallel calls to the modified DACRT algorithm must be made: Thread_0:NewDacrt(terminatedRay_1, rEnd_1, tStart_0, tEnd_0, box_0) and Thread_1:NewDacrt(terminatedRay_0, rEnd_0, tStart_1, tEnd_1, box_1),

B. Basic Implementation

A basic implementation of our algorithm that is adapted to multi-core processors is showed in Figure 6. Setups with an arbitrary number of physical and virtual threads are target. As mentioned earlier, a number of optimizations to the parallel scheme are implemented and discussed.

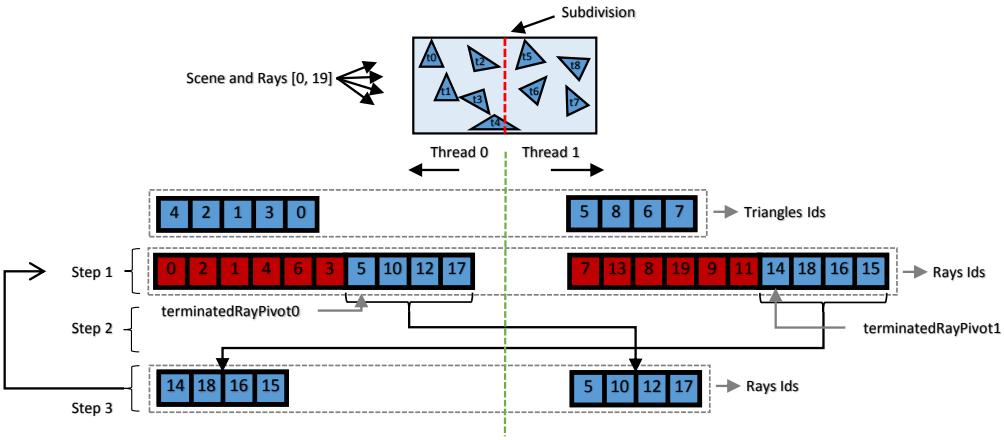

Fig. 5: Scheme with two parallel threads. Squares in red are terminated rays that are delimited by *terminatedRayPivot*$_i$, now local to each thread.

```
procedure DACRTNONPACKETPARALLEL(Space S, SetOfRays R, SetOfTrian-
gles T, NOfThreads nThreads)
    // remove unused rays
    [R, T] = [R, T] ∩ S

    // choosing the best subdivision axis
    DivisionAxis axis = CalculateBestAxis(CamPosition, S)

    // bounding box expansion
    S.boundBox += 1.f

    // scene partition (rayStart/End, triangleStart/End)
    DacrtPartition P = S
    do recursively until (P.NumberOfSubRegions < (log(nThreads) / log(2)))
        DacrtPartition P = SubdivideSpace(P, axis)
    end do

    // bounding box overlapping
    for all DacrtPartition P do
        P.boundBox += 1.f
    end for

    // create thread pool work items
    ThreadPool threads[nThreads]

    // step 1
    for all DacrtPartition P do
        threads.QueueWorkItem(P_i.rStart, P_i.rEnd, P_i.tStart, P_i.tEnd,
P_i.boundBox)
    end for
    threads.WaitForAll()

    // steps 2 and 3 // from left to right
    for step = 1, step < nThreads, step+=1 do
        for i = nThreads - 1, i < step - 1, i -=1 do
            if CameraIntersect(P_{i-step}, axis) then
                threads.QueueWorkItem(P_{i-step}.terminatedRay, P_{i-step}.rEnd,
P_i.tStart, P_i.tEnd, P_i.boundBox)
            end if
        end for
    end for

    // steps 2 and 3 // from right to left
    ...
end procedure
```

Fig. 6: Pseudocode of the multithreaded solution.

To allow an arbitrary number of threads, **Steps 2** and **3** must execute a circular transference, like a shift, of the unterminated rays between divisions of the scene. This is due to the fact that some of the unterminated rays transferred to neighboring regions still do not intersect any triangle, demanding to be transferred to all other divisions/threads, as shown in Figure 7. More specifically, in an implementation of the solution with only two threads, each thread must transfer its unterminated rays to another, performing a simple swap operation, whereas the choice of an arbitrary number of threads implies that still exist valid unterminated rays that must be transferred even after the simple swap.

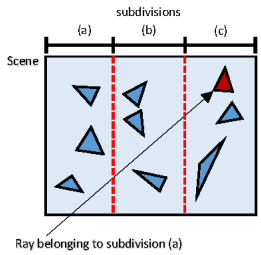

Fig. 7: A simple swap between divisions will not transfer the rays from (a) to (c).

The transference order of unterminated rays for the cases of arbitrary number of threads depends on the camera position in relation to the bounding box position of each division, as shown in Figure 8. In case A the camera is placed at an external region, and in case B at an internal region. As depicted by Figure 9, both cases are divided into two settings: **first**, the transference of rays to the left and **later**, the transference of rays to the right, selecting the applicable setting case by case.

Each transference setting is divided by rounds, as shown in Figure 8. A set of rays (pivots in the vector of ids) cannot be used by two divisions simultaneously. The number of rounds depends directly on the number of divisions and is given by $NDivisions - 1$. Note that the simple swap or the circular transference of unterminated rays does not represent an increase in cost when compared to the original DACRT

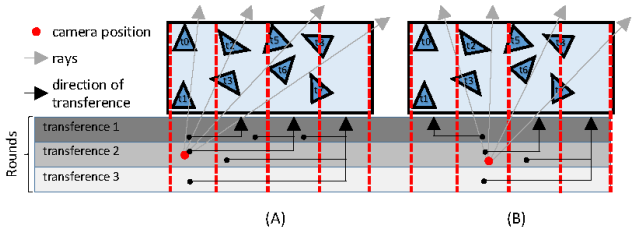

Fig. 8: Model of ray transference. In case (A) the camera is in an external division. In case (B) the camera is in an internal division.

```
procedure CAMERAINTERSECT(DacrtPartition P, DivisionAxis axis)
    if camPosition[axis] ¡= P_i.boundBox.center[axis] then
        return true
    end if

    if camPosition[axis] + 1.f >= P_i.boundBox.min[axis] and camPosition[axis]
- 1.f <= P_i.boundBox.max[axis] then
        return true
    end if
    return false
end procedure
```

Fig. 9: Test performed in ray transference to the left.

algorithm [14], as it also performs the same process, but in a single-core setup.

In every round the $terminatedRayPivot_i$ of each division should be updated to include the newly terminated rays, avoiding the intersection of terminated (invalid) rays.

IV. Optimizations

In this section we present a number of optimizations to ensure the correct and evenly distribution of the workload across an arbitrary number of threads. Note that these optimizations are not present in the original DACRT algorithm [14], thus are unique to our parallel scheme.

A. Choosing the Best Subdivision Axis

The division of the scene happens in any axis of the 3d-space. It should take into consideration, however, the vector created between the camera position and the central position of the whole scene, which will define a principal intersection plane. The use of a simple ray/plane intersection algorithm makes trivial the task of finding the principal intersection plane to all camera positions.

Considering that the scene can be enclosed by a cube, it has 6 sides and, consequently, 6 faces. The camera will always be in front of one of these sides (side with greater visibility). The selected side will determine the principal intersection plane and defines the appropriate axis of division. For instance, if the camera is contained in the front or rear side, the best division axis will be x or y. If the camera is contained in a lateral side, the best division axis will be y or z. Finally, if the camera is contained in the lower or upper side, the best division axis will be x or z.

Note that each side will always have two candidates to the best division axis. A good choice is to select the axis that represents the largest scene dimension, since this will help to better distribute the data in the next step of the algorithm (Subsection IV-B). The choice of the correct division axis is crucial to the adequate performance of the algorithm, once the use of an incorrect axis can generate a thread wait chain, causing situations where some threads (divisions) have much more work (data) than others.

B. Subdividing the Space

Once the scene division axis has been chosen, the next step is the data division. Given a region called *Root*, two subregions, *Left* and *Right*, as in a binary tree, are calculated. The simplest way to find the splitting point of the Root region is to use half of its bounding box size related to the chosen scene division axis, thus generating two uniform subregions. This process is performed by the *AverageSpaceCut* function, as shown in Figure 10.

```
procedure SUBDIVIDESPACE(Space Root, DivisionAxis Axis)
    DacrtPartition[Left, Right] = AverageSpaceCut(Root, Axis)

    if Length(Right.boundBox - CamPosition) < Length(Left.boundBox - Cam-
Position) then
        Swap(Left, Right)
    end if

    integer rPivot = SplitRays(Root.rStart, Root.rEnd, Left.boundBox)
    integer tPivot = SplitTriangles(Root.tStart, Root.tEnd, Left.boundBox)

    DacrtPartition[Left].rStart = Root.rStart
    DacrtPartition[Left].tStart = Root.tStart
    DacrtPartition[Left].rEnd = rPivot
    DacrtPartition[Left].tEnd = tPivot

    DacrtPartition[Right].rStart = rPivot
    DacrtPartition[Right].tStart = tPivot
    DacrtPartition[Right].rEnd = Root.rEnd
    DacrtPartition[Right].tEnd = Root.tEnd

    return DacrtPartition[Left, Right]
end procedure
```

Fig. 10: Pseudocode of the *SubdivideSpace* function used in Figure 6.

The use of the *AverageSpaceCut*, a method that only considers the size of regions, can cause an unbalanced data division, once it ignores two main factors: triangles distribution and ray incidence. A better method needs to take into consideration these two factors and produce subregions of different sizes to maximize the uniformity between threads/cores use. Our first attempt, however, shows that more elaborate methods tend to consume more time than the *AverageSpaceCut*, negatively impacting the time consumed by the algorithm as a whole.

Data division is performed by pivots, as shown previously, so that *rPivot* and *tPivot* represent, respectively, the splitting point between two regions of rays and triangles in the vectors of ids (that are related to real data). The functions *SplitRays* and *SplitTriangles*, as shown in Figure 10, are responsible for splitting data, moving the indices of the tested region to the beginning of the vectors in a process similar to the filtering method described in [14].

To the correct working of steps 2 and 3, the definition of rays pivots [rStart, rEnd] and triangles pivots [tStart, tEnd], which

implicitly delimit the divisions, must obey the order given by the smallest distance between camera position and centers of the subregions bounding boxes. For example, consider a given a region R to be divided into Left and Right. In this case the pivots of the Left subregion must contain the data closest to the camera. If the camera is closest to the Right, a swap between subregions is required.

C. Bounding Box Expansion and Overlapping

The process of subdividing the space (Subsection IV-B) creates artifacts at the subregions edges. Rays that intersect the space close to the edges are prevented from intersecting the triangles that normally would do, as shown in Figure 11A. Triangles are split by its center and rays by the nearest bounding box intersect point, thus steps 2 and 3 of rays transference will not perform properly, as depicted in Figure 11B, once the ray's nearest intersect point is located in a neighboring subregion to the triangle.

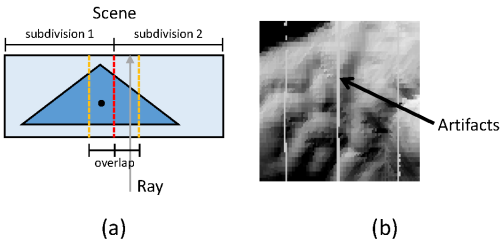

Fig. 11: (A) Triangle and ray are splitted in different subdivisions. (B) Artifacts generated in the intermediate image of case (A).

To address this problem it is necessary a method for bounding box expansion and overlapping. The scene bounding box is expanded by incrementing one point at each dimension before the space subdivision, as shown in Figure 6. This will prevent the problem from occurring in the scene edges.

Note that the artifacts generated by rays not intersecting triangles still occur in the subregions edges even after scene bounding box expansion. This way, after the space subdivision an one point increment in all subregions bounding boxes is performed, causing a multiple overlap and avoiding the problem, as shown in Figure 11A.

D. Thread Pool

The number of subregions may be different than the number of threads/cores of the deployment hardware and, to achieve higher performance in these scenarios, we adopt a scheme for distribution of work through a thread pool pattern, as seen in Figure 6.

The thread pool pattern consists in worker threads that consume work items, organized in a queue, until they are unavailable (empty queue). Worker threads are permanent between renderization frames, so overhead by thread creation and destruction is insignificant. Furthermore, this solution prevents that context-switching of an excessive number of threads damages the overall performance. Work items record the tasks to be performed.

Another feature given by the thread pool solution is the easy distribution of the worker threads across available resources. In a modern CPU it is common to find the number of virtual threads bigger than physical cores, so in these cases it is firstly necessary to examine the algorithm performance with different setups. In our tests, for example, there are some scenarios that performed better with more virtual threads than physical cores.

Each ray transference round in steps 2 and 3 (Subsection III-B) of the proposed algorithm is considered a set of work items, since each subregion can be responsible for one transference per round, as shown in Figure 6. After each round, and consequently creation of work items, the algorithm waits until the current tasks finishes. Due to rays transference nature, the number of available (sleeping) worker threads will increase as the rounds are performed.

The thread pool implementation can be performed by a multi-platform Application Programming Interface (API) or by specific Operational System (OS) API, however, it is important to be able to ensure processor affinity, so that the worker threads will execute only on designated CPU cores to ensure maximum cache efficiency. APIs that implement fork/join threading models and offer no control over threads are not an option for implementing thread pools, therefore APIs such as OpenMP [25] cannot be used. Thus, we chose to utilize the Microsoft Windows Thread Pool API [26]. In our tests it shows the minimum overhead when compared against third-party alternatives for the specific OS.

V. Experiments and Results

All experiments were conducted at a fixed resolution of 640x480 pixels in an Intel Core i7-4700MQ CPU, Turbo Boost disabled (all 4 cores clocked at 2.4 GHz) and 8 GB of RAM. During the tests, conic packet optimization was not utilized. We used public available and well-know models [1] that range from 69K to 7.21M of triangles (Figure 1). These models were intentionally chosen to recreate the setup used by Mora [14].

All scenes are considered as dynamic and only primary rays were casted, though the proposed solution can be easily extended to cast secondary and random rays (path-tracing).

In Table I we show an experiment with four different threading settings. Firstly, we tested the models against the original DACRT algorithm [14], single threaded, taken the results as baseline to the others experiments. After we tested our algorithm against the same models with the following hardware configurations: [2 threads, 2 cores], [4 threads, 4 cores] and [8 threads, 4 cores]. In these scenarios, our algorithm ran up to 2.42x faster than the original solution.

As expected, our algorithm is very sensitive to memory bandwidth limits. As we increase the size of the models, from 69K to 7.21M, which represents a two order of magnitude increase, the algorithm speedup, and consequently the algorithm efficiency, tends to decrease, as shown in Figure 12.

Analyzing the *Bunny* and *Armadillo* models we noted that, even with significantly different sizes, the speedup behavior of

TABLE I: Benchmark between multi-threaded and single-threaded approaches.

Scene	No. Triangles	Best Division Axis	Threads/Speedup			
			Original DACRT	2	4	4+4
Bunny	69,451	Y	1x	1.49x	2.42x	2.42x
Armadillo	345,944	X	1x	1.52x	1.98x	2.26x
Dragon	871,414	X	1x	1.21x	1.84x	1.96x
Happy Buddha	1,087,716	Y	1x	1.29x	1.81x	2.01x
Asian Dragon	7,219,045	X	1x	1.08x	1.52x	1.68x

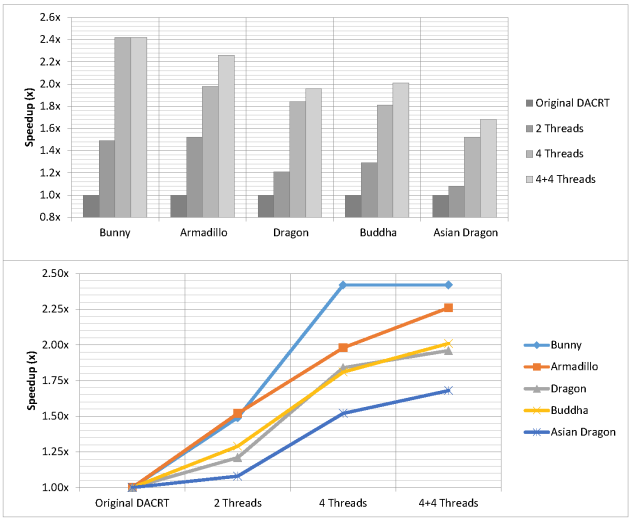

Fig. 12: Speedup comparative of the tested models.

the models when performed with 2 threads is nearly identical. However, when performed with more threads, the *Bunny* model reaches its maximum gain with 4 threads, whereas the *Armadillo* model continues showing performance gains when the number of threads was increased from 4 to 8. Such difference can be explained by the natural tendency of memory bandwidth saturation with more threads sharing the same memory channel.

The models *Dragon* and *Happy Buddha* have similar sizes, and consequently similar speedup behaviors. The *Asian Dragon* model shows the worst parallel efficiency, since its size is about 7x times larger than *Happy Buddha*.

Furthermore, if we analyze the five models tested, we can observe that, with the exception of the Bunny model, all others demonstrated performance gains when performed with more virtual threads than physical cores, situation where CPU cache is filled with smaller chunks of data given by the bigger number of divisions.

The choice of the correct division axis, as discussed in Subsection IV-A, can greatly impact in the algorithm performance. Table I shows the best division axis for each tested model. In our tests the camera is always placed in the front side of the scene, therefore the presence of only the x and y axes. Figure 13 shows that we gained up to 13% in performance when choosing the most appropriate axis in a 4 threads setup.

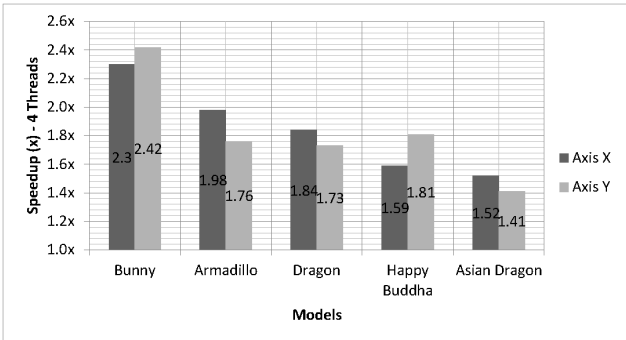

Fig. 13: Speedups for difference choices of division axes.

The main reason for the performance difference between division axes is due to the fact that models are usually asymmetric, impacting in the data division performed by the algorithm. If a model has a high density triangle region, as in the upper side of the *Armadillo* model, some threads could have much more work (data) than others if the best axis is not selected.

Conic packet optimization, an important feature of the original DACRT [14], can be easily adapted to our parallel approach by simply modifying part of the code in Figure 6 to use conic packets instead of rays.

VI. Conclusions and Future Work

In this paper we introduced a new algorithm of divide-and-conquer ray-tracing that exploits parallel hardware and proposes a generic scheme that can be adapted to an arbitrary number of threads. This algorithm maintains memory management minimal and deterministic and does not use any data structure for spatial subdivision.

Given the increasing number of CPUs cores in recent releases, our algorithm proved to be more suitable than the original solution [14], presenting considerable gains of performance when compared to single thread solutions.

Recently, Ravichandran and Narayanan [27] presented a parallel version of DACRT that runs exclusively on GPU, exploiting primitives like sort and reduce. A roughly performance comparison shows that our solution is significantly faster, although this is difficult to argue due to the different architectures.

For future work, we plan to investigate the implementation of an efficient divide-and-conquer ray-tracing that uses ray sampling [28], which exploits the distribution of rays to construct an acceleration data structure and derive a new cost metric to avoid inefficient subdivision where the number of rays is not sufficiently reduced. We believe that the combination of our and Nabata [28] methods can enable ray-tracing of high resolution frames in real-time.

Furthermore, we are currently investigating an automatic and dynamic method for choosing the best division axis,

analyzing each potential subdivision individually, and looking for better methods of subdividing the space that take in consideration the density of triangles and rays incidence, since the algorithm will scale better as the polygon count grows.

APPENDIX

```
struct DacrtPartition
    Pivot terminatedRay
    Pivot rStart, rEnd // rayStart and rayEnd pivots
    Pivot tStart, tEnd // triangleStart and triangleEnd pivots
    AxisAlignedBoundingBox boundBox
end struct
```

Fig. 14: Details of the *struct DacrtPartition* used in Figure 6.

ACKNOWLEDGMENT

Cícero Augusto de Lara Pahins would like to thank CAPES-Brazil for financial support.

REFERENCES

[1] S. C. G. Laboratory, "The stanford 3d scanning repository," http://www-graphics.stanford.edu/data/3Dscanrep/, Mar. 2014, online. Accessed 04-March-2014.

[2] H. Friedrich, J. Günther, A. Dietrich, M. Scherbaum, H.-P. Seidel, and P. Slusallek, "Exploring the use of ray tracing for future games," in *Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*, ser. Sandbox '06. New York, NY, USA: ACM, 2006, pp. 41–50. [Online]. Available: http://doi.acm.org/10.1145/1183316.1183323

[3] A. Reshetov, A. Soupikov, and J. Hurley, "Multi-level ray tracing algorithm," in *ACM SIGGRAPH 2005 Papers*, ser. SIGGRAPH '05. New York, NY, USA: ACM, 2005, pp. 1176–1185. [Online]. Available: http://doi.acm.org/10.1145/1186822.1073329

[4] C. Wächter and A. Keller, "Instant ray tracing: The bounding interval hierarchy," in *Proceedings of the 17th Eurographics Conference on Rendering Techniques*, ser. EGSR'06. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2006, pp. 139–149. [Online]. Available: http://dx.doi.org/10.2312/EGWR/EGSR06/139-149

[5] I. Wald, T. Ize, A. Kensler, A. Knoll, and S. G. Parker, "Ray tracing animated scenes using coherent grid traversal," in *ACM SIGGRAPH 2006 Papers*, ser. SIGGRAPH '06. New York, NY, USA: ACM, 2006, pp. 485–493. [Online]. Available: http://doi.acm.org/10.1145/1179352.1141913

[6] A. Lagae and P. Dutré, "Compact, fast and robust grids for ray tracing," in *ACM SIGGRAPH 2008 Talks*, ser. SIGGRAPH '08. New York, NY, USA: ACM, 2008, pp. 20:1–20:1. [Online]. Available: http://doi.acm.org/10.1145/1401032.1401059

[7] M. E. Lee, R. A. Redner, and S. P. Uselton, "Statistically optimized sampling for distributed ray tracing," in *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '85. New York, NY, USA: ACM, 1985, pp. 61–68. [Online]. Available: http://doi.acm.org/10.1145/325334.325179

[8] A. J. F. Kok and F. W. Jansen, "Adaptive sampling of area light sources in ray tracing including diffuse interreflection," *Computer Graphics Forum*, vol. 11, no. 3, pp. 289–298, 1992. [Online]. Available: http://dx.doi.org/10.1111/1467-8659.1130289

[9] H. Brönnimann and M. Glisse, "Octrees with near optimal cost for ray-shooting," *Comput. Geom. Theory Appl.*, vol. 34, no. 3, pp. 182–194, Jul. 2006. [Online]. Available: http://dx.doi.org/10.1016/j.comgeo.2005.09.003

[10] A. Lagae and P. Dutré, "Accelerating ray tracing using constrained tetrahedralizations," IRT 2008 Poster, Symposium on Interactive Ray Tracing 2008, Los Angeles, USA, p. 1, August 2008. [Online]. Available: http://www.sci.utah.edu/rt08/

[11] N. A. Carr, J. Hoberock, K. Crane, and J. C. Hart, "Fast gpu ray tracing of dynamic meshes using geometry images," in *Proceedings of Graphics Interface 2006*, ser. GI '06. Toronto, Ont., Canada, Canada: Canadian Information Processing Society, 2006, pp. 203–209. [Online]. Available: http://dl.acm.org/citation.cfm?id=1143079.1143113

[12] M. Shevtsov, A. Soupikov, and A. Kapustin, "Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes," *Computer Graphics Forum*, vol. 26, no. 3, pp. 395–404, 2007. [Online]. Available: http://dx.doi.org/10.1111/j.1467-8659.2007.01062.x

[13] B. Choi, R. Komuravelli, V. Lu, H. Sung, R. L. Bocchino, S. V. Adve, and J. C. Hart, "Parallel sah k-d tree construction," in *Proceedings of the Conference on High Performance Graphics*, ser. HPG '10. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2010, pp. 77–86. [Online]. Available: http://dl.acm.org/citation.cfm?id=1921479.1921492

[14] B. Mora, "Naive ray-tracing: A divide-and-conquer approach," *ACM Trans. Graph.*, vol. 30, no. 5, pp. 117:1–117:12, Oct. 2011. [Online]. Available: http://doi.acm.org/10.1145/2019627.2019636

[15] T. Whitted, "An improved illumination model for shaded display," in *Proceedings of the 6th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '79. New York, NY, USA: ACM, 1979, pp. 14–. [Online]. Available: http://doi.acm.org/10.1145/800249.807419

[16] D. Barboza and E. Clua, "A gpu-based data structure for a parallel ray tracing illumination algorithm," in *Proceedings of SBGames 2011*. SBC, 2011.

[17] H. Du, M. Sanchez-Elez, N. Tabrizi, N. Bagherzadeh, M. L. Anido, and M. Fernandez, "Interactive ray tracing on reconfigurable simd morphosys," in *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*, ser. ASP-DAC '03. New York, NY, USA: ACM, 2003, pp. 471–476. [Online]. Available: http://doi.acm.org/10.1145/1119772.1119869

[18] P. Santos, "Ray tracing dynamic scenes on the gpu," Master's thesis, Pontífica Universidade Católica do Rio de Janeiro, 2009.

[19] A. Keller and C. Waechter, "Efficient ray tracing without acceleration data structure," Patent Application, 09 2009, uS 2009/0225081 A1. [Online]. Available: http://www.patentlens.net/patentlens/patent/US_2009_0225081_A1/en/

[20] M. Eisemann, P. Bauszat, S. Guthe, and M. Magnor, "Geometry presorting for implicit object space partitioning," *Comp. Graph. Forum*, vol. 31, no. 4, pp. 1445–1454, Jun. 2012. [Online]. Available: http://dx.doi.org/10.1111/j.1467-8659.2012.03140.x

[21] A. Williams, S. Barrus, R. K. Morley, and P. Shirley, "An efficient and robust ray-box intersection algorithm," in *ACM SIGGRAPH 2005 Courses*, ser. SIGGRAPH '05. New York, NY, USA: ACM, 2005. [Online]. Available: http://doi.acm.org/10.1145/1198555.1198748

[22] T. Akenine-Möller, "Fast 3d triangle-box overlap testing," in *ACM SIGGRAPH 2005 Courses*, ser. SIGGRAPH '05. New York, NY, USA: ACM, 2005. [Online]. Available: http://doi.acm.org/10.1145/1198555.1198747

[23] T. Möller and B. Trumbore, "Fast, minimum storage ray-triangle intersection," *J. Graph. Tools*, vol. 2, no. 1, pp. 21–28, Oct. 1997. [Online]. Available: http://dx.doi.org/10.1080/10867651.1997.10487468

[24] A. Hoedt, "Divide-and-conquer ray tracing on the gpu," http://asgerhoedt.dk/?page_id=240, Nov. 2012, online. Accessed 04-July-2013.

[25] O. A. R. Board, "The openmp api specification for parallel programming," http://openmp.org/wp/, Dec. 2013, online. Accessed 09-December-2013.

[26] M. Corporation, "Thread pool api (windows)," http://msdn.microsoft.com/en-us/library/windows/desktop/ms686766\%28v=vs.85\%29.aspx, Mar. 2014, online. Accessed 04-March-2014.

[27] S. Ravichandran and P. J. Narayanan, "Parallel divide and conquer ray tracing," in *SIGGRAPH Asia 2013 Technical Briefs*, ser. SA '13. New York, NY, USA: ACM, 2013, pp. 30:1–30:4. [Online]. Available: http://doi.acm.org/10.1145/2542355.2542393

[28] K. Nabata, K. Iwasaki, Y. Dobashi, and T. Nishita, "Efficient divide-and-conquer ray tracing using ray sampling," in *Proceedings of the 5th High-Performance Graphics Conference*, ser. HPG '13. New York, NY, USA: ACM, 2013, pp. 129–135. [Online]. Available: http://doi.acm.org/10.1145/2492045.2492059