UNIVERSIDADE FEDERAL DO CEARÁ

CENTRO DE CIÊNCIAS

DEPARTAMENTO DE COMPUTAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

RODOLFO FELIPE SGANZERLA SABINO

EXPLORANDO O HARDWARE GRÁFICO: DESENVOLVIMENTO DE ALGORITMOS PARALELOS DE COMPUTAÇÃO DE OBBS, GERAÇÃO DE HIERARQUIA DE VOLUMES, MODELAGEM DE BVH DE DOIS NÍVEIS E IMPLEMENTAÇÃO DE UMA PIPELINE DE RENDERIZAÇÃO HIBRIDA

FORTALEZA

2020

RODOLFO FELIPE SGANZERLA SABINO

EXPLORANDO O HARDWARE GRÁFICO: DESENVOLVIMENTO DE ALGORITMOS PARALELOS DE COMPUTAÇÃO DE OBBS, GERAÇÃO DE HIERARQUIA DE VOLUMES, MODELAGEM DE BVH DE DOIS NÍVEIS E IMPLEMENTAÇÃO DE UMA PIPELINE DE RENDERIZAÇÃO HIBRIDA

Dissertação apresentada ao Curso de Programa de Pós-Graduação em Ciência da Computação do do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Ciência da Computação. Área de Concentração: Computação Gráfica

Orientador: Prof. Dr. Creto Augusto Vidal

Coorientador: Prof. Dr. Joaquim Bento Cavalcante Neto

FORTALEZA

2020

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

S121e Sabino, Rodolfo Felipe Sganzerla.
Explorando o Hardware Gráfico: Desenvolvimento de Algoritmos Paralelos de Computação de OBBs, Geração de Hierarquia de Volumes, Modelagem de BVH de Dois Níveis e Implementação de uma Pipeline de Renderização Hibrida / Rodolfo Felipe Sganzerla Sabino. – 2020.
80 f. : il. color.

Dissertação (mestrado) – Universidade Federal do Ceará, 1, Fortaleza, 2020.
Orientação: Prof. Dr. Creto Augusto Vidal.
Coorientação: Prof. Dr. Joaquim Bento Cavalcante Neto.

1. Computação Gráfica. 2. Ray Tracing. 3. Bounding Volume Hierarchy. I. Título.

CDD

RODOLFO FELIPE SGANZERLA SABINO

EXPLORANDO O HARDWARE GRÁFICO: DESENVOLVIMENTO DE ALGORITMOS PARALELOS DE COMPUTAÇÃO DE OBBS, GERAÇÃO DE HIERARQUIA DE VOLUMES, MODELAGEM DE BVH DE DOIS NÍVEIS E IMPLEMENTAÇÃO DE UMA PIPELINE DE RENDERIZAÇÃO HIBRIDA

Dissertação apresentada ao Curso de Programa de Pós-Graduação em Ciência da Computação do do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Ciência da Computação. Área de Concentração: Computação Gráfica

Aprovada em:

BANCA EXAMINADORA

Prof. Dr. Creto Augusto Vidal (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Joaquim Bento Cavalcante Neto (Coorientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. José Gilvan Rodrigues Maia
Universidade Federal do Ceará (UFC)

AGRADECIMENTOS

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

RESUMO

Recentes iterações no hardware programável paralelo moderno introduziram GPUs com suporte a ray tracing nativo, oferecendo oportunidades de aceleração de hardware e incentivando o desenvolvimento de técnicas que possibilitam renderização foto realista. Esse cenário reforça a demanda por métodos eficientes de renderização que suportem tecnologias de ray tracing. Neste trabalho, são desenvolvidos algoritmos paralelos de computação de **Oriented Bounding Boxes (OBBs)** e geração de hierarquia de volumes na GPU para acelerar ainda mais os algoritmos de ray tracing existentes. É proposto um modelo de estruturação de cena, baseado em uma **Bounding Volume Hierarchy (BVH)** de dois níveis, capaz de gerar hierarquias de qualidade superior, e de proporcionar maior eficiência de travessia. Esse modelo, além de não gerar custos adicionais de execução, aumenta a flexibilidade de implementação de diversos algoritmos de construção diferentes simultaneamente. Todos os algoritmos propostos são implementados em uma pipeline de ray tracing hibrida integralmente implementada em GPU. Resultados experimentais demonstram que os algoritmos propostos são robustos, eficientes e adequados para as pipelines de renderização hibridas modernas.

Palavras-chave: OBB; PCA; GPU; Paralelismo; Computação Gráfica; Ray Tracing; Path Tracing; Ray Casting; BVH; Hierarquia de Volumes; LBVH; Pipeline Hibrida; Deferred Rendering; GI; Monte Carlo; Ray Differentials; Cone Tracing; Filtragem de Textura; Importance Sampling; Stratified Sampling; Lorentz Transformation; AABB; Slabs Algorithm; Ambient Occlusion; Diffuse Shading.

ABSTRACT

Recent iterations on modern programmable graphics pipeline introduced native ray tracing support on GPUs, offering opportunities for hardware acceleration and encouraging the development of techniques that enable photo realistic rendering. This scenario strengthen the demand for efficient rendering methods in alignment to ray tracing technologies. This work develops parallel algorithms for computing **Oriented Bounding Boxes (OBBs)** and hierarchical volume generation on the GPU for further acceleration of existing ray tracing algorithms. It is proposed a scene structure model based on a two-level **Bounding Volume Hierarchy (BVH)**, capable of generating hierarchies of superior quality, and offering more traversal efficiency. This model, on top of not adding further execution costs, endues the flexibility of implementation of a variety of building algorithms simultaneously. All proposed algorithms are implemented within a hybrid ray tracing pipeline running entirely on the GPU. Experimental results demonstrate that the proposal algorithms are robust, efficient and adequate for modern hybrid rendering pipelines.

Keywords: OBB; PCA; GPU; Paralelismo; Computação Gráfica; Ray Tracing; Path Tracing; Ray Casting; BVH; Hierarquia de Volumes; LBVH; Pipeline Hibrida; Deferred Rendering; GI; Monte Carlo; Ray Differentials; Cone Tracing; Filtragem de Textura; Importance Sampling; Stratified Sampling; Lorentz Transformation; AABB; Slabs Algorithm; Ambient Occlusion; Diffuse Shading.

SUMÁRIO

1 INTRODUÇÃO 9
2 TRABALHOS RELACIONADOS 12
2.1 Volumes Delimitadores 12
2.2 Hierarquia de Volumes 12
2.3 BVHs de Dois Níveis 14
2.4 Primitivas de Dados Paralelos 15
2.5 Arquitetura Paralela 17
2.6 *Deferred Rendering* e *Pipelines* de Renderização Hibrida 18
3 METODOLOGIA 20
3.1 OBBs 20
3.1.1 Representando OBBs 20
3.1.2 Construindo e Atualizando OBBs 21
3.1.3 Construindo OBBs 22
3.1.4 Atualizando OBBs 23
3.2 Algoritmo de Interseção Entre Raio-OBB 24
3.3 BVH 26
3.3.1 Computação de Morton Codes 27
3.3.2 Ordenação dos Morton Codes 28
3.3.3 Construção da Árvore Binária 29
3.3.4 Computação dos BVs 34
3.4 Travessia da BVH 36
3.5 União de Duas OBBs 38
3.6 BVH de Dois Níveis 41
3.6.1 Construção do Nível de Baixo 42
3.6.2 Construção do Nível de Cima 43
3.6.3 Travessia da BVH de Dois Níveis 44
3.7 Pipeline Hibrida 46
3.7.1 Construção de cena 47
3.7.2 Passe de Rasterização 48
3.7.3 Passe de Path Tracing 49

3.7.4 ***Geração de Raios*** 50

3.7.5 ***Travessia*** 53

3.7.6 ***Interpolação e Filtragem de Textura*** 54

3.7.7 ***Cálculo de Iluminação*** 56

4 **RESULTADOS** 63

4.1 **Base de Dados de Avaliação** 63

4.2 **Qualidade da BVH** 64

4.3 **Eficiência Dos Algoritmos de Construção de BVHs** 65

4.4 **Robustez e Desempenho do *Path Tracer*** 66

5 **CONCLUSÕES E TRABALHOS FUTUROS** 75

REFERÊNCIAS 78

1 INTRODUÇÃO

A adoção de representações de cena de forma hierárquica tem o potencial de dinamicamente reduzir a complexidade computacional de *ray tracing* com respeito ao número de interseções. **Bounding Volume Hierarchies (BVHs)** são consideradas a melhor opção para a representação hierárquica de cenas. Além do agrupamento hierárquico de primitivas, as estruturas de aceleração das BVHs também proporcionam a garantia de uso delimitado de espaço de memória (WYMAN; MARRS, 2019). Embora BVHs possam acelerar o processo de renderização de forma significativa, o custo envolvido na construção e na atualização dessas estruturas em cenas dinâmicas impõe um limite no número de primitivas que podem ser animadas a *frame rates* interativas.

O paralelismo vem sendo a principal força por trás da melhoria de desempenho computacional no *hardware* moderno. Portanto, métodos que implementam BVH devem explorar execução paralela se quiserem escalar com arquiteturas de *hardware* futuras (LAUTERBACH *et al.*, 2009). A especialização de micro arquiteturas para BVHs aceleradas por hardware tem o potencial de alcançar ganhos significativos de desempenho (DOYLE *et al.*, 2013). Avanços recentes em **hardware gráfico programável (GPU)** e **interfaces de programação (APIs)** (TATARCHUK; LEFOHN, 2017) criaram plataformas poderosas e flexíveis que possibilitam aos desenvolvedores implementar *frameworks* de auto grau de paralelismo de dados e aceleração de *harware* para algoritmos de *ray tracing* (PARKER *et al.*, 2010).

Duas das APIs de computação gráfica mais populares para *ray tracing* atualmente, *DirectX (DXR)* e *Vulkan*, são multiplataforma, e oferecem suporte para *hardware* de diferentes fornecedores. Em suas versões mais recentes, essas APIs fazem uso de BVHs implementadas a critério de cada fabricante. O modelo de suas BVHs segue uma estrutura de dois níveis. A parte inferior da BVH contém os dados de primitivas geométricas, tais como vértices e seus atributos, os quais representam objetos em uma cena. A parte superior da BVH, por sua vez, organiza descritores de instâncias, cujas estruturas contêm referências para conjuntos particulares de geometria em associação a suas matrizes de transformação (WYMAN; MARRS, 2019; SUBTIL, 2018).

As APIs dão suporte à instanciação dos nós pertencentes à parte inferior da BVH por meio dos descritores da parte superior. Em ambas as APIs, o uso de BVHs é dividido em uma fase de pré-processamento, responsável pela construção das estruturas de aceleração da parte inferior, e uma fase de rebalanceamento, que atualiza a estrutura da parte superior a

custo relativamente baixo para cenas dinâmicas (WYMAN; MARRS, 2019; SUBTIL, 2018; BARRÉ-BRISEBOIS *et al.*, 2019).

O contexto no qual a estrutura de aceleração será aplicada influencia o seu projeto. Ao longo dos anos, enquanto algumas BVHs apresentam tempo de construção baixo e bom desempenho para cenas dinâmicas, ao custo de percurso elevado (LAUTERBACH *et al.*, 2009), outras focam em eficiência de descarte aprimorada,mas que são mais adequadas a cenas estáticas (STICH *et al.*, 2009). Ademais, o projeto de cada BVH salienta um comprometimento constante entre suas vantagens e desvantagens.

O projeto de uma BVH de dois níveis propõe uma abordagem diferente em que, ao invés de comprometer-se com um único tipo de estrutura de aceleração para todas as primitivas geométricas, permite a correspondência de diferentes BVHs, criando um sistema onde diferentes estruturas de BVH podem coexistir. Essa ideia permite que APIs sejam projetadas para serem o mais abstratas e inclusivas possíveis. Porém, é o contexto de cada aplicação que dita qual BVH funciona melhor. O paradigma que dá suporte à instanciação de objetos em uma cena assemelha-se muito ao modelo de programação orientada a objetos e ao fluxo de trabalho artístico.

Neste trabalho, é apresentado o desenvolvimento de uma BVH de dois níveis com foco em *ray tracing* na GPU. A BVH é composta de uma estrutura hierárquica lógica e única na parte superior, representando instâncias de malhas em uma cena, enquanto a parte inferior contêm uma lista de BVHs para cada malha. Sua computação é feita predominantemente na GPU. A BVH faz uso de **oriented bounding boxes (OBBs)**, em contraste com **axis-aligned bounding boxes (AABBs)** normalmente utilizadas, com o objetivo de maximizar a região de interseção da bounding box com seu conteúdo.

São fornecidos algoritmos para construção e atualização das OBBs, algoritmos eficientes para computação de interseções entre raio e OBB na GPU, bem como algoritmos para construção e percurso das BVHs. Em seguida, a proposta de uma arquitetura da BVH de dois níveis é descrita em termos de primitivas de dados paralelos. Finalmente, essa BVH é inserida no projeto de um *ray tracer* prático e implementado na GPU.

Em síntese, as contribuições deste trabalho são:

- O desenvolvimento de algoritmos eficientes para representação e computação de OBBs, bem como para computação da interseção entre raio e OBB na GPU.
- O desenvolvimento de algoritmos paralelos de computação hierárquica dos volumes da

BVH usando OBBs, apresentando diferentes níveis de granularidade, desempenho e qualidade dos volumes gerados em relação a construções usando AABBs.

- A modelagem e a proposta de um método paralelo para a construção e percurso de uma BVH de dois níveis como estrutura de representação da cena baseado em instância.
- Métodos paralelos para a construção e percurso das hierarquias que formam a BVH de dois níveis.
- A implementação dos algoritmos propostos em uma *pipeline* de renderização hibrida fazendo uso de aceleração gráfica para o desenvolvimento de um *path tracer* estocástico.

O restante desta dissertação está organizado em mais quatro capítulos. No Capítulo 2, são apresentados os trabalhos relacionados. No Capítulo 3, é apresentada a metodologia proposta. No Capítulo 4, são apresentados os resultados. No Capítulo 5, são apresentadas as conclusões e a discussão sobre possíveis trabalhos futuros.

2 TRABALHOS RELACIONADOS

2.1 Volumes Delimitadores

Uma **caixa delimitadora (BB)** é um tipo de **volume delimitador (BV)** de formato paralelepipédico, comumente representado por **Axis-Aligned Bounding Boxes (ABBs)**, as quais possuem restrições de alinhamento, ou por **Oriented Bounding Boxes (OBBs)**, as quais podem assumir orientação arbitrária no espaço. Uma BB justa substitui o feixo convexo de um conjunto de primitivas para possibilitar a computação de algoritmos de interseção de forma mais rápida e simples.

BBs tornaram-se relevantes em computação gráfica como estruturas de aceleração para rápido percurso em cena, detecção de colisão e descarte de polígonos. A origem de sua conceptualização como primitiva volumétrica remonta ao **Teorema de separação de eixos (SAT)**. A aplicação desse teorema pode ser observada em uma vasta quantidade de algoritmos de interseção com BBs, como por exemplo (KAY; KAJIYA, 1986; WILLIAMS *et al.*, 2005; MAJERCIK *et al.*, 2018). O cerne desses algoritmos está na redução do problema de interseção de raio com AABB para a união das interseções de pares de hiperplanos ortogonais espaçados que delimitam um volume de interesse no espaço.

Muitos algoritmos de **bounding volume hiararchies (BVHs)** usam AABBs para particionamento espacial, mas outras representações de BBs também podem ser usadas para o mesmo propósito. A escolha de OBBs afeta o desempenho de uma BVH, embora OBBs ofereçam melhor desempenho em descarte de polígonos em comparação com ABBs. Por essa razão, neste trabalho, é estudado o uso de OBBs. Deriva-se uma representação para essas OBBs e desenvolvem-se algoritmos para seus cálculo e para suas atualizações. Também, desenvolvem-se algoritmos de interseção de OBBs com raios. Vale enfatizar que essas operações são necessárias para as BVHs de forma que seus custos sejam similares aos obtidos com o uso de AABBs. Nesse sentido, na visão do autor, o uso de OBBs em BVHs é mais eficiente do que o uso de AABBs.

2.2 Hierarquia de Volumes

BVHs separam as primitivas de uma cena em grupos, de modo a explorar a coerência espacial existente na disposição desses objetos. Tipicamente, esses grupos são organizados em uma estrutura de árvore, onde cada nó contém a informação espacial necessária para facilitar

testes de descarte da parte associada da cena, a qual, normalmente é representada por uma AABB. O objetivo da BVH é minimizar o número de cálculos de interseção entre os raios e as primitivas. Os benefícios do uso de BVHs estão associados a seus custos de construção, atualização e percurso de suas estruturas.

Linear Bounding Volume Hierarchy (LBVH) (LAUTERBACH *et al.*, 2009) e **Split Bounding Volume Hierarchy (SBVH)** (STICH *et al.*, 2009) são dois algoritmos de BVH de destaque (PARKER *et al.*, 2010). Enquanto o algoritmo LBVH foca na velocidade de construção, o SBVH foca na qualidade de geração da hierarquia da árvore. O LBVH utiliza uma abordagem de computação paralela na construção de sua BVH, operando sobre cada primitiva da cena, alcançando um nível de paralelismo fino e gerando uma BVH de tamanho fixo. A atualização feita pelo LBVH à estruturas de cenas dinâmicas pode ser realizada em tempo linear.

O SBVH também opera sobre cada primitiva, e, como o LBVH, também faz uso de AABBs, resultando em BBs mal justas. No entanto, o SBVH tenta evitar a sobreposição de BBs a fim de melhorar a eficiência de descarte, mas acaba introduzindo o problema de divisão de primitivas, que pode gerar um número imprevisível de nós, e tornar a operação de construção da BVH bem mais complexa de se paralelizar. Ademais, isso também faz com que o processo de atualização da BVH seja mais difícil de se implementar.

O algoritmo de construção de BVH abordado neste trabalho é desenvolvido com base no LBVH. O processo de construção da BVH segue o paradigma de primitivas de dados paralelos e permite fácil construção e atualização da BVH para cenas dinâmicas. Diferentemente dos algoritmos previamente mencionados, entretanto, o algoritmo apresentado não gera uma única BVH para toda a cena.

O algoritmo apresentado busca uma abordagem diferente daquela utilizada na construção de uma única BVH. Assim, constrói múltiplas BVHs diferentes seguindo a heurística de coerência espacial presente na organização dos objetos da cena. Dentro de cada malha sobre a qual ele opera, cada primitiva é processada de forma paralela, mantendo um nível de paralelismo fino. O projeto desse algoritmo é orientado a uma *pipeline* que suporte a instanciação de objetos, em semelhança à arquitetura de APIs de *ray tracing* modernos (WYMAN; MARRS, 2019; SUBTIL, 2018).

Outras pesquisas foram desenvolvidas especialmente no tópico de BVHs aplicáveis a *ray tracing*. **Motion SBVH (MSBVH)** (GRüNSCHLOSS*et al.*, 2011) itera sobre SBVH na direção de suporte adicional a transformações em pequenas escalas de tempo. (DOYLE *et*

al., 2013) propõe uma micro arquitetura especializada para possibilitar BVHs com aceleração de hardware. (DU *et al.*, 2016) usa **Tetrahedron Swept Sphere (TSS)** como BV para sua BVH enquanto mantém complexidade linear na construção da BV e melhora a eficiência de descarte. Neste trabalho são exploradas as propriedades de transformações lineares no processo de atualização da BVH. O uso de OBBs também apresenta melhor eficiência de descarte.

Repetidas atualizações aplicadas a BVHs em cenas dinâmicas podem levar ao desbalanceamento das estruturas e implicam degradação de desempenho de sua percurso. Algoritmos adicionais de rebalanceamento de BVH, tal como (KARRAS; AILA, 2013), os quais modificam a sua estrutura e melhoram a qualidade da árvore, são necessários. Contudo, implicam o acréscimo de mais um passo no processamento. No método apresentado neste trabalho, as consequências decorrentes da degradação da BVH podem ser minimizadas. Para isso, apresentam-se diferentes abordagens, com custos computacionais diferentes, que ajudam a mitigar esse problema e que podem ser combinadas na implementação.

2.3 BVHs de Dois Níveis

BVH de dois níveis são estruturas atualmente usadas tanto no DirectX quanto no Vulkan (WYMAN; MARRS, 2019; SUBTIL, 2018), e possuem distintas vantagens em comparação com uma única estrutura de aceleração sobre todas as primitivas de uma cena.

Essas são estruturas de dados divididas em dois níveis, com a parte superior designada hierarquicamente superior à parte inferior. Os dois níveis podem estar a serviço de critérios de particionamento diferentes. Grupos de primitivas, como malhas, são atribuídos ao nível de baixo, cada grupo possuindo sua própria BVH, formando uma lista de BVHs. Em uma cena composta pelas instâncias dessas malhas, a parte superior é formada por uma única BVH. Ela é construída a partir das instâncias e suas folhas fazem referência às BVHs das respectivas malhas instanciadas.

Essa distinção de primitivas entre objetos lógicos, as instâncias, em contraste com objetos espaciais, malhas, permite a atualização seletiva de cada objeto individualmente e funciona bem em casos comuns contendo tanto as geometrias estáticas quanto as dinâmicas, para instanciação de objetos e animações de corpos rígidos (BENTHIN *et al.*, 2017).

A contribuição de (BENTHIN *et al.*, 2017) foca no aperfeiçoamento de BVH de dois níveis e aborda o problema da sobreposição de nós da parte superior da estrutura por meio da execução de uma operação de reentrelaçamento na estrutura de dados da árvore. Seu trabalho

lida com as restrições de um rastro de memória fixa, na busca por uma melhor aplicabilidade na GPU, porém o sucesso de sua abordagem é limitado pelo uso de espaço de armazenamento restringido para a criação de novos nós.

Neste trabalho o algoritmo de LBVH é usado para a construção das hierarquias. Apesar de possuir baixo custo de construção, LBVH gera uma hierarquia de baixa qualidade e se beneficiaria de algoritmos como (BENTHIN *et al.*, 2017). Entretanto, este trabalho foca no aspecto específico de desenvolvimento de uma BV alternativa e paralela, que fornece melhor eficiência de descarte e que pode ser implementada em um algoritmo de construção de BVH de dois níveis com custos comparáveis aos dos métodos existentes que adotam AABBs.

2.4 Primitivas de Dados Paralelos

O paralelismo em larga escala envolve a divisão de uma base de dados, larga o suficiente para a saturação de uma arquitetura *many-core*, e a sua computação distribuída por uma série de processos executando um mesmo algoritmo de forma independente, com um mínimo ou sem nenhuma comunicação entre si.

Aplicações que tiram proveito desse tipo de computação paralela, as quais fazem uso intensivo de memória, classificam suas operações quanto ao seu padrão de acesso de dados. APIs que adotam a arquitetura de linguagens centradas no paralelismo de dados têm o potencial para otimização do código pelo compilador, programabilidade flexível e melhor desempenho (LARSEN *et al.*, 2015).

(BLELLOCH, 1990) introduz o concento das primitivas de dados paralelos. O trabalho de (LARSEN *et al.*, 2015) explora *ray tracing* descrevendo sua arquitetura inteiramente em termos dessas primitivas. Pelo seu contexto de execução, *ray tracing* se torna um dos principais exemplos no uso de primitivas de dados paralelos.

O trabalho de (LARSEN *et al.*, 2015) descreve algumas das primitivas e como elas são usadas em *ray tracing*. Cada operação realiza uma chamada de execução onde vários processos paralelos operam em elementos que compõem o vetor de dados de entrada e escrevem elementos que compõem um vetor de dados de saída de acordo com a relação entre índices de entrada e saída pré-determinada pela natureza da operação. A tabela 1 faz um sumário das primitivas de dados paralelos abordadas.

Os métodos de computação paralela neste trabalho são modelados em função de primitivas de dados paralelos, trazendo clareza e padronização, assegurando a portabilidade dos

Tabela 1 – Primitivas de dados paralelos e seu uso em *ray tracing*.

Operação	**Opera Em**	**Relação Entrada / Saída**	**Fluxo de Dados**	**Uso Em *Ray Tracing***
Map	Opera em cada elemento do vetor de entrada	1:1 vetor de saída possui o mesmo tamanho que o vetor de entrada	Cada elemento do vetor de entrada é mapeado para um elemento do vetor de saída	Geração de raios, interseção de raios, acumulação de cor
Gather	Opera em um vetor de índices de tamanho igual ao vetor de saída	Entrada e saída podem possuir tamanhos diferentes	O índice especifica o local a ser lido do vetor de entrada	Computação do volume de cada nó da BVH, acumulação de amostras de raios para cada pixel
Scatter	Opera em um vetor de índices de tamanho igual ao vetor de entrada	Entrada e saída podem possuir tamanhos diferentes	O índice especifica o local a ser escrito do vetor de saída, possivelmente escrevendo no mesmo local múltiplas vezes	Travessia da BVH para a computação dos volumes
Reduce	Opera em todos os elementos do vetor de entrada	N:1 gera um único elemento de saída	Combina todos os elementos de entrada em uma única saída	Computação dos pontos de mínima e máxima de um conjunto de primitivas
Scan	Opera em cada elemento do vetor de entrada	1:1 vetor de saída possui o mesmo tamanho que o vetor de entrada	Executa *reduce* na sequência de elementos que se estende do início ao índice do elemento na respectiva localidade	Parte essencial no algoritmo de ordenação de raios

algoritmos apresentados entre múltiplas APIs, bem como princípio de validação conceitual e de aplicabilidade, e a fim de trazer uma idéia sobre o desempenho e limitações dos algoritmos.

2.5 Arquitetura Paralela

O *hardware* gráfico moderno expõe um paradigma de computação paralela capaz de executar grandes quantidades de processos concorrentes de forma extremamente eficiente. As versões mais recentes desse tipo de *hardware* permitem que sistema operacional submeta à execução na GPU cópias de um mesmo algoritmo em milhares de processos e a movimentação de dados entre a GPU e a memória principal RAM.

APIs tais como CUDA e OpenCL, generalistas, e DirectX, OpenGL e Vulkan, voltadas especificamente para computação gráfica de alto desempenho, expõem uma linguagem de programação para o desenvolvimento flexível de programas em *hardwares* especializados quais suportam a execução de milhares de processos concorrentes.

A GPU organiza uma coleção de milhares de unidades de processamento homogêneas capazes de executar um conjunto de instruções vetoriais (SIMD). A comunicação entre esses elementos é dada pela leitura e escrita de dados entre vários tipos de memória. Implementações expõem três níveis de memória explicitamente gerenciadas; Registradores dos processos, memória compartilhada de grupo que e acessível entre grupos de processos e *buffers* de armazenamento fora do *cache* visíveis entre todos os processos. A execução de um algoritmo nesse ambiente exige a leitura de movimentação de memória de um espaço para outro.

O sistema operacional deve realizar a alocação de memória e o instanciamento do algoritmo na GPU de forma explicita. A realização de um passe, a execução de vários grupos de processos, ocorre seguindo o modelo de dados paralelos. Uma típica execução envolve a leitura de uma posição no *buffer* de memória seguindo o índice atrelado ao processo, uma computação finita desse dado, possivelmente envolvendo cooperação entre processos de um mesmo grupo, e a escrita do resultados em um *buffer* de memória global. Cabe ao sistema operacional o controle do fluxo de dados pela ordem da invocação de passes e garantia de visibilidade após operações de escrita na memória exige o uso de barreiras de sincronização (MERRILL; GRIMSHAW, 2010).

Vários fatores podem afetar o desempenho de um algoritmo paralelo, não só no *hardware* gráfico moderno, mas qualquer arquitetura de paralelismo fino está sujeita um certo ponto a essas limitações. Dentre elas, a residência dita a quantidade de processos quais podem ser executados simultaneamente em um passe e é afetada pelo tamanho do algoritmo. Um

algoritmo muito grande exige a alocação de muitos recursos (registradores, memoria cache, etc), consequentemente, a GPU terá um desempenho pior executando algoritmos grandes.

A granularidade é a medida da quantidade de instruções executadas por um processo. Um algoritmo de granularidade fina executa um pequeno conjunto de instruções distribuídas em grande número de processos. Já um algoritmo de granularidade grossa executa um conjunto complexo de instruções em um numero reduzido de processos. Como o estamos lidando com um paradigma baseado em dados paralelo, a complexidade de um algoritmo muitas vezes está relacionada com a quantidade de dados que ele processa. Nas próximas sessões, são discutidas operações que exigem uma grande quantidade de dados. Operações de uso delimitado de dados apresentam melhor desempenho.

atividades similares ou não. Um alto grau de divergência de memória significa que processos vizinhos estão acessando fragmentos de memória global diferentes, prejudicando o desempenho da memória cache. Por sua vez, um alto grau de divergência de execução significa o grau em que processos ramificam e seguem linhas de execução diferente, e consequentemente, alguns processos podem terminar enquanto outros processos seguem executando (KARRAS, 2012b).

Um alto grau de divergência de execução pode afetar negativamente a ocupância, a medida de quantos processos estão em execução simultaneamente a um dado momento. Naturalmente, é preciso maximizar a quantidade de processos sendo executados. Arquiteturas de auto grau de paralelismo têm seu desempenho prejudicado quando possuem processos ociosos.

2.6 *Deferred Rendering* e *Pipelines* de Renderização Hibrida

Deferred rendering é uma técnica usada em computação gráfica para a redução dos custos de renderização pela divisão da *pipeline* em múltiplos passes e a delimitação da carga computacional de passes a se sucederem pelas dimensões de um *buffer (g-buffer)* de tamanho constante. O conceito de renderização hibrida se refere ao estratégia de combinação de diferentes faculdades do *hardware* gráfico servindo a diferentes paradigmas de renderização.

Esse trabalho desenvolve uma *pipeline* de renderização hibrida fazendo uso da técnica de *deferred rendering*. Estágios do processo de renderização são racionados em múltiplas operações sequenciais de dados paralelos. Rasterização é usada no primeiro passe para a computação dos raios primários e cálculos de iluminação são deferidos aos passos seguintes. A execução de passes subsequentes acontece com base no *buffer* gerado pelo primeiro passe onde

há a computação da iluminação e operações de pós processamento.

Rasterização é um paradigma de renderização em tempo real qual opera em um *stream* estruturado de dados de uma cena. Vértices e seus atributos são processados em paralelo, projetados em espaço de tela e fragmentos de regiões internas interpoladas das primitivas formadas são rasterizadas e combinadas para compor os *pixels* da tela. Operando em cada primitiva de forma separada, o algoritmo *raster* é capaz de aproveitar a coerência espacial de dados e a ele é concedido alto poder de paralelismo. Porém, sendo incapaz de ter acesso ao resto da geometria da cena, seu uso é usualmente limitado apenas à computação de iluminação direta.

Ray tracing, em contrapartida, opera em um *stream* de raios. *Path tracing* é um algoritmo de *ray tracing* estocástico. Nele, um conjunto de raios, parametrizados em função de cada *pixel* na tela, são lançados da posição da câmera para interceptar a geometria da cena, passando por suscetivas reflexões de maneira recursiva finita, resultando em um dados sobre radiância qual são combinados de volta para os *pixels* da tela. Após a computação de um grande número de raios, a estimativa de radiância é esperada convergir em valores quais representam a medida de **global illumination (GI)** da cena. A *pipeline* de renderização desse trabalho usa dados de entrada do *raster* para a execução de um *path tracer*. Raios são gerados de acordo com dados contidos no *g-buffer* e testados contra a geometria da cena. Dados resultantes da amostragem de radiância ao redor do ponto são combinados de volta à cor do *pixel*.

(MATTAUSCH *et al.*, 2015) usa métodos baseados em rasterização para resolver o problema de visibilidade de raios primários, eles são gerados e compactados em um *buffer* de tamanho proporcional à resolução de tela. (DAVIDOVIČ *et al.*, 2012) demonstra como resultados obtidos pelo processo de rasterização podem ser análogos ao processo de *ray tracing* de raios primários de origem comum, e também descreve métodos para *viewports* não planares e até suporte para *antialisasing*. (LARSEN *et al.*, 2015) faz uso do *g-buffer* da *pipeline* de *deferred rendering* como dado de entrada para a sua *framework* de ray tracing. Esse trabalho explora a correspondência existente rasterização e *ray tracing* de raios primários, aproveitando ao máximo da aceleração gráfica da *pipeline raster* para a geração do *g-buffer* contendo informação geométrica e de material da cena em espaço de tela, sendo usado como entrada para os próximos passes do algoritmo de *ray tracing*.

3 METODOLOGIA

3.1 OBBs

OBBs são usadas como unidades de particionamento espacial e são parte integral de uma BVH. Cada nó da árvore têm seu domínio espacial delimitado de forma justa pelo volume de uma respectiva OBB. O teste de interseção de raios contra esses volumes produz uma estimativa conservadora da possibilidade de interseção do raio com a geometria da malha subjacente.

Um método de construção da hierarquia de volumes de uma árvore requer dados de vértices das primitivas para fins de computar a OBB de um grupo de primitivas. Esse passo pode ser realizado integralmente *offline*, ou como um passe de pré-processamento anterior à execução do laço principal em uma aplicação de renderização iterativa.

Uma característica fundamental da implementação proposta permite que transformações lineares dadas a objetos de uma cena possam ser aplicadas diretamente à OBB durante a execução, dessa maneira evitando a necessidade de acesso a dados de vértice novamente.

3.1.1 Representando OBBs

Em contraste à AABBs, quais possuem em sua representação um cubo unitário centrado na origem, em relação a posição e escala de um objeto usado como ponto de referência, uma OBB é um paralelepípedo de orientação arbitrária. Em outras palavras, uma OBB pode ser vista como um cubo, similar à AABB, com a vantagem que a permite assumir qualquer orientação arbitrária espacial, podendo ser ter sido esticada, rodada e transladada no espaço.

Sua representação geralmente recorre à três escalares, um ponto e uma base ortonormal (três vetores unitários perpendiculares uns aos outros). Esses atributos representam o meio comprimento de cada dimensão de arestas, o seu centro e a rotação da OBB, respectivamente. Na pratica, esses atributos são tipicamente empacotados em uma variável e atribuída a um objeto ou a uma estrutura de dados usada por um programa de computador.

Ao invés de armazenar cada um desses valores separadamente, observamos que as operações apresentadas pertencem a classe de transformações lineares. Sendo assim, podem ser representadas por uma única matriz 4×4. A matriz de transformação resultante M é dada pela operação cumulativa das transformações $M = TRS$, na ordem de S, R, e por último T, sendo matrizes de escala, rotação e transformação, respectivamente. A escolha de representação da OBB de forma matricial e suas propriedades emergentes se torna conveniente quando consideramos a

integração de OBBs em *pipeline* de renderização existentes. Na seção abaixo é apresentado uma forma de extração dessas transformações de uma malha.

3.1.2 Construindo e Atualizando OBBs

A construção e atualização de OBBs são dois processos distintos que são executados com base em entradas diferentes. Esses processos representam diferentes algoritmos e rodam em diferentes etapas da *pipeline* de renderização.

É traçado um paralelo ao método usado pela *pipeline* de renderização onde é realizado o carregamento e instanciamento de geometria. Em um primeiro momento existe um processo onde dados de primitivas geométricas são armazenadas em espaço de memória da GPU em um *buffer* de tamanho fixo. E então, fazendo parte do processo de renderização, iterativamente, esses dados na memória são usados para instanciar a geometria correspondente na cena.

Em um caso comum envolvendo vários objetos em movimento, primeiramente é construída a OBB para cada malha *offline*. Iterativamente, a OBB de cada instância de malha é atualizada antes do processo de renderização de cada quadro. Objetos deformantes, por sua vez, exigem uma total reconstrução da OBB.

Com relação aos dados requeridos para a construção de OBBs de uma malha, a entrada do algoritmo, é usado o conjunto de vértices que existem em coordenada de referência local pertencente à este conjunto. Consequentemente, uma OBB possui uma correspondência com o sistema de coordenadas de referência de sua malha.

Em contrapartida, uma instância de malha é uma cópia (lógica) de uma malha com seu devido posicionamento na cena (espaço de mundo), descrito por uma matriz de transformação local para mundo. Dessa forma, a OBB de cada instância é representada em coordenadas de mundo, com sua matriz de transformação resultante da associação entre matriz da OBB da malha e a matriz de transformação da instância.

A construção e atualização das OBBs portanto requerem dois passos: Um primeiro, *offline*, onde uma OBB local é derivada de o conjunto de pontos de uma malha em espaço local; Em seguida, seguindo modificações à cena, uma OBB global para cada instância é computada a partir da OBB local da malha associada e a matriz de transformação da instância. A tabela 2 sumariza as diferenças entre os dois processos.

Tabela 2 – Sumário das diferenças entre os processos de construção e atualização.

Processo	**Construção das OBBs**	**Atualização das OBBs**
Operação	Gather	Map
Tamanho de saída	proporcional ao numero de malhas	proporcional ao número de instâncias
Tamanho de entrada	Vértices da malha	OBB da malha e a matriz de transformação da instância
Saída	OBB da malha	OBB da instância
Etapa de renderização	*offline*	após cada quadro de animação

3.1.3 Construindo OBBs

Dando continuação à definição da matriz de transformação M usada para representar as transformações cumulativas que levam do cubo unitário da AABB para nossa OBB de orientação arbitrária, nesta seção é descrito como computar a escala, rotação e translação que compõem M.

É empregado o método proposto por (GOTTSCHALK, 2000) para o cálculo da OBB justa de um conjunto de pontos usando *Principal Component Analysis (PCA)*. o método roda em tempo linear ao número de vértices de uma malha. O processo de construção de uma OBB seguindo esse método envolve na computação da matriz de covariância do conjunto de pontos da malha. A rotação é dada pelos autovalores dessa matriz, a escala é computada pelos pontos de mínima e máxima da malha transformada pelo inverso da rotação e a translação é computada levando em consideração a ponto de centro da malha e o valor da média amostral computada junto com a matriz de covariância.

Seja s um vetor contendo a escala do meio comprimento da caixa, U sua base ortonormal e p seu centro. Para um conjunto de pontos V, V_i cada ponto, primeiramente é computada a matriz de covariância $C, m = cov(V)$ do conjunto de pontos, resultando também no vetor m da média amostral do conjunto. C é uma matriz simétrica e seus autovalores compõem uma base ortogonal. A essa altura já é possível calcular $U = eigs(C)$ como sendo os autovetores de C. m será usado para a computação de p à frente.

O próximo passo, para se descobrir s, seguindo a ideia de transformações suscetivas codificadas em M, com a operação de escala vindo antes da rotação pela definição dada anteriormente, V é levado para o sistema de coordenadas antes das operações de escala e rotação, onde a caixa delimitadora do conjunto $V^{'} = U^{t}(V - m)$ está alinhada com os eixos canônicos. É

computado $s = v'_{max} - v'_{min}$ como o comprimento da AABB de V'. Em síntese:

$$V' = U^t(V - m)$$
$$v'_{min} = min(V')$$
$$v'_{max} = max(V')$$
$$v'_{center} = (v'_{min} + v'_{max}) * 0.5$$

Onde:

- v'_{min}, v'_{max} são respectivamente os pontos mínimo e máximo em termos de coordenada do conjunto V'.
- v'_{center} é o ponto no centro de V'.

Finalmente, é computado $p = m + Uv'_{center}$. Com s, U, p é possível construir as matrizes de transformação S, R, T e matriz $M = TRS$ da OBB da malha.

$$S = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_x & 0 & 0 \\ 0 & 0 & s_y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R = \begin{bmatrix} & U & & 0 \\ & & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad T = \begin{bmatrix} 0 & 0 & 0 & p_x \\ 0 & 0 & 0 & p_y \\ 0 & 0 & 0 & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad M_{OBBmalha} = TRS$$

3.1.4 Atualizando OBBs

$M_i = M_o M_m$ atualiza a OBB M_i de uma instância i dada uma matriz de transformação M_o da instância e a OBB M_m da malha m correspondente à instância.

A atualização da OBB equivale a uma simples multiplicação de matrizes para cada objeto em cena. Tal computação é eficiente o suficiente para ser computada para cada quadro, no caso de cenas dinâmicas, ou apenas sempre quando a matriz de transformação da instância de uma malha é atualizada.

Essa operação é realizada durante a construção da BVH das instâncias, no nível de cima. Em uma animação onde a geometria da cena não tenha sido deformada, o nível de baixo da BVH de dois níveis não precisa ser atualizado. A BVH do nível de cima é reconstruída usando a M_i da instância onde M_o tenha sido atualizada. No caso em que a geometria tenha sofrido uma deformação linear M_d, a OBB da geometria é atualizada $M_m = M_m * M_d$ o que leva novamente a atualização da M_i e a reconstrução da BVH do nível de cima.

Apenas quando a animação de uma cena leva a deformação não linear da geometria é causa para a completa reconstrução da BVH de dois níveis. As BVHs de cada malha deformada são reconstruídas junto com a BVH das instâncias.

3.2 Algoritmo de Interseção Entre Raio-OBB

É observado que a interseção entre um raio e uma OBB pode ser simplificada à interseção de um raio transformado e uma AABB. Mudanças de coordenadas de referência foram discutidas nas sessões anteriores, e também foi demonstrada a computação da matriz que representa a OBB da instância de uma malha, esta que codifica a transformação completa que leva uma ABB em formato de um cubo unitário para uma OBB de orientação arbitraria. Consequentemente, aplicando essa transformação inversa ao raio, ele será levado de volta ao sistema de coordenadas de referência da AABB. Representações paramétricas de interseção nesse espaço se mantêm válidas devido às propriedades fundamentais de transformações lineares.

Esse tipo de operação já tem sido abordada em computação gráfica. No trabalho de (BARR, 1987) a inversa da transformação local-para-global foi usada a fins de modificar os raios para a computação de primitivas sólidas deformáveis. O trabalho de (MAIA *et al.*, 2006) também cita os benefícios de usar transformações de raios como meio para simplificar algoritmos de interseções. (BENTHIN *et al.*, 2017) se posiciona a favor do uso da transformação de raios à beneficio de animação de corpos rígidos baseados em instâncias e BVH de dois níveis.

Desse modo é recorrida à transformação de Lorentz, citada na bibliografia de Física, cobrindo transformações lineares entre sistemas de coordenadas diferentes. O grupo de Lorentz pode ser representado por matrizes 4×4, referenciadas como Λ, e transforma vetores x para $x' = \Lambda x$ onde x' pertence ao novo sistema de coordenadas. Em nosso contexto, é aplicada uma transformação afim aos vetores componentes do raio em forma paramétrica; origem o_{world} e direção d_{world}, em coordenadas homogenias.

$$o_{AABB} = (\ M^{-1}(o_{world}, 1))$$

$$d_{AABB} = (\ M^{-1}(d_{world}, 0))$$

$$o_{AABB}.xyz /= \ o_{AABB}.w$$

Onde:

- o_{AABB} e d_{AABB} são os componentes do raio transformado no espaço do cubo unitário da AABB.

- M é a matriz da OBB da instância da malha.

A matriz M computada da forma descrita na sessão anterior representa uma transformação local para global. Ou seja, do espaço da AABB unitária centrada na origem para a OBB com vertices em espaço de mundo. Com o objetivo de transformar o raio, dado em espaço global, para o espaço local, é aplicada a transformação afim com o inverso M^{-1} da matrix M.

Solucionar a interseção do raio-objeto equivale à solução da equação de equivalência entre as equações paramétricas do raio e do objeto e resolvendo t. Um t não negativo indica a existência de uma interseção na direção positiva do raio. O algoritmo de recorte de lajes, proposto pela primeira vez por (KAY; KAJIYA, 1986) e suas variantes (WILLIAMS *et al.*, 2005) e (MAJERCIK *et al.*, 2018), são usados na bibliografia de computação gráfica para computar a interseção entre raios e AABBs. O algoritmo a seguir apresenta ramificação mínima:

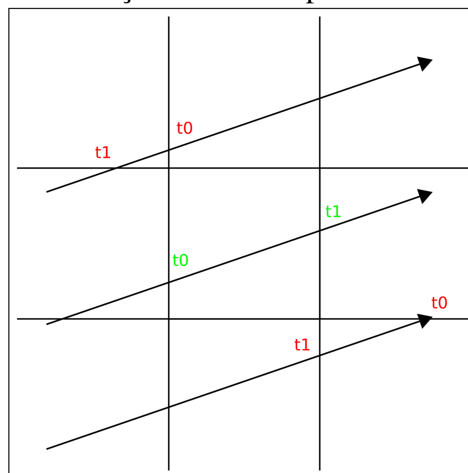
$$t_0 = max(min(\frac{p_{min_i} - o_{AABBi}}{d_{AABBi}}, \frac{p_{max_i} - o_{AABBi}}{d_{AABBi}}))$$
$$t_1 = min(max(\frac{p_{min_i} - o_{AABBi}}{d_{AABBi}}, \frac{p_{max_i} - o_{AABBi}}{d_{AABBi}}))$$

Onde:

- $i \in (x, y, z)$.
- p_{min} e p_{max} são respectivamente os vértices mínimos e máximos da AABB.

A interseção existe se $t_0 \leq t_1$ e $t_1 \geq 0$. O gráfico 1 mostra uma visualização da lógica por trás do algoritmo de lajes.

Figura 1 – Visualização do algoritmo de lajes em 2D. Pontos de interseção são computados para cada hiperplano. A interseção é definida pela ordem dos pontos de interseção.



As funções min, max executam uma operação de máximo/mínimo em termos de coordenada. Implementado dessa forma, existe interseção quando a condição acima for válida

e $t_0 o.w$ nos dá a distância assinalada do raio com a OBB, válida quando a origem do raio se encontra fora da caixa. Vale observar que a execução desse algoritmo em um computador depende do comportamento padrão de ponto flutuante quando $d_i = 0$.

Para o propósito $p_{min} = (-0.5, -0,5, -0.5)$ e $p_{max} = (0.5, 0.5, 0.5)$ são constante, uma vez que foi definido a AABB neste trabalho como um cubo unitário centrado na origem.

3.3 BVH

Esse trabalho implementa LBVH (LAUTERBACH *et al.*, 2009) buscando maximizar o paralelismo na construção e percurso da BVH. Dentre as motivações que levaram à escolha de LBVH estão a característica modular de construção da estrutura, o que nos facilita a combinação de aprimoramentos à técnica original e a sua adoção ao modelo de paradigma de dados paralelo que facilita a sua implementação em GPUs.

LBVH aborda a construção de sua BVH como uma redução a um problema de ordenação. O método de (KARRAS, 2012a) descreve a construção da hierarquia dividida em 4 passes:

- O método começa com a codificação das primitivas geométricas de um conjunto por meio da computação de *Morton Codes*, que tenta reduzir a informação espacial de uma primitiva a uma dimensão, gerando um vetor de códigos.
- No segundo passe há o ordenamento desse vetor com base no valor do código atrelado a cada primitiva.
- No terceiro passe, a árvore binária é construída a partir do vetor de códigos ordenados e seus nós são armazenados em um outro vetor.
- No quarto e último passe, as BVs são computadas para cada nó da árvore.

Embora esse algoritmo possibilite a geração de uma BVH de forma paralela, a árvore gerada apresenta baixa qualidade. Razões citadas na apresentação de (KELLER *et al.*, 2019) argumentam que a divisão da geometria em uma grade regular não leva em consideração a distribuição da malha no espaço e o algoritmo ignora a forma da primitiva em sua heurística de subdivisão dos nós da árvore.

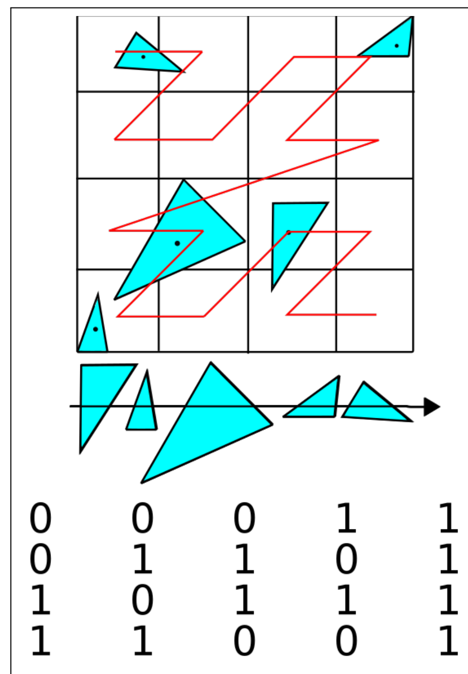
A seguir é explicado mais detalhadamente cada um desses passes para a construção da BVH. Em seguida, é mostrado um algoritmo de percurso da árvore proposto.

3.3.1 Computação de Morton Codes

A primeira parte da construção da BVH começa com a computação dos *Morton Codes*, também chamados de *z-order curve* pelo modo o qual a intercalação de coordenadas binárias geram um código de ordem numérica a apresentar a forma de uma curva em formato de "Z"recursiva. Desse modo, *Morton Codes* mapeiam dados multidimensionais a uma dimensão ao passo que preservam a propriedade de localidade desses pontos.

É considerado que a geometria do conjunto a ser computado existe dentro de um espaço cujo intervalo de cada coordenada pertencente a $[0, 1]$. Dentro desse cubo unitário cada coordenada é exibida como ponto fixo $0.x_1x_2x_3...$, $0.y_1y_2y_3...$, $0.z_1z_2z_3...$, onde cada x, y, z_i representa um bit. A intercalação dos *bits* de cada coordenada para formar um único número inteiro no formato $x_1y_1z_1x_2y_2z_2...$ gera o *Morton Code*. O gráfico 2 mostra o padrão de visitação de primitivas pelo seu *Morton code*.

Figura 2 – Visualização do padrão de visitação de cada célula da grade gerado pela ordenação de primitivas pelo seu *Morton code*.



Definindo uma grade regular de $2^k \times 2^2 \times 2^k$ no domínio do conjunto, pode-se representar a coordenada de cada célula usando k bits. Assim, um *Morton Code* de $3k$ bits é capaz de identificar uma primitiva em cada uma dessas células. Na prática, os *Morton Codes* usados têm o comprimento de 30 bits para que seja possível armazená-los em um vetor de variáveis inteiras de 32 bits e tirar proveito do padrão de alinhamento de dados de memória.

Dado um conjunto de primitivas, uma malha, por exemplo, visto que cada primitiva terá seu respectivo *Morton Code* computado, o passe de computação dos códigos pode ser executado em paralelo por uma operação de *map*. É usado o baricentro da primitiva como o valor representativo de sua localização espacial. O algoritmo a seguir, de (KARRAS, 2012c), é responsável por computar o código dado uma coordenada no intervalo de [0, 1].

```
1  uint expandBits ( uint v )
2  {
3    v = ( v *0 x00010001u ) & 0 xFF0000FFu ;
4    v = ( v *0 x00000101u ) & 0 x0F00F00Fu ;
5    v = ( v *0 x00000011u ) & 0 xC30C30C3u ;
6    v = ( v *0 x00000005u ) & 0 x49249249u ;
7    return v ;
8  }
9  uint mortonCode ( vec3 p )
10 {
11   p = min ( max ( p *1024 , vec3 (0) ) ,1023) ;
12   uint x = expandBits ( p .x ) ;
13   uint y = expandBits ( p .y ) ;
14   uint z = expandBits ( p .z ) ;
15   return 4* x + 2* y + z ;
16 }
```

3.3.2 Ordenação dos Morton Codes

A segunda parte para a construção da BVH executa um algoritmo de ordenação com base no valor do código atrelado a cada primitiva computado no passo anterior e armazenado em um vetor, gerando um novo vetor contendo os códigos ordenados.

É lembrado que o *Morton Code* atrelado à primitiva representa a sua posição espacial em uma célula da grade ocupando o domínio do conjunto. Os bits do código descrevem o caminho singular percorrido do nó raiz até a folha. A intenção da ordenação é simplificar o problema de geração de BVH primeiro escolhendo a ordem a qual as folhas estarão dispostas na árvore. Ao iterar sobre vetor de códigos ordenados estamos seguindo a *z-order curve*. Nessas

condições, as primitivas correspondentes também estarão ordenadas de forma espacialmente coerente.

O algoritmo de ordenação de (SATISH *et al.*, 2009) é usado no trabalho original de LBVH (LAUTERBACH *et al.*, 2009), já a contribuição de (KARRAS, 2012a) faz uso de (MERRILL; GRIMSHAW, 2010). Ambos algoritmos de ordenação paralela em GPU fazem uso de *radix sort*. A especialização da arquitetura da GPU para o processamento paralelo de *stream* estruturado de dados torna *radix sort* a abordagem mais rápida para o processamento de inteiros de 32-bits na GPU (MERRILL; GRIMSHAW, 2011). Diferentes *designs* de *radix sort* para a GPU buscam descrever o método na forma de primitivas de dados paralelos, porém pelas limitações inerentes ao natureza do hardware paralelo, todo trabalho que tenta propor seu método para ordenação paralela em arquiteturas multi processador têm que apresentar soluções não triviais para problemas como a sincronização e barramento de memória.

O desenvolvimento de um algoritmo de *radix sort* paralelo é complexo e foge to escopo deste trabalho. Em vez disso, é usado um algoritmo de ordenação na CPU.

3.3.3 Construção da Árvore Binária

O próximo passo é responsável pela construção da estrutura dos nós da árvore binária com base apenas no vetor de códigos ordenados do passo anterior. É executada uma operação de mapeamento onde a associação de parentescos para cada nó é computada em paralelo resultando em um vetor de nós.

(KARRAS, 2012a) propõe um algoritmo paralelo para a construção da árvore a partir do vetor de *Morton codes* ordenado. A escolha de um tipo especifico de árvore possibilita o processamento de cada nó de maneira independente.

A estrutura da árvore é a versão compacta de uma árvore de prefixos binários ordenada, também chamada de árvore PATRICIA, e tem como características;

- A árvore é uma representação hierárquica de um vetor associativo de prefixos representados em código binário.
- Nesse tipo de árvore, as chaves não são explicitamente armazenadas no nó. A posição de um nó na árvore define a sua chave.
- Não existe nó com apenas um filho. Cada nó possui dois filhos, ou é uma folha. Neste trabalho é feita essa distinção referindo-os como nós internos e folhas.
- Os filhos têm um prefixo comum à cadeia associada ao nó. Cada nó interno corresponde a

cadeia de prefixos mais longa compartilhada pelas folhas pertencente à subárvore.

- O número de nós internos não ultrapassa o número de folhas. Seja n o número de folhas, a árvore possui $n-1$ nós internos.

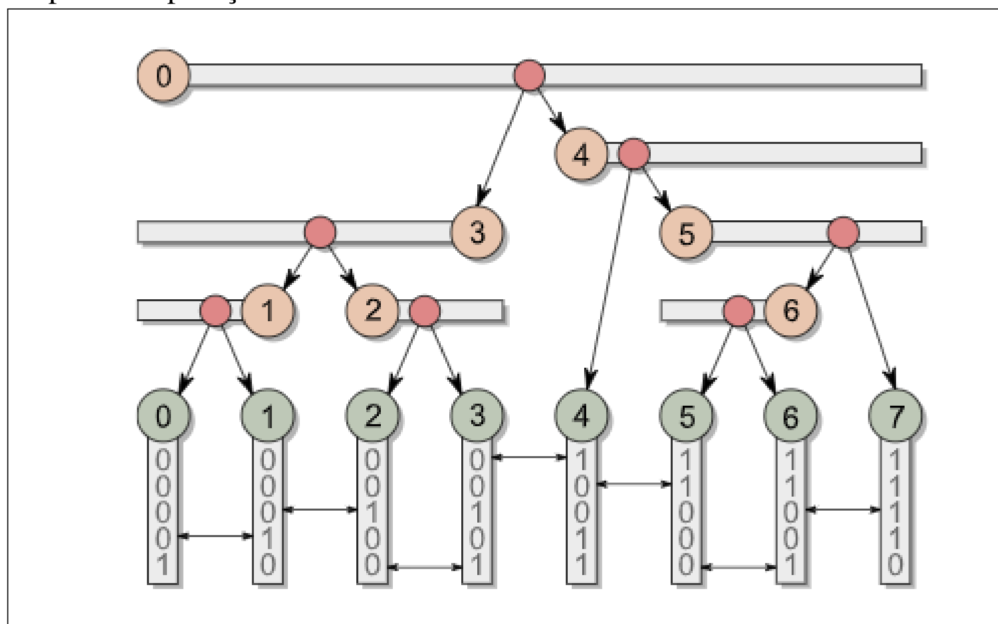
É estabelecido o vetor de códigos ordenados como folhas. Tendo as folhas como ponto de partida, um algoritmo simples construiria a árvore de baixo para cima de forma recursiva. A solução proposta por (KARRAS, 2012a) processa cada nó de forma independente. Nos próximos parágrafos é apresentada a intuição por trás do algoritmo.

Tendo como ponto de partida o vetor de códigos ordenados como folhas, resta computar os nós internos. Uma vez que a árvore está em ordem lexicográfica, cada nó interno possui uma cadeia de prefixos comum.

Como cada nó interno possui dois filhos, é esperada que essa sequencia de códigos seja particionada em duas. Essa partição acontece na posição em que o primeiro bit após o prefixo comum da sequência seja diferente.

Atribuindo o índice de cada nó interno de acordo com as posições dos dois lados da partição faz com que o índice do nó interno corresponda com uma das extremidades de sua cadeia de prefixos associada. A figura 3 ilustra os elementos da árvore binária discutidos.

Figura 3 – Distribuição dos nós na árvore binária, mostrando a correspondência de cada nó interno pelo alinhamento vertical com a folha de mesmo índice. A barra horizontal representa a cadeia de prefixos pertencente ao nó. O círculo vermelho representa o ponto de partição da cadeia em dois subintervalos.



Fonte: (KARRAS, 2012a)

A árvore é construída pela associação de parentesco entre nós internos. Para cada nó

interno, é preciso encontrar o começo e fim da cadeia de prefixos comum associada. Em seguida, é encontrado o ponto de particionamento da sequência de códigos, daí os índices dos filhos já podem ser computados. A seguir, o algoritmo é descrito novamente em detalhe.

Reiterando o que foi dito anteriormente, a cadeia de prefixos comum pertencente a um nó segue a seguinte formulação matemática; Seja $[i, j]$ o intervalo da cadeia associada a um nó e $\delta(i, j)$ o comprimento do prefixo comum entre os códigos k_i e k_j, a ordenação dos códigos implica $\delta(i', j') \geq \delta(i, j)$ para $i', j' \in [i, j]$ e $\delta(i'', j'') < \delta(i, j)$ para $i'', j'' \notin [i, j]$.

O índice $\gamma \in [i, j-1]$ particiona a cadeia nos dois subintervalos $[i, \gamma]$ e $[\gamma+1, j]$ de modo que $\delta(\gamma, \gamma+1) = \delta(i, j)$. Seja o vetor de nós internos I e folhas L, o nó filho da esquerda será I_γ se o intervalo $[\gamma+1, j]$ cobrir mais de um código ou L_γ se cobrir apenas um código, sendo uma folha. Uma atribuição similar é dada para o nó filho da direita, $I_{\gamma+1}$ ou $L_{\gamma+1}$.

Para a construção da árvore, é preciso primeiramente determinar o intervalo da cadeia associada a um nó $i \in [0, n-1]$. Pelas formulações anteriores, o índice i define uma das extremidades do intervalo e portanto o código k_i pertence a cadeia. Tendo que os índices de fora do intervalo possuem o comprimento do prefixo comum menor que os indices de dentro do intervalo, é consultada as mediações k_{i-1}, k_i, k_{i+1} pelo maior prefixo comum. É computada a direção $d = \pm 1$ tal que $\delta(i, i+d) > \delta(i, i-d)$, ou seja, $d = +1$ aponta a direção de um intervalo começando em i e $d = -1$ a um intervalo terminando em i.

É realizada uma busca binária exponencial na direção d buscando o maior $l \in [0, l_{max}-1]$ que satisfaz $\delta(i, i+ld) > \delta_{min}, \delta_{min} = \delta(i, i-d)$. O limite superior l_{max} é computado pela busca exponencial partindo de 2 satisfazendo a inequação. A busca binária itera em $[\frac{l_{max}}{2}, \frac{l_{max}}{4}, ..., 1]$. Assim, encontramos a outra extremidade do intervalo $j = i + ld$.

É feita outra busca binária exponencial na direção d buscando $s \in [0, l-1]$ que satisfaz $\delta(i, i+sd) > \delta_{node}, \delta_{node} = \delta(i, j)$, iterando em $[\lceil \frac{l}{2} \rceil, \lceil \frac{l}{4} \rceil, ..., 1]$. Finalmente, é encontrado o índice de partição $\gamma = i + sd + min(d, 0)$.

Dados i, j e γ, os filhos da esquerda e direita têm suas cadeias associadas $[min(i, j), \gamma]$ e $[\gamma+1, max(i, j)]$, respectivamente. Se o tamanho do intervalo for 1, o filho é uma folha. Senão, um nó interno. Em outras palavras:

$$\text{filho da esquerda} = \begin{cases} I_\gamma, & \text{se } \gamma \neq min(i, j) \\ L_\gamma, & \text{se } \gamma = min(i, j) \end{cases} \quad \text{filho da direita} = \begin{cases} I_{\gamma+1}, & \text{se } \gamma+1 \neq max(i, j) \\ L_{\gamma+1}, & \text{se } \gamma+1 = max(i, j) \end{cases}$$

Idealmente, cada primitiva é associada a um *Morton code* único. Na prática, existem

casos em que a resolução de 30 *bits* da grade que particiona o conjunto de primitivas não é suficiente para evitar que múltiplas primitivas ocupem a mesma célula. Uma modificação da função $\delta'(i, j)$ para computar o comprimento do prefixo comum das concatenações $k_i \oplus i$ e $k_j \oplus j$ basta para incluir o caso de códigos duplicados.

O código a seguir implementa o algoritmo descrito:

```
1  int commonPrefixLength ( int i , int j )
2  {
3      if (i < 0 || j < 0 || i >= codes . size () || j >= codes . size
           () ) return -1;
4      uint32_t codeXor = codes [ i ]. code ^ codes [ j ]. code ;
5      return ( codeXor )? __builtin_clz ( codeXor ) :32+ __builtin_clz (
           i ^ j );
6  }
7  int binarySearchFloor ( int cmp , int i , int n , int d )
8  {
9      int j = 0;
10     do {
11         n /= 2;
12         if ( commonPrefixLength (i , i +( j + n) * d ) > cmp ) j += n ;
13     } while ( n > 1) ;
14     return j ;
15 }
16 int binarySearchCeil ( int cmp , int i , int n , int d )
17 {
18     int j = 0;
19     do {
20         n = ( n + 1) >> 1;
21         if ( commonPrefixLength (i , i +( j + n) * d ) > cmp ) j += n ;
22     } while ( n > 1) ;
23     return j ;
24 }
25 void findRange ( int i , int & left , int & right , int & direction )
```

```
26 {
27     direction       =  ( commonPrefixLength (i ,i -1)         >
           commonPrefixLength (i , i +1) ) ? -1:1;
28
29     int    prefixMin      =  commonPrefixLength (i ,i - direction ) ;
30     int    rangeMax  =  2;
31     while ( commonPrefixLength (i , i + rangeMax * direction )          >
           prefixMin )      rangeMax   *=   2;
32
33     int    range   =  binarySearchFloor ( prefixMin ,i , rangeMax ,
           direction );
34
35     left       =  ( direction >0) ?i :i - range ;
36     right     =  ( direction >0) ?i + range : i ;
37 }
38 void    buildNode ( int      i )
39 {
40     int    l ,r , d;
41     findRange (i ,l ,r ,d ) ;
42
43     int    prefixCommon    =  commonPrefixLength (l , r ) ;
44     int    split      =  i    +  binarySearchCeil ( prefixCommon ,i ,r -l , d ) * d +
           fmin (d ,0) ;
45
46         int    childL     =  ( split          ==   l ) ? - codes [ split        ]. index : split
               ;
47         int    childR     =  ( split +1      ==   r ) ? - codes [ split +1]. index : split
               +1;
48
49     nodes [ i ]    =  { childL , childR };
50 }
```

3.3.4 *Computação dos BVs*

O ultimo passo no processo de construção da BVH é a computação dos BVs a partir da estrutura de nós gerada no passo anterior e as primitivas geométricas indexadas pelas folhas da árvore. O algoritmo é executado para cada caminho a partir da folha até o nó raiz de forma paralela, apresentando um nível de granularidade grosso e a necessidade de sincronização de memória. O resultado desse passe gera uma lista contendo estruturas de BVs de tamanho e indexação de acordo com a lista de nós.

Os nós da árvore providenciam uma forma de acesso eficiente às primitivas geométricas da cena de modo a acelerar operações de renderização, porém sozinhos não são suficiente para se realizar operações de busca. É necessário um dado atrelado a cada nó que oriente a suo percurso. A natureza do uso da árvore dita o tipo de dados atrelados ao nó. No contexto deste trabalho, a arvore representa o particionamento espacial da cena e serve como suporte para acelerar operações de interseção de raios com a geometria da cena. Desse modo, cada nó da árvore necessita conter informação sobre o domínio espacial delimitado pela sua partição.

Reiterando as características de árvore usada neste trabalho, cada nó está atrelado a uma cadeia de prefixos comum, que por sua vez representa uma partição do domínio espacial do conjunto geométrico. Cada nó interno, um nó que não é folha, tem dois filhos. Os filhos compartilham o prefixo comum de seu pai, e entre si expandem a cadeia de prefixos, sub particionando o espaço recursivamente até chegar à folha.

O BV de cada nó deve representar o domínio do conjunto geométrico pertencente a sua cadeia de prefixos comum. Com o auxílio de uma lista associativa entre os *Morton codes* e o índice de sua primitiva, é possível computar o BV de forma paralela para cada nó através de uma operação de compactação, porém tal computação não possui um desempenho que a torne viável na prática, uma vez que o hardware é limitado por banda de memória.

O algoritmo original proposto por (LAUTERBACH *et al.*, 2009) gera os BVs de cada nível da árvore em sequência, de baixo para cima, aproveitando os BVs dos filhos gerados pelo passo anterior, computando o BV do nó do próximo nível como a união dos BVs de seus filhos. Dessa maneira a banda de memória é minimizada, porém é introduzido uma barreira de sincronização global de memória para cada nível. Esse algoritmo também apresenta um baixo nível de ocupação. Para cada próximo nível computado, metade do número de proce

Intended to be implemented in GPU hardware SIMD acceleration No branching Supports affine transformations Based on ray transformations Uses slabs algorithm [reference]

OBB encoded into a 4x4 transformation matrix Supports degenerative cases

Returns intersection Returns distance to OBB (if ray is outside the box) TODO returns surface normals TODO (future work) returns face hit TODO (future work) surface parametrizationssos serão chamados. Nos níveis superiores, um numero pequeno de processo estarão sendo executados, subutilizando a GPU.

(KARRAS, 2012a) adota uma abordagem diferente, onde cada caminho da árvore da folha até o nó raiz, em contraste à cada nó, é computado de forma paralela. O algoritmo traça o caminho de baixo para cima começando pela folha, subindo em cada nó pai, registrando quais nós são visitados usando contadores atômicos, e encerrando quando acessa um nó que não tenha sido visitado anteriormente. Esse algoritmo remove a necessidade de sincronização global de memoria porém ainda depende da sincronização dos contadores. Também existe a vantagem dessa abordagem poder ser executada em um único passe.

Enquanto esse algoritmo possui baixo nível de ocupação ele gera um alto nível de divergência de execução. Cada processo segue diferentes fluxos de execução e metade dos processos são encerrados após o processamento de cada nível, porém o autor argumenta que pela natureza do algoritmo ser limitada pela banda de memória disponível, minimizando sua divergência de execução teria um impacto negligenciável.

Neste trabalho é implementado o algoritmo de (KARRAS, 2012a). De inicio, um processo paralelo é executado para cada nó interno que é imediatamente seguido de uma folha, subindo cada nível da árvore iterativamente até o nó raiz. Para cada iteração, o BV do nó é computado pela união dos BVs dos filhos e armazenado em um novo vetor de BVs com a mesma indexação que o nó. Em seguida, o algoritmo prossegue para o índice referente ao pai e o contador referente a este índice é incrementado. Caso o contador indique que esse processo seja o primeiro a visitar esse novo nó, o processo é terminado. O código a seguir implementa o algoritmo descrito.

```
1 uint id = gl_GlobalInvocationID . x ;
2
3 int nodeId = nodesLeaf [ id ];
4 do {
5   writeVolume ( computeNodeVolume ( nodeId ) , nodeId ) ;
6
7   nodeId = nodesParent [ nodeId ];
```

```
8      const   Node  node  =  nodes [ nodeId ];
9      int   j  =  ( node . l  >  0  && node . r  >  0) ?1:0;
10     if ( atomicCounterIncrement ( nodeVisitCounter [ nodeId ])          !=   j )
           break ;
11   } while ( true ) ;
```

O algoritmo usa contadores atômicos para cada nó fim de evitar trabalho duplicado e de garantir que o processo terá a informação dos BVs de ambos os filhos escritas em memória antes de computar o BV do pai. Os valor nos contadores é usado como um sinal de que um processo que visitou aquele nó, subindo na hierarquia por outro caminho, foi o responsável de computar a BV do outro filho. O primeiro processo que acessa o nó é terminado, enquanto o segundo prossegue. Isso garante a computação do BV de cada nó uma única vez, não antes que os BVs de ambos os filhos tenham sido computados primeiro.

3.4 Travessia da BVH

Nessa sessão é apresentado um algoritmo de percurso da BVH destinado a uso em um *path tracer*. Em termos de paralelismo, o algoritmo a ser descrito executo percurso independente, a visitação de nós segue o padrão de profundidade, da direita para a esquerda e de forma iterativa por meio de uma pilha.

A BVH é construída com o intuito de servir como estrutura de aceleração para operações de interseção. O algoritmo de percurso apresentado nessa sessão realiza uma operação de busca na árvore e retorna uma lista contendo primitivas geométricas com potencial de interseção com o raio de entrada. Durante a percurso, testes de interseção com as BVs dos nós são usados como critério de descarte de partições da árvore. Nós a serem visitados são mantidos em uma pilha. O algoritmo consome a pilha de forma iterativa.

Dado como entrada um raio, a lista de nós e a lista de BVs da BVH, o algoritmo de percurso adiciona o nó raiz da árvore, I_0, na pilha. As seguintes operações são executadas de forma iterativa enquanto houverem nós na pilha; O último nó da pilha é removido e é realizado um teste de interseção entre o raio e o BV do nó. Existindo interseção, os dois filhos do nó são adicionados à pilha, ou, se os filhos forem folhas, elas são adicionadas à lista de primitivas geométricas a ser retornada pelo algoritmo.

Na prática, a restrição ao uso de *arrays* estáticos na GPU impõe limites sobre a

qualidade de nós que são possíveis de se manter na pilha e ao tamanho da lista de primitivas. Esse problema pode ser remediado implementando o paradigma de produtor consumidor. O código a seguir implementa o algoritmo descrito.

```
1  void traverse ( vec3 ro , vec3 rd )
2  {
3      do {
4          int nodeId = nodeStackPop () ;
5          if ( nodeId < 0) break ;
6
7          if (! isVolume ( ro , rd , getVolume ( nodeId ) )) continue ;
8
9          const Node node = getNode ( nodeId ) ;
10         if ( pushNodeLeaf ( node . l ) < 0 || pushNodeLeaf ( node . r ) <
               0) break ;
11     } while ( leafStackSize < MAX_LEAVES -2) ;
12 }
13 Fragment castRay ( vec3 ro , vec3 rd )
14 {
15     Fragment f = fNone ;
16
17     nodeStackSize = 0;
18     nodeStackPush (0) ;
19     do {
20         leafStackSize = 0;
21         traverse ( ro , rd ) ;
22
23         for ( int i =0; i < leafStackSize ; ++ i )
24             f = opU (f , isTriangle ( ro ,rd , leafStack [ i ]) ) ;
25     } while ( leafStackSize > 0) ;
26
27     return f ;
28 }
```

Em um algoritmo de percurso recursivo, cada processo percorre a árvore de forma independente, a decisão de descartar um nó ou descer na subárvore é feita de forma independente por cada processo e não há garantia que os processos permanecerão em sincronia após seguirem diferentes caminhos. Implementando a percurso iterativa, gerenciando a pilha de recursão explicitamente, faz com que todos os processos executem as mesmas operações, em sincronia, mesmo visitando regiões da árvore diferente. Assim, a divergência de execução do algoritmo é minimizada.

A percurso independente, porém, traz a desvantagem de alta divergência de memória. Em certo ponto, especialmente nos níveis mais baixos da árvore, é espera-se que cada processo esteja acessando diferentes partes da árvore, e o trabalho realizado por um processo não é acessível ao outro. A solução para esse problema não é trivial. A chave para a paralelização do percurso simultânea está na oportunidade de ordenar processos em um número suficiente de grupos de modo a saturar a GPU. Uma solução seria, por exemplo, agrupar processos que percorrem percurso similares.

Em conclusão, enquanto o percurso independente oferece um algoritmo simples e generalista, o algoritmo realiza operações redundantes e possui custo de memória elevado. A percurso simultânea, em contrapartida, tem o potencial de diminuição de divergência de memória, porém exige um algoritmo mais complexo e é limitado à casos especificas onde existe uma certa coerência na execução e uso de memória entre processos de um mesmo grupo.

3.5 União de Duas OBBs

Nas sessões anteriores foi apresentado um algoritmo para de computação das OBBs a partir de conjuntos de pontos e uma estratégia para a construção da árvore binária que junto com a ordem de visitação de nós. Agora, são discutidos diferentes algoritmos para a construção de OBBs em cascata, aproveitando os dados computados das OBBs dos níveis anteriores, computando a OBB de um nó como a união das OBBs dos dois filhos e apresentando níveis de desempenho variados que afetam tanto a velocidade do algoritmo de construção quanto a qualidade final da árvore.

São abordados três algoritmos. O primeiro envolve a computação da matriz de covariância a partir do conjunto de pontos pertencente à subárvore seguindo o algoritmo apresentado na seção anterior. O segundo algoritmo evita o acesso ao conjunto de pontos da subárvore, substituindo-os pelos 16 vértices das duas caixas das OBBs filhas. O terceiro algoritmo evita

computar PCA inteiramente, ele usa como heurística a combinação da parte rotacional das OBBs e computa as extremidades da caixa pela projeção de seus vértices nesse novo eixo.

O primeiro algoritmo computa a OBB por meio de PCA, método proposto por (GOTTSCHALK, 2000) e descrito na seção anterior sobre a computação de OBBs dado um conjunto de pontos. Ele é capaz de gerar a OBB de cada nó de forma independente, não precisando dados dos filhos, e gera OBBs de melhor qualidade em relação aos outros dois algoritmos.

A princípio, pode-se argumentar que tal método seja um bom candidato para um algoritmo paralelo, afinal sua execução é independente, é possível executar um algoritmo paralelo com um processo para cada nó, sem a necessidade de sincronização de memória e comunicação entre processos. Porém, esse algoritmo sofre com um alto uso de banda de memória. Cada processo precisa acessar todo o conjunto de pontos pertencente ao subconjunto atrelado ao nó. Todos os processos competem entre si por banda de memória. Esse problema é agravado a medida que são computados nós pertencentes aos níveis superiores, uma vez que os níveis mais altos da árvore possuem uma cadeia de prefixos maior.

Esse algoritmo também apresenta alta taxa de divergência, não só afetando a eficiência do uso da memória *cache* mas, como o tempo de execução do algoritmo é linear em relação ao número de pontos, ele também sofre com uma alta divergência de execução.

Tais fatores fazem que o algoritmo paralelo na verdade tenha seu tempo de execução limitado pelos tempo de execução dos processos nos níveis superiores da árvore que por sua vez não só consomem mais memória como também demoram mais tempo para terminar.

O segundo algoritmo, citado em (ERICSON, 2004), procura delimitação no uso de memória. Ele também computa a OBB por meio de PCA a partir de um conjunto de 16 pontos pré definidos, pertencentes aos vértices das duas caixas delimitada pelas OBBs de cada nó filho. Como o algoritmo necessita ter acesso às OBBs dos filhos, a sincronização de memória é necessária. O algoritmo de computação de BVs descrito na seção anterior segue cada caminho de baixo para cima forma paralela fornece a ordem e a sincronização necessária para a execução desse algoritmo de união de OBBs.

Esse algoritmo apresenta pouca divergência, uma vez que todos os processos usam um mesmo numero de pontos, e o uso de memória é limitado ao acesso das OBBs dos nós filhos. A qualidade da OBB gerada é muitas vezes inferior a do primeiro algoritmo por conta do baixo nível de precisão e artifícios gerados pelo método de PCA dado um número reduzido de pontos.

O terceiro algoritmo, também visto em (ERICSON, 2004), substitui a computação da OBB por PCA, que por sua vez possui um alto nível de complexidade, por um método mais simples. Ele propõe uma heurística alternativa ao PCA, sugerindo a orientação da OBB a partir da combinação da parte rotacional das OBBs dos filhos. As extremidades da OBB são determinadas pelos 16 vértices das OBBs projetados nesse novo eixo.

A parte rotacional da OBBs é extraída em forma de *quaternions* e a interpolação pela metade entre ambas rotações dá o novo eixo, o qual depois será convertido de volta a sua forma matricial.

Na prática, é observado que o método comum de conversão entre *quaternions* e matrizes usando a transformação de Cayley é instável. O método iterativo de (MüLLER *et al.*, 2016) é usado para extrair a parte rotacional da matriz em forma de *quaternion* com resultados satisfatórios, sendo capaz de lidar com reflexões e matrizes degeneradas.

Tendo a matriz de rotação, são computadas a escala e rotação de forma similar à descrita na seção anterior. Esse algoritmo não precisa gerar a matriz de covariância nem extrair seus autovetores, reduzindo a sua carga computacional. A qualidade da árvore, porém, é de nível inferior aos demais algoritmos apresentados.

O método de computação das OBBs proposto neste trabalho combina os três algoritmos. Nos níveis mais baixos da árvore, onde nós são imediatamente seguidos de folhas, a OBB é computada seguindo o primeiro algoritmo. A medida que o algoritmo sobe a árvore, se um nó interno possuir ao menos um nó folha, é utilizado o segundo algoritmo, combinando tanto vértices da OBB quanto vértices da folha. Em níveis superiores da árvore, onde existem apenas nós internos, é usado o terceiro algoritmo a usando como entrada as duas OBBs dos filhos.

Todos os métodos apresentados estimam uma OBB justa ao redor da subárvore, mas existem casos em que a OBB gerada possa ser bem maior do que a ideal. A acumulação de OBBs 'frouxas' tem o potencial de aumentar o volume total da árvore, a soma do volume de todas as OBBs de cada nó, exponencialmente. O volume extra da OBB de um nó é propagado para o nível superior afetando criticamente a performance da árvore. Por essa razão, é muito importante a escolha da heurística correta para a construção das OBBs.

Esse problema é remediado parte computando também a AABB. Nos casos em que a AABB possui volume menor que o da OBB (podemos computar o volume por $abs(det(M_{obb,aabb}))$), a OBB é substituída pela AABB.

3.6 BVH de Dois Níveis

Uma hierarquia de dois níveis é uma solução elegante para o problema de representação hierárquica de cena. Ela cria uma separação entre *assets* e suas instâncias em cena e é compatível com a lógica de programação orientada a objetos e o fluxo de trabalho artístico.

É usado o algoritmo de LBVH para a construção das BVHs. Introduzido por (KARRAS, 2012a), o autor cita em seu trabalho sobre LBVH a ideia de remediar as deficiências da natureza de sua árvore por meio da construção de uma árvore hibrida, que produz um nível de cima de melhor qualidade para os nós próximos ao nó raiz, mantendo o nível de baixo da árvore, onde existe um número bem maior de nôs, sob a eficiência do algoritmo de LBVH.

(KELLER *et al.*, 2019), em sua apresentação sobre o estado da arte no desenvolvimento de estruturas de aceleração, fala sobre estratégias de implementação de BVH de dois níveis no *hardware* da RTX. O modo como fazer um construtor robusto ao agrupamento de instâncias ainda é um problema em aberto nessa área. (KELLER *et al.*, 2019) aponta que o agrupamento de instâncias baseado na localidade reduz a taxa de sobreposição de volumes em relação a demais critérios.

É optado nessa implementação pelo uso de LBVH para a construção dos dois níveis de nossa hierarquia. Porém, em vez de construir a BVH de cima abaixo usando a geometria da cena, no nível de cima é construída uma hierarquia para as instâncias, enquanto no nível de baixo é mantido o particionamento baseado em primitivas geométricas.

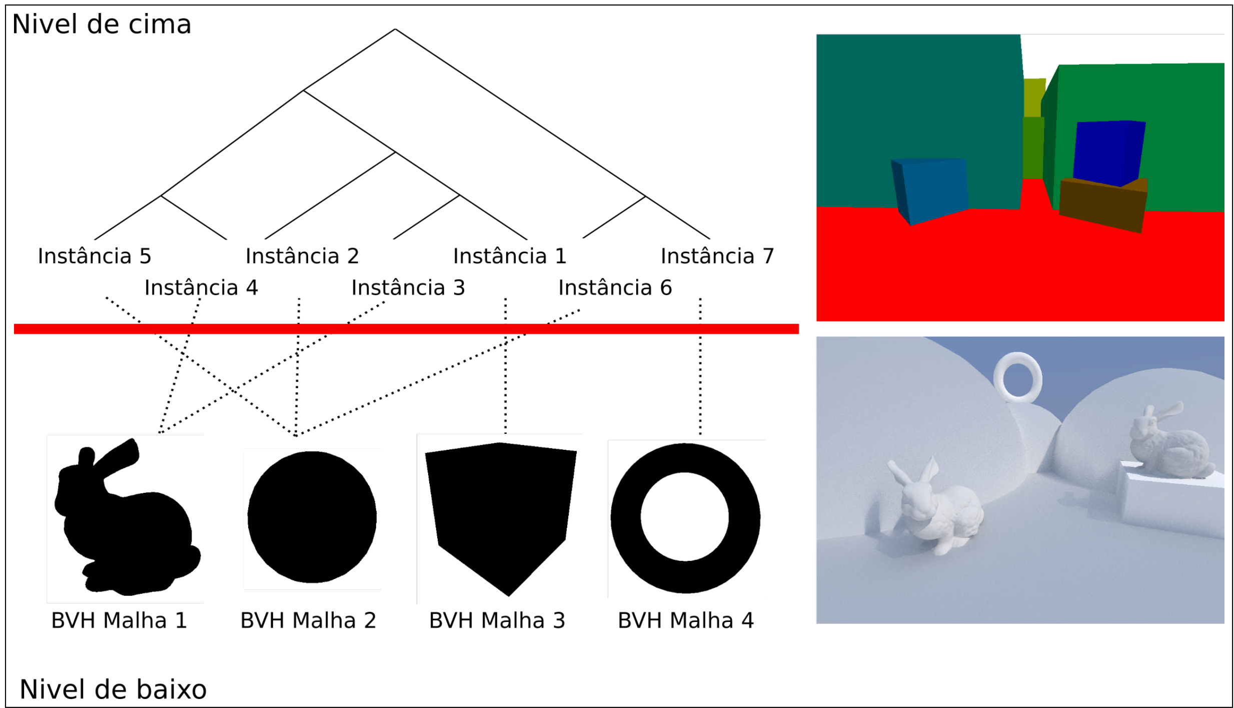
A heurística escolhida se baseia nas observações de (KELLER *et al.*, 2019). Desse modo, o nível de cima, o qual representa o particionamento de objetos volumétricos discretos, gera uma hierarquia de qualidade superior, dado como maior fator contribuidor a coerência espacial na distribuição das instâncias na cena. O particionamento que leva em conta apenas as primitivas geométricas, por sua vez, realizado no nível de baixo da hierarquia, é ignorante aos valores espaciais codificados pelo volume da instância.

O processo de construção da BVH de dois níveis começa pela computação de uma lista de BVHs correspondentes a cada malha. Em seguida, a BVH das instâncias é computada com base nessa lista.

O nível de baixo é composto pela lista de BVHs das malhas tendo as primitivas geométricas como folhas. O nível de cima é representado pela BVH das instâncias e possui como folhas cada instância. O gráfico 4 ilustra a BVH de dois níveis.

O processo de percurso da cena começa pela busca na BVH do nível de cima. Cada

Figura 4 – O gŕafico ilustra uma cena representada por uma BVH de dois níveis. No nível de baixo exite uma única BVH que particiona as instâncias. Instâncias fazem a malhas. No nivel de baixo existe uma lista de BVHs de malhas. O algorítmo de cima começa o percurso no nível de cima. Figura dee cima ilustra a interseção do raio com as instâncias. A figura de baixo mostra a cena renderizada.



folha atingida nesso percurso corresponde a interseção com uma instância. A percurso se estende ao nível de baixo, navegando pela BVH da malha correspondente a cada instância e retornando a lista de primitivas interceptadas.

Nas sessões abaixo é descrito o processo de construção e percurso da BVH de dois níveis.

3.6.1 Construção do Nível de Baixo

No nível de baixo da hierarquia é computado um vetor de BVHs de malhas a partir de suas primitivas. O algoritmo é capaz de processar cada malha em paralelo. A alocação de memória e a chamada de processos para trabalho em paralelo são lineares em função ao número de primitivas das malhas.

É esperado que primitivas de entrada para cada malha tenham suas coordenadas pertencentes ao intervalo [0, 1]. Como a intenção é manter suporte para malhas de dimensões arbitrárias, um passe de preprocessamento é inserido onde são calculadas as dimensões do domínio de cada malha e é computada uma transformação que mapeia os pontos da malha para o intervalo unitário.

Percorrendo os vértices da malha i, os pontos de minima e máxima p_{min}, p_{max} são computados. Essa operação pode ser realizada por meio de uma uma redução paralela. A transformação $G_i = S_i T_i$ mapeia os vértices da malha para o intervalo $[0, 1]$, onde S_i é a matriz que escala o vetor $max(abs(p_{max} - p_{min}))$ e T_i aplica uma translação $-p_{min}$.

O vetor G possui dimensão igual ao número de malhas. Um vetor C é criado para armazenar os pares de *Morton codes* e índices, de dimensão igual a soma do número de primitivas de todas as malhas. O vetor N, contendo os nós de todas as árvores das malhas, V, contendo os volumes, e demais *buffers*, todos têm dimensões iguais a $\#C$.

Os próximos passes para a construção da hierarquia seguem o método descrito na sessão anterior sobre LBVH. No primeiro passe, a computação de *Morton codes*, transforma o centroide da primitiva por G_i. O algoritmo escreve no vetor de nós N e volumes V aplicando os *offsets* necessários de modo a alinhar os índices de N e V ao início da sequência de primitivas da respectiva malhas. Em outras palavras N_0, V_0 representam respectivamente o nó raiz e volume da malha 0 de n primitivas, N_n, V_n para a malha 1 de tamanho n, N_{n+m}, V_{n+m} para a malha 2 de tamanho m, e assim por diante.

Graças a esse critério no alinhamento de memória nós, volumes e códigos compartilham o mesmo código. Assim, é possível executar o último passo, o cálculo de volumes dos nós, em uma única execução. É lembrado que o último passo na construção da árvore realiza umo percurso de baixo para cima, executando processos paralelos em função do número de nós-folha, nós seguidos de duas folhas. Devido ao número reduzido de processos, especialmente no caso de árvores desbalanceadas, normalmente esse passe se torna mais caro para execução paralela. Concatenando os conjuntos de nós-folha de todas as malhas em uma única chamada aumenta a possibilidade de saturação da GPU e o desempenho do algoritmo.

3.6.2 Construção do Nível de Cima

No nível de cima da hierarquia uma única BVH é computada a partir das instâncias da cena. A alocação de memória e a chamada de processos para o trabalho em paralelo são lineares em função do número de instâncias.

Como mencionado anteriormente, uma instância é uma entidade lógica, mas também possui uma representação geométrica. Uma instância é composta por uma matriz de transformação M_i e um índice i apontando para a malha correspondente.

Similarmente ao etapa de construção do nível de baixo, é preciso realizar um passe

de preprocessamento para mapear o conjunto de instâncias ao intervalo unitário. A matriz $I_i = M_i G_i^{-1}$ transforma os vértices de um cubo unitário em $(0,0,0)(1,1,1)$ de uma instância i no espaço de mundo. p_{min}, p_{max} é computado usando os vértices transformados de todas as instâncias e uma matriz H é gerada mapeando o domínio dos vértices das instâncias ao espaço unitário de forma igual a descrita na seção anterior.

Os vetores I, C, N, V de transformação das instâncias, códigos, nós e volumes, respetivamente, possuem dimensão igual ao número de instâncias.

Os próximos passes para construção da hierarquia seguem o método de LBVH sem *offsets*, porque apenas uma BVH é gerada e N_0, V_0 são o nó raiz e seu volume. Dessa vez, as primitivas geométricas usadas são os vértices dos volumes das instâncias. No primeiro passe, o cálculo dos códigos, o centroide de uma instância é dado por $HI_i * (0.5, 0.5, 0.5)$. No último passe, a computação dos volumes, os vértices das instâncias são os 16 vértices do cubo unitário $(0,0,0)(1,1,1)$ transformados por I_i.

Uma outra forma equivalente para a representação dos vértices das instâncias pode usar o volume do nó raiz V_i em vez de G_i^{-1}. Nesse caso, os vértices do volume de cada instância são dados pelo cubo unitário em $(-0.5, -0.5, -0.5)(0.5, 0.5, 0.5)$ e transformados por $M_i V_i$.

3.6.3 Travessia da BVH de Dois Níveis

O algoritmo de percurso da BVH de dois níveis descrito a seguir faz uso das da BVH das instâncias, o vetor de BVHs das malhas, descritas nas sessões anteriores, e uma estrutura de dados que faz o mapeamento entre cada instância e sua respectiva malha, também fornecendo a matriz de transformação da instância.

A percurso acontece de forma similar a descrita em sessões anteriores. Faz-se necessários dois pares de pilhas de nós e listas de folhas. Uma par para BVH de cima, outro para as BVHs de baixo. Não alcançando os limites das pilhas, o algoritmo percorre a BVH de instâncias por completo, para em seguida percorrer a BVH de cada malha em sequência.

O algoritmo começa pela BVH do nível de cima, a BVH de instâncias. O nó raiz de índice 0 é adicionado à pilha. A percurso gera uma lista de instâncias e cada instância é processada em sequência. Para cada instância da lista, o *offset* do nó raiz da BVH da malha correspondente é consultado, o raio é transformado pela inversa da matriz de transformação da instância e é realizada uma busca na BVH da malha usando o raio transformado. Por fim, no laço mais interno, interseção do raio com a lista de primitivas de cada malha é computada. O

código abaixo demonstra o algoritmo descrito.

```
1  Fragment  castRay ( vec3   ro , vec3  rd )
2  {
3    Fragment  f  =  fNone ;
4
5      nodeStackInstancePush (0) ;
6      do {
7      traverseInstance ( ro , rd ) ;
8        for ( int    i =0;   i < leafStackSizeInstance ;         ++ i )
9        {
10       int   instanceId     =  leafStackInstance [ i ];
11       BaseInstance     baseInstance     =  baseInstances [ instanceId
            ];
12       int   nodeOffset     =  meshNodeOffsets [ baseInstance . meshId
            ]. nodeOffset ;
13
14       mat4  m =  inverse ( baseInstance . modelMatrix );
15       vec3  d  =  normalize (( m * vec4 ( rd ,0) ) . xyz ) ;
16       vec4  o  =  m * vec4 ( ro ,1) ;
17       o  /=  o . w ;
18
19       nodeStackMeshPush ( nodeOffset ) ;
20       do {
21         traverseMesh ( o. xyz , d );
22           for ( int    j =0;   j < leafStackSizeMesh ;       ++ j )
23           {
24             const  int   id   =  leafStackMesh [ j ];
25             const  vec3  p1  =  getPosition ( id ) ;
26             const  vec3  p2  =  getPosition ( id +1) ;
27             const  vec3  p3  =  getPosition ( id +2) ;
28             f  =  opU (f , Fragment ( isTriangle ( o . xyz ,d ,p1 , p2 , p3 )
                 ,id , instanceId ) );
```

```
29                }
30            } while ( nodeStackSizeMesh       >  0) ;
31        }
32    } while ( nodeStackSizeInstance       >  0) ;
33
34    return    f ;
35 }
```

É preciso estar atento às limitações descritas anteriormente na sessão sobre percurso. Restrições no modo de execução e na alocação de memória na GPU força a lidar com a fila de forma explícita e o custo de pré alocação de espaço em memória afeta o fator de residência do algoritmo. Ainda, a necessidade de inserir rotinas para a distinção entre diferentes instâncias sobre a mesma malha, aplicando transformações ao raio, atribuindo índices ao fragmento, etc, nos exige a criação de uma ramificação adicional.

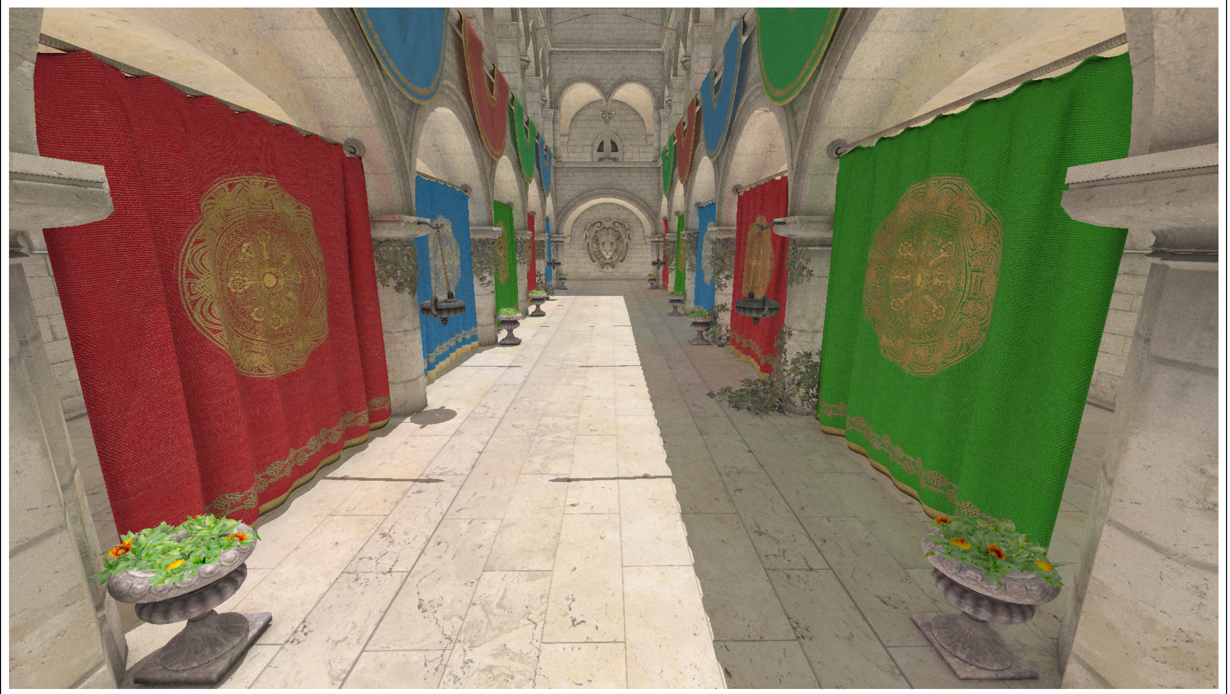
O algoritmo de percurso resultante possui quatro laços aninhados, o efeito de sobrecarga que essas ramificações causam no *hardware* paralelo deve ser estudado.

3.7 Pipeline Hibrida

Essa sessão descreve o modelo de uma *pipeline* de renderização hibrida, similar a outros modelos tais como (BARRÉ-BRISEBOIS *et al.*, 2019; WILLBERGER *et al.*, 2019; SCHANDER, 2017). Esse modelo combina passes de rasterização e de computação generalista paralela (*compute*) em um *ray tracer* estocástico similar ao (WHITTED, 1980) para a geração de imagens com suporte à GI e materiais **Physically Based Rendering (PBR)**, fazendo uso das capacidades de aceleração gráfica e intercomunicação entre passes fornecidas pelo *hardware* gráfico.

A *pipeline* executa o passe de rasterização usando a técnica de *deferred rendering* para a computação dos raios primários. *Ray tracing* é executado a partir das informações contidas no *g-buffer* para a computação de 3 componentes; *ambient occlusion*, sombra e iluminação difusa indireta. A imagem final 5 passa por um passe de pós-processamento de correção de gama.

Figura 5 – Render da cena Sponza.



3.7.1 Construção de cena

Uma cena é formada a partir do instanciamento de malhas, quais, por sua vez, são compostas por primitivas geométricas e um material atrelado à cada malha. A cena também é iluminada por fontes de luz.

Cada vértice, além de uma posição, também possui os atributos de normal, coordenada de textura, tangente e bitangente. Os atributos associados ao vértice existem como parâmetros para especificar a forma de renderização da primitiva. A posição e os demais atributos são armazenados em dois *buffers* diferentes.

Os *buffers* são espaços contíguos em memória que armazenam valores intercalados. O *buffer* de vértices armazena vetores de posições, enquanto o *buffer* de atributos de vértices armazena sequências de atributos intercalados na mesma ordem da sequência de posições no *buffer* de vértices.

As primitivas geométricas são indexadas. Um *buffer* armazena sequências de três índices, formando triângulos em ordem anti-horária, referenciando posições no buffer de vértices. A indexação de primitivas, em contraste ao armazenamento explicito de sequências de vértices, tem o potencial de redução do custo de armazenamento e movimentação de memória.

Cada malha possui um índice dando um *offset* na posição do *buffer* de índices e apontando para a sequência de primitivas pertencente à malha, bem como um valor para o

tamanho dessa sequência. Uma malha também possui um índice apontando para um elemento no *buffer* de materiais.

Um material é uma estrutura de dados que fornece coeficientes para a execução de algoritmos que simulam a interação da luz com um objeto. Em conjunto aos atributos de vértice, eles fornecem os dados necessários para o cálculo de iluminação na superfície de uma malha. Tipos avançados de materiais armazenam índices apontando para texturas que fornecem esses coeficientes e são uma parte fundamental para a renderização foto realista.

Texturas são imagens que codificam coeficientes de iluminação tais como cor, normal, rugosidade e transparência, dando detalhes de alta resolução aos objetos. O mapeamento da textura na superfície de um objeto é dado pela interpolação das coordenadas de textura de uma primitiva geométrica enquanto seu método de amostragem usa diversos algoritmos de filtragem de textura quais usam como heurística a área aproximada da amostra.

3.7.2 Passe de Rasterização

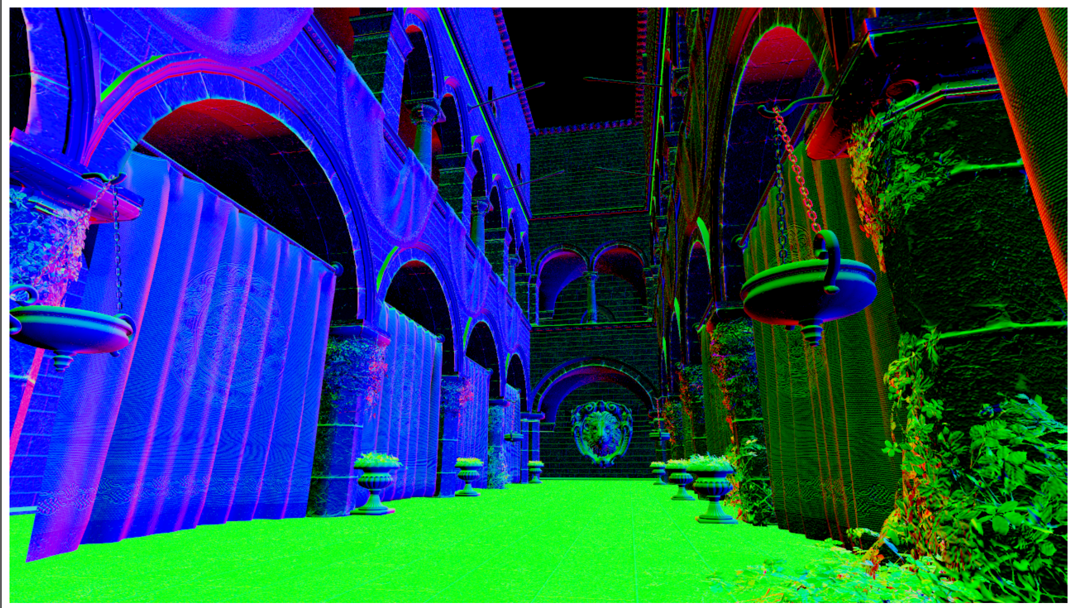
O processo de renderização começa com a execução do passe de rasterização, gerando um resultado equivalente a computação dos raios primários, e geração do *g-buffer* contendo a informação geométrica e de material PBR da cena em resolução de tela.

Esse passe rasteriza malhas indexadas instanciadas em um única chamada. Durante a etapa de processamento de vértices cada malha pode ser processada múltiplas vezes, em cada vez seus vértices são transformados em espaço de mundo pela respectiva matriz de transformação da instância M_i. Atributos de normal, tangente e bitangente são transformados pela matriz de transformação normal $N_i = (M_i^{-1})^t$.

Na etapa de geração de fragmentos os atributos de vértice das primitivas interpoladas e os valores de posição, albedo (cor difusa) e normal são armazenados em três texturas diferentes no *g-buffer*. Enquanto a posição é resultado da interpolação dos vértices de cada primitiva, os demais valores são provenientes das amostras de texturas.

O mapeamento de texturas vem das coordenadas de texturas interpoladas. Os valores de textura são amostrados usando filtragem anisotrópica. O atributo de normal da textura n_{tex} é transformado do espaço tangencial (espaço local da textura) para espaço de mundo usando a base formada pelos atributos de vértice normal n_{vert}, tangente t_{vert} e bitangente b_{vert}, $n_{world} = [b_{vert}, t_{vert}, n_{vert}]n_{tex}$. A figura 6 mostra uma visualização das normais armazenadas no *g-buffer*.

Figura 6 – Normais armazenadas no *g-buffer*.



Embora esse passe proporcione a computação dos raios primários de forma eficiente, ele herda as mesmas limitações da *pipeline* de rasterização, Ele tem problemas na renderização de objetos semitransparentes, por exemplo, não é capaz de simular efeitos de *depth of field*, exposição, e o *g-buffer* não dá suporte para *antialiasing*.

3.7.3 Passe de Path Tracing

O passe de *path tracing* é executado após a rasterização e usa dados do *g-buffer* para a geração de raios, também são realizadas operações de percurso de raios na cena, interseções com geometria e o cálculo da iluminação. O passe faz múltiplas chamadas a grupos de processos paralelos. Cada chamada opera em partições da imagem. Grupos de processos operando em *pixels* vizinhos compartilham dados obtidos pela computação de diferentes amostras para cada *pixel*.

Durante a simulação do transporte de luz, o custo da computação de múltiplas reflexões e as diferentes interações com os materiais na cena podem variar radicalmente entre diferentes areas da cena e de forma imprevisível. Muitas vezes não é possível saber a priori o custo de computação de cada *pixel*. Esquemas de particionamento de imagem contribuem para a eficiência de de distribuição de carga. Esse trabalho adota o modelo comum de particionamento da imagem em *tiles*, onde grupos de processos paralelos compartilham memória de *pixels* adjacentes. Outro modelo comum é o particionamento em *scan lines*. Trabalhos como (ANTWERPEN *et al.*, 2019) propõem esquemas de particionamento ainda mais elaborados mas

que proporcionam melhor eficiência.

O método estocástico de *ray tracing* exige a computação de diversas amostras que depois precisam ser integradas para formar a cor final de cada *pixel*. Esse trabalho simula o transporte de múltiplos raios de luz em paralelo. Cada *pixel* possui um grupo de processos paralelos responsáveis pela computação dessas amostras e armazenamento em memória compartilhada. A computação da cor de cada *pixel* acontece após o término da execução do algoritmo e a sincronização de memória de todos os processos do grupo.

O auto custo da computação de múltiplas amostras para cada *pixel* gera um resultado próximo ao conceitualmente ótimo. Outras abordagens optam por um número reduzido de raios. (SCHANDER, 2017) usa um *g-buffer* redimensionado para metade de sua resolução, gerando um raio para cada quatro *pixels* da imagem. (BARRÉ-BRISEBOIS *et al.*, 2019) também opta pela geração de um quarto de raios por *pixel* para a computação de reflexões. (SCHANDER, 2017; BARRÉ-BRISEBOIS *et al.*, 2019; WILLBERGER *et al.*, 2019) fazem uso de filtragem espaço temporal para corrigir pelo número limitado de amostras.

3.7.4 Geração de Raios

Os raios são gerados a partir das informações contidas no *g-buffer* usando um algoritmo determinístico pseudo aleatório e apresentando distribuição normal. Durante a simulação do transporte de luz, os novos raios gerados também seguem o mesmo padrão de geração.

Com os raios primários gerados por rasterização, cada *pixel* do *g-buffer* possui informações geométricas e de propriedade de material necessárias para a geração de raios secundários. Os dados geométricos para a geração de raios terciários adiante vêm da interpolação de atributos de vértices usando valores paramétricos computados pela interseção do raio com as primitivas geométricas.

Os valores de posição e normal dão origem do raio e o hemisfério da sua direção. O método de (WäCHTER; BINDER, 2019) é usado para a definição do ponto de origem do raio de modo a evitar auto interseção com a geometria de origem.

O desafio para a geração de raios está na seleção de um grupo finito de amostras que contêm o maior número possível de informação sobre um domínio de modo a maximizar o ritmo de convergência. Dois fatores afetam a distribuição de raios, o modo de geração de número aleatórios e o modo de mapeamento desses valores no semi hemisfério.

A natureza determinística dos algoritmos confere aos computadores grande poder e

aplicabilidade, porém essa características é detrimental para algoritmos de geração de números aleatórios. Algoritmo pseudo aleatórios são algoritmos determinísticos quais geram numero perceptivelmente aleatórios. A sequência de Halton (HALTON, 1964) é um exemplo. Nos trabalhos de (WILLBERGER *et al.*, 2019; BARRÉ-BRISEBOIS *et al.*, 2019), esse algoritmo é computado durante a execução do programa. (SCHANDER, 2017) escolhe usar valores pé computados em uma textura de ruído. Esse trabalho faz uso do algoritmo de (VIVO; LOWE, 2015) computado em tempo de execução. Ele é relativamente simples necessitando apenas operações vetoriais e trigonométrica e não apresenta ramificações.

A qualidade das amostras é dependente da uniformidade da distribuição dos números aleatórios gerados por esses algoritmos. O método proposto por (PHARR, 2019a) melhora a qualidade dos valores gerados pelo técnica de estratificação em intervalos regulares e, é capaz de ser executada em tempo constante, ideal para ser usado em tempo de execução.

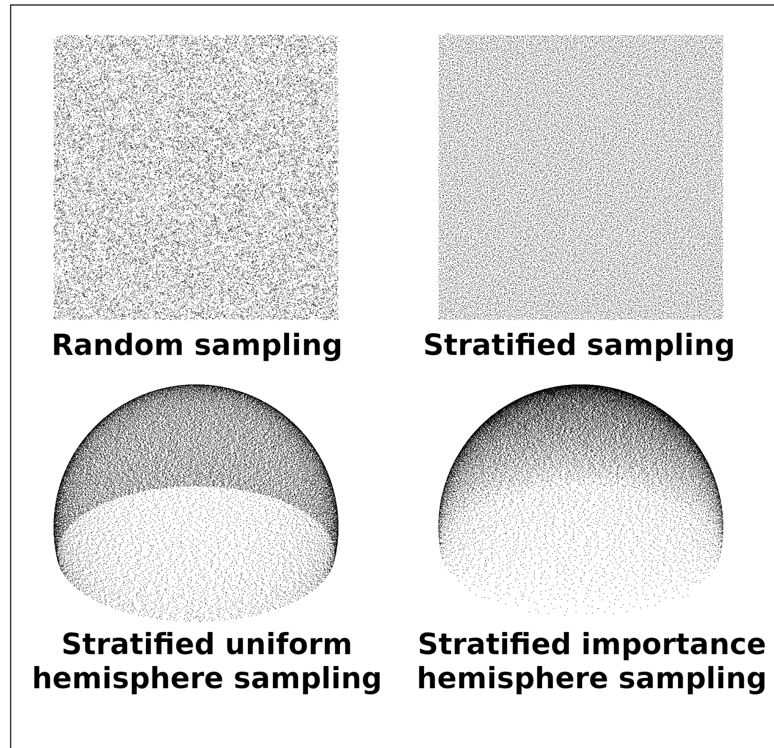
Sendo ξ_1 e ξ_2 dois números aleatórios uniformemente distribuídos no intervalo $[0, 1]$, o próximo passo para a geração do raio envolve o mapeamento desses valores no semi hemisfério. O mapeamento uniforme é dado como $(x, y, z) = (\sqrt{1-\xi_1^2}\cos(2\pi\xi_2), \sqrt{1-\xi_1^2}\sin(2\pi\xi_2), \xi_1)$ e mapeia os valores 2D para a superfície de uma semi esfera na direção do eixo z.

O resultado da integração Monte Carlo sobre todo um semi hemisfério usando essa distribuição uniforme é dado por $\frac{1}{\pi N}\sum_i^N V(\omega_i)\cos\theta_i$. Amostras próximas ao horizonte têm $\cos(\theta_i) \approx 0$ e contribuem muito pouco para o resultado final.

Importance sampling procura remediar esse problema dando prioridade na escolha de amostras que têm maior potencial de contribuição. (PHARR, 2019b) sugere o uso de uma distribuição normal no mapeamento esférico, dada como $(x, y, z) = (\sqrt{\xi_1}\cos(2\pi\xi_2), \sqrt{\xi_1}\sin(2\pi\xi_2), \sqrt{1-\xi_1})$, resultando em 30% menos erro em relação ao mapeamento uniforme. A integração Monte Carlo implementada neste trabalho usando essa nova distribuição é dada como $\frac{1}{N}\sum_i^N V(\omega_i)$. A imagem 7 mostra as diferenças entre os métodos de amostragem. O código abaixo implementa o algoritmo de geração de raio.

```
1 vec2 genSample ( const float t )
2 {
3     float u = rand ( vec2 (13.9898 ,18.2394) ,t );
4     float v = rand ( vec2 (24.7264 ,15.8765) ,t );
5     return vec2 (u , v );
6 }
```

Figura 7 – Esta figura mostra as diferenças entre os métodos de amostragem. *Stratified sampling* proporciona uma melhor distribuição de amostras para algoritmos de geração de números pseudo aleatórios. *Importance sampling* altera as propriedades de distribuição de um conjunto de amostras baseado em modelos probabilísticos a fim de acelerar convergência em métodos estocásticos.




```
7  vec2 stratifiedSample ( const float nSamples , const float i)
8  {
9      float strataSize = sqrt ( nSamples );
10     vec2 strata = ivec2 (i , floor ( i / strataSize )) % ivec2 (
           strataSize ) ;
11     return ( strata + genSample ( i ) ) / strataSize ;
12 }
13 # define PI 3.14159265
14 vec2 uniformDistribution ( const vec2 s )
15 {
16   return vec2 ( acos ( sqrt (1 - s . x ) ) ,s . y * PI ) *2;
17 }
18 vec3 cosineDistribution ( const vec2 s)
19 {
20   return vec3 ( sqrt ( s . x ) * cos (2* PI * s . y ) , sqrt ( s .x ) * sin (2* PI * s .
```

```
            y ) , sqrt (1 - s. x ) ) ;
21  }
22  vec3  transformToFrame ( const    vec3  d , const  vec3  z )
23  {
24    vec3  y  =  normalize (( abs ( z . x ) > abs ( z .y ) ) ? vec3 ( - z .z ,0 , z . x ) :
            vec3 (0 , z .z , -z . y ) ) ;
25    vec3  x  =  cross (y , z ) ;
26    return   ( mat3 (x ,y , z ) * d ) ;
27  }
28  vec3  rayBRDF ( const  vec3  rd , const  vec3  n , const  int  nSamples ,
        const  int  i )
29  {
30      return   transformToFrame ( cosineDistribution (
              stratifiedSample ( nSamples , i )) ,n ) ;
31  }
```

3.7.5 Travessia

O cálculo de interseção dos raios gerados com a geometria da cena involve o uso de uma BVH de dois níveis como estrutura de aceleração. Nas sessões anteriores foram descritos métodos de construção e percurso dessa BVH. Porém, trabalhos como (BARRÉ-BRISEBOIS *et al.*, 2019; WILLBERGER *et al.*, 2019) apresentam outros métodos interessantes de percurso. Nessa sessão são apresentadas de forma comparativa métodos alternativos de percurso de cena bem como um algoritmos de interseção de raio com triângulo.

Em alguns casos, a BVH não precisa representar a geometria da cena como completo. A BVH usada no trabalho de (WILLBERGER *et al.*, 2019) representa apenas um subconjunto da cena. Sua BVH precisa ser reconstruída a cada quadro e contêm apenas um subconjunto centralizado ao redor da câmera. Conforme a camera se move, novos elementos são adicionados ou removidos da BVH, baseado não só na posição espacial da geometria, mas também por fatores como a área de superfície do objeto e outras constantes definidas pelo usuário que atribuem a importância do objeto para a cena. A BVH resultante possui tamanho reduzido, o que também reduz os custos de suo percurso, porém a constante necessidade de reconstrução adiciona um

custo significativo. Devido à falta de uma informação completa sobre a cena, essa BVH também afeta gera erros na computação de iluminação em especial de materiais semitransparentes.

Uma BVH também não precisa representar a geometria da cena. (BARRÉ-BRISEBOIS *et al.*, 2019) constrói uma estrutura de aceleração usada para representar *surfels*, que são "pequenas fontes luminosas", posicionadas em espaço de mundo, cobrindo superfícies geométricas da cena visíveis pela câmera e usadas para a computação aproximada de GI. A BVH resultante tem sua complexidade independente do tamanho da cena. Porém, a falta de detalhes finos inerente faz com que essa técnica seja viável apenas para a computação de iluminação difusa indireta.

Outras técnicas de percurso de raio de uma cena sequer envolvem BVHs. (BARRÉ-BRISEBOIS *et al.*, 2019; WILLBERGER *et al.*, 2019) implementam técnicas de rasterização para a aproximação de cálculos de iluminação, como *shadow mapping*, *screen space ambient occlusion (SSAO)* e *screen space reflections (SSR)* para a computação de sombras, oclusão e reflexões, respectivamente. Todas elas são limitadas pela resolução do *raster*, e são dependentes da *viewport*. Na prática, essas técnicas são usadas em conjunto e tem papel supplemental ao *ray tracing* geométrico.

3.7.6 Interpolação e Filtragem de Textura

O algoritmo de (MöLLER; TRUMBORE, 1997) de interseção de um raio com um triângulo é amplamente usado em computação gráfica, enquanto o algoritmo de (BALDWIN; WEBER, 2016) oferece alto potencial de vetorização e oportunidades para pré computação. Tais algoritmos fornecem não só o ponto de interseção e a normal do triângulo sem custos adicionais, quais são usadas na geração de raios, como também a parametrização do ponto de contato em termo de coordenadas baricêntricas, valores que possibilitam a interpolação de atributos de vértices e filtragem de textura.

A interpolação dos atributos de vértices é dada pela combinação linear de cada atributo de cada vértice do triângulo pelas coordenadas baricêntricas do ponto de interseção. Sejam $\lambda_1, \lambda_2, \lambda_3$ as coordenadas baricêntricas e uv_1, uv_2, uv_3 os atributos de coordenadas de textura de cada vértice, por exemplo, a coordenada de textura interpolada$uv = uv_1\lambda_1 + uv_2\lambda_2 + uv_3\lambda_3$. A mesma operação pode ser realizada para os demais atributos de vértice, e até para as próprias posições do triângulo.

A consulta pelo *pixel* mapeado seguindo a posição definida pela coordenada de textura interpolada gera artefatos. Isso porque a superfície da geometria não necessariamente

corresponde a resolução e orientação da textura mapeada a ela. Uma unidade da superfície da geometria projetada em tela que forma um *pixel* na tela, por exemplo, pode representar menos de um, ou um conjunto de *pixels* da textura mapeada à geometria.

Diversos algoritmos de filtragem de textura remediam tais artefatos de forma diferente, *nearest neighbor interpolation* faz uma amostragem no *pixel* mais próximo à coordenada. Algoritmos mais avançados como o linear interpola conjuntos de *pixels* da textura de forma regular. Finalmente, o algoritmo de filtragem anisotrópica realiza amostras de *pixels* de forma não linear e possui os melhores resultados dente os algoritmos de filtragem existentes.

O algoritmo de filtragem linear (e suas variantes, bilinear e trilinear), recebem com entrada, além de coordenadas de textura, um valor de resolução correspondente ao tamanho da área de amostragem, também chamado de **level of detail (LOD)**. O algoritmo anisotrópico recebe dois valores, também chamados de gradientes, correspondentes a area não regular de amostragem.

Ao computar a interseção de um raio com um triângulo,   após a interpolação de atributos de vértices, o objetivo de um *ray tracer* durante a simulação do transporte de luz é de extrair uma amostragem das propriedades do material PBR no ponto de interseção, quais estão codificadas em texturas. Extensões ao algoritmo de interseção permitem a computação do LOD e gradientes, dando os valores finais necessários para a execução de algoritmos de filtragem de textura.

O método de *cone tracing* (AMANATIDES, 1984) e *ray differentials* (IGEHY, 1999) proporcionam a computação de LOD e dos gradientes, respectivamente. Elas são duas estratégias usadas em *ray tracing* e abordadas em (AKENINE-MöLLER *et al.*, 2019) para a computação de filtragem de textura.

Usando *ray differentials*, torna-se possível implementar filtragem anisotrópica. Essa técnica computa o gradiente pelas derivadas parciais da coordenada de textura, que por sua vez são computadas encontrando as coordenadas baricêntricas da interseção de dois raios adicionais com o triângulo, "raios diferenciais"criados a partir de pequenas perturbações em espaço de tela do raio original.

Embora trivial para a computação usando raios primários, quais são gerados a partir da tela, o algoritmo torna-se muito complexo a medida que o raio sofre reflexões. Esse trabalho faz uso de *ray differentials* para a filtragem de textura dos raios primários e é realizado no raster. Na pipeline raster é usado cone tracing, que gera filtragem de qualidade inferior, porém pela

natureza da baixa frequência da iluminação indireta, a diferença torna-se imperceptível.

O algoritmo de *cone tracing* é usado no passe de *path tracing* e incrementa o algoritmo de interseção de raio com triângulo para a computação do LOD da textura e isso possibilita a filtragem trilinear da textura associada ao material PBR da malha. Dado o vetor normal $\hat{n}$ do triângulo, o vetor de direção $\hat{d}$ do raio e o escalar t da equação paramétrica do raio que indica a distância da origem do raio ao ponto de interseção do raio com o triângulo, o valor de LOD $l = \log_2(\frac{-\alpha t}{n \cdot d})$ para um α suficientemente pequeno, onde α é uma constante atribuída como ângulo de propagação. Esse trabalho usa $\alpha = 5e-3$. A imagem 8 mostra uma comparação entre as diferentes técnicas de filtragem.

3.7.7 Cálculo de Iluminação

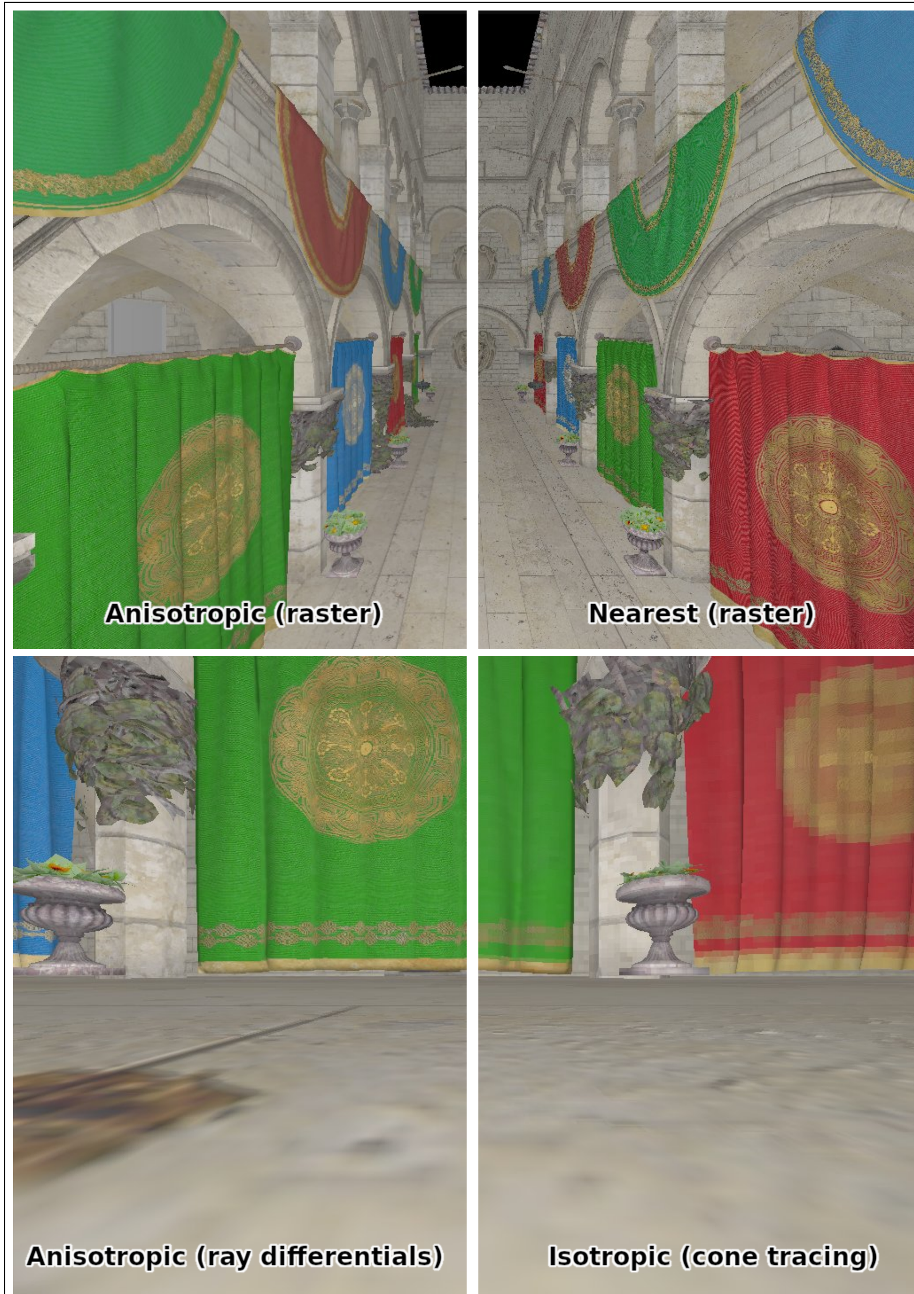
A cálculo de iluminação estima o valor de radiância chegando a cada *pixel*. Ele é o resultado de um processo estocástico de integração do semi hemisfério ao redor do *pixel* do *g-buffer*. O cálculo combina os fatores de **ambient occlusion (AO)**, iluminação direta com sombras, iluminação indireta difusa e a iluminação ambiente.

GI pode ser aproximada pela equação de renderização (KAJIYA, 1986), ela evidencia a natureza integral e recursiva do transporte de luz. Esse trabalho usa uma simplificação dessa equação, materiais emissivos são desconsiderados, são computadas apenas fontes de luz pontuais e ambiente e o modelo Lambertiano de reflexão difusa em superfícies opacas é usado. A cor de um *pixel* é dada pela equação $L_o(x, \omega_o) = \int_\Omega f_r(x, \omega_i) L_i(x, \omega_i) d\omega_i$, a integral sobre o hemisfério Ω no ponto x da luz L_i chegando a x pela amostra na direção ω_i com a distribuição de refletância f_r. A expansão dessa equação, vista abaixo, gera uma recursão sem fim.

$$L_o(x_o, \omega_o) = \int_{\Omega_o} f_r(x_o, \omega_i) \int_{\Omega_i} f_r(x_i, \omega_j) \int_{\Omega_j} f_r(x_j, \omega_k) \ldots \; d\omega_j \; d\omega_i$$

A computação da equação dessa maneira é inviável. O que é feito, em vez disso, é a computação da integral do primeiro nível L_o, de forma estocástica, e a computação do caminho percorrido por cada raio ω_i, acumulando uma única amostra durante um número finito de reflexões, dando o nome *path tracing*. O calculo de cada caminho acumula valores de iluminação difusa modulada pelo *tracing* de sombras e iluminação ambiente modulada por *tracing* de AO. O algoritmo abaixo resume o cálculo do transporte de luz.

Figura 8 – Tipos de filtragem de textura e diferentes pipelines de renderização. Acima, filtragem de textura na pipeline de rasterização. Abaixo, filtragem de textura usando a técnica de *ray differentials* e *cone tracing*.



$$c_{pixel} = \frac{1}{N_{samples}} \sum_{i}^{N_{samples}} L(x_o, \omega_i)_{N_{depth}}$$

$$L(x, \omega)_k = m_{mat}(l_{amb} r_{occlusion} + l_{dif} r_{shadow} + L(x', \omega')_{k-1}) \text{ até k = 1}$$

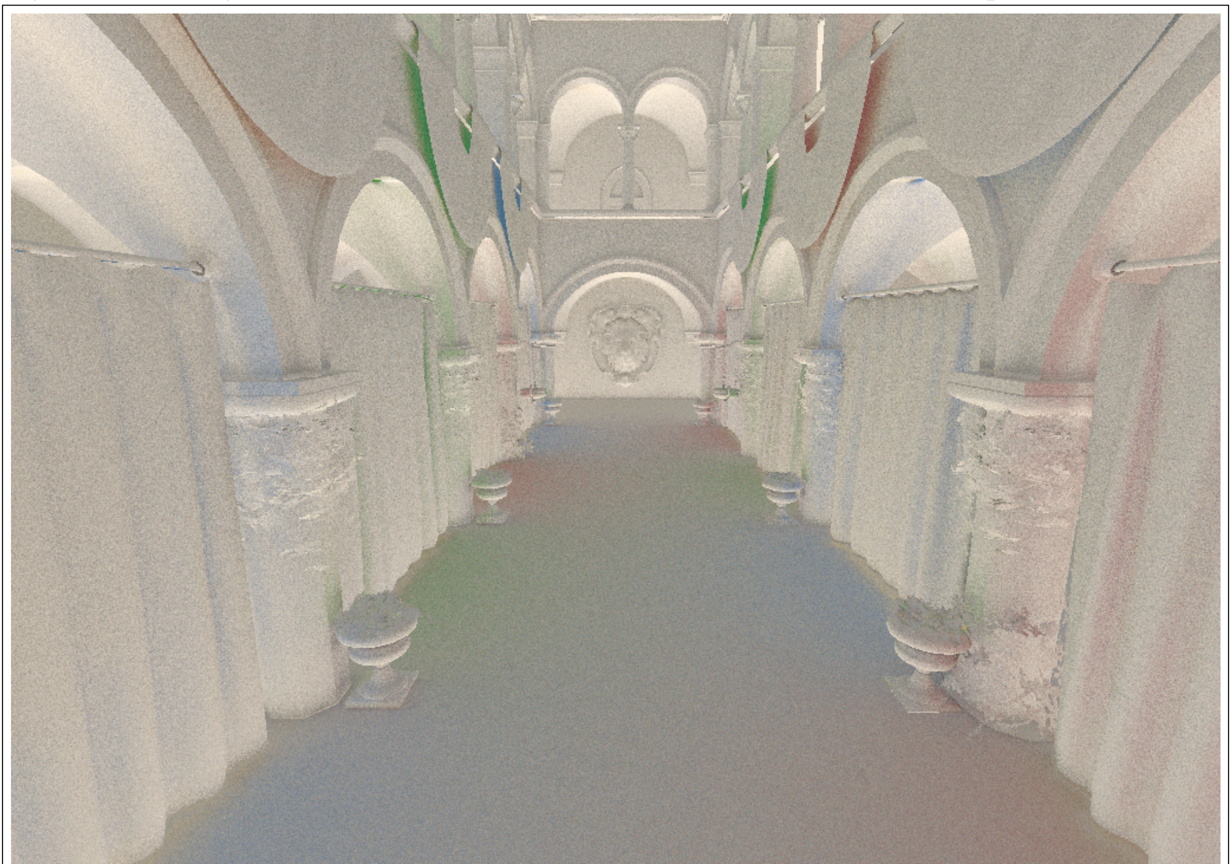
Onde:

- $N_{samples}$ e N_{depth} são constantes que delimitam o número de amostras e número de reflexões por caminho;

- x_o, ω_i são a origem e direção do raio da amostra gerada e x', ω' são suas sucessivas reflexões;
- c_{pixel} é a cor final do *pixel*;
- m_{mat} é a cor albedo do material da primitiva;
- l_{dif} e l_{amb} são as contribuições de iluminação difusa e ambiente;
- r_{shadow} e $r_{occlusion}$ são fatores de visibilidade e oclusão.

O número de amostras e reflexões são constantes definidas antes da execução do algoritmo. As origens dos primeiros raios antes da primeira reflexão vêm do *g-buffer*, enquanto a origem dos próximos raios vêm da interseção com a geometria da cena. A cor de albedo e a normal são amostradas do material PBR. A iluminação difusa e ambiente são realizadas de forma analítica enquanto os fatores de visibilidade e oclusão são *raytraced*. A imagem 9 mostra a contribuição da iluminação indireta na cena. O código abaixo implementa o algoritmo descrito.

Figura 9 – Esta figura mostra a contribuição da iluminação indireta na cena Sponza.



```
1 vec4 renderPath ( vec3 mask , vec3 ro , vec3 rd )
2 {
3     vec4 color = vec4 (0 ,0 ,0 ,1) ;
```

```
4       float   ao  =  1;
5       for ( int    i =0;   i < NUM_LEVELS ;  ++ i)
6       {
7           Fragment   f   =  castRayClosestHit ( ro , rd ) ;
8           if (f . t     <  0)
9           {
10              color . rgb    +=  mask * computeAmbientLight ( rd ) ;
11              break ;
12          }
13
14      Vertex    v  =  interpolate (f ) ;
15      Material     mat  =  evaluateMaterial (f , v) ;
16
17          ro        =  ro   +  f . t * rd   +  f . n *5 e -5;
18          rd        =  rayBRDF ( v . n );
19          mask  *=  mat . dif . rgb ;
20
21          color . rgb    +=
22              mask *( computeDirectLight (ro , mat .n )
23                  + computeAmbientLight ( mat . n ) * ao );
24          ao  =  computeOcclusion ( f. t ) ;
25          if (i     ==  0)   color . a    =  ao ;
26      }
27      return    color ;
28  }
29  vec3   renderScene ( vec2     coords )
30  {
31      vec3   c  =  imageLoad ( imgAlbedo , coords ) . rgb ;
32      vec3   p  =  imageLoad ( imgPosition , coords ) . xyz ;
33      vec3   n  =  imageLoad ( imgNormal , coords ) . xyz ;
34
35          vec4   samples   =  vec4 (0) ;
```

```
36      vec3  color      =  vec3 (0) ;
37    vec3  mask       =  c ;
38    vec3  ro         =  p  +  n *5 e -5;
39    for ( int   i =0;   i < NUM_SAMPLES ; ++ i )
40    {
41      vec3  rd  =  rayBRDF ( n ) ;
42      color   +=  renderPath ( mask , ro , rd ) ;
43    }
44    samples  /=  NUM_SAMPLES ;
45
46      vec3  color0   =
47        mask *( computeDirectLight (ro , n )
48              + computeAmbientLight ( n ) * samples . a ) ;
49    color   =  color0 + samples . rgb ;
50
51    return   color ;
52  }
```

A iluminação ambiente compensa a falta de amostras. Ela pode ser apresentada como apenas um valor constante adicionado à radiância do ponto. Em cenas ao ar livre, a iluminação ambiente normalmente vem da luz espalhada na atmosfera e possui cor azul.

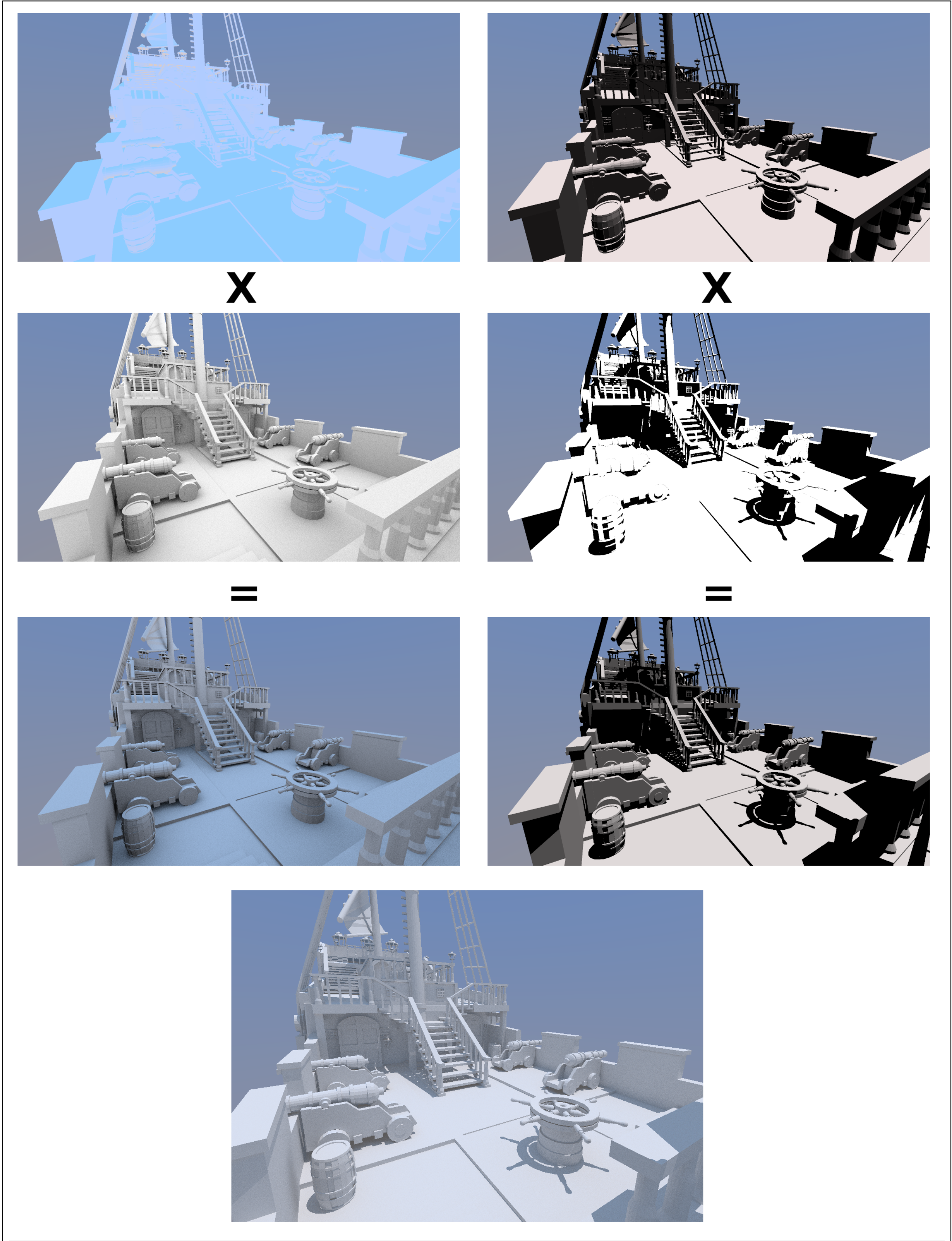
A iluminação difusa é uma simplificação para o fenômeno de espalhamento de luz em superfícies difusas. A reflexão difusa de Lambert é modulada pelo produto interno $clamp(\hat{n} \cdot \hat{l}, 0, 1)$ onde $\hat{n}$ é a normal da superfície e $\hat{l}$ é o vetor de direção do ponto na superfície à fonte de luz.

AO é uma aproximação de GI, ela modula a exposição de uma amostra ao termo de iluminação ambiente e é computada por $e^{(}-t)$ onde $t_{closest}$ é a distância do ponto à primitiva mais próxima interceptada pelo raio da amostra. A implementação de AO necessita uma rotina de percurso na BVH que compute a interseção mais próxima.

A sombra modula a visibilidade de uma amostra ao termo de iluminação difusa. Ela é computada por uma única amostra considerando a fonte de luz como pontual. A visibilidade de uma fonte de luz é computada pela interseção de um raio à qualquer objeto da cena gerando um

valor binário de 0 se houver interseção ou 1 se não houver interseção. A rotina de percurso da BVH é relativamente mais simples, precisando apenas computar a primeira interseção. A figura 10 mostra a combinação dos fatores de iluminação no cálculo de GI da imagem.

Figura 10 – Esta figura mostra a combinação dos fatores de iluminação ambiente, à esquerda, e difusa, à direita, modulados por AO e sombras, quais contribuem para o calculo de GI da imagem abaixo.



4 RESULTADOS

Os algoritmos propostos foram implementados em CPU e em GPU, em uma *pipeline* de renderização hibrida, fazendo uso da computação generalista paralela (*compute*) fornecida pelo hardware gráfico moderno. Testes foram realizadas em uma máquina possuindo uma CPU *Intel i7-8700K* e uma GPU *Nvidia TITAN Xp* e usando *OpenGL 4.6*.

4.1 Base de Dados de Avaliação

Foram coletados dados de diversas cenas diferentes. A tabela 3 mostra a lista de cenas usadas neste trabalho. Para cada cena, durante a execução do algoritmo de construção da BVH, foram registrados o tempo de construção da BVH em CPU e em GPU, dividido em 4 passes; **Computação dos códigos** (*Morton codes*), **ordenação dos códigos**, **computação (da hierarquia) de nós** da árvore e **computação de volumes**.

O passe de computação de volumes possui 3 variantes. Ele foi submetido a testes com 3 algoritmos diferentes, apresentados anteriormente na sessão sobre união de OBBs. São estes; O algoritmo de união de OBBs usando PCA sobre o conjunto de 16 vértices das duas OBB filhas **PCA bounded**, O algoritmo que usa a heurística de interpolação de rotação por *quaternions* **Quaternion** e o algoritmo de união de OBBs usando PCA sobre um conjunto não delimitado de vértices da subárvore **PCA unbounded**.

A tabela 5 mostra a lista completa dos tempos de execução de cada passe na CPU e na GPU, incluindo as 3 variantes do passe de computação de volumes.

Também foram coletados dados que inferem sobre a qualidade das BVs. Esse trabalho fornece diversos algoritmos para a computação de OBBs para BVHs. A tabela 4 fornece uma lista de valores relativos ao volume das OBBs computadas por cada algoritmo, ao lado do valor de volumes computados por um algoritmo de controle que implementa AABBs, para efeitos comparativos.

Especificamente, a tabela 4 mostra o valor da soma de todos os volumes das OBBs da BVH de cada cena usando as três variantes do passe de computação de volumes e um algoritmo de controle que implementa AABBs. A tabela também mostra a frequência com que a OBB gerada para um nó por um dado algoritmo é de melhor qualidade, menor volume, que uma AABB gerada para o mesmo nó.

Para algumas cenas que possuem várias malhas, são construídas dois tipos de BVHs.

Uma **BVH única**, tratando a cena como uma única malha, e uma **BVH de dois niveis**, fazendo uso da heurística de coerência espacial entre as malhas da cena para fins de reduzir o volume total da BVH e assim melhorar a qualidade.

4.2 Qualidade da BVH

A tabela 4 mostra a vantagem do uso da BVH de dois níveis em comparação a construção de uma única BVH para a cena. O uso de BVHs de dois níveis têm o potencial de redução do volume total de das BVs da hierarquia de cena por mais da metade, 52%. O gráfico 12 oferece suporte para a visualização do impacto da BVH de dois níveis em relação à BVH única no custo de percurso.

A distinção entre malhas e suas instâncias, feita pela BVH de dois níveis, não é apenas compatível com modelos de programação e com o fluxo de trabalho artístico, também fica evidente como o particionamento lógico da cena baseado na localidade de instância é capaz de gerar hierarquias qualidade superior e proporcionar maior eficiência de percurso.

Todas as três variantes do algoritmo de computação de volumes geram OBBs de qualidade superior às AABBs de controle. A tabela 4 mostra uma forte correlação entre as variantes e o gráfico 13 mostra o impacto do uso de OBBs em em relação à AABBs no custo de percurso da BVH. Em média, OBBs geradas pela variante PCA unbounded apresentam até 77% do volume em relação a AABBs. A segunda melhor variante em termos de qualidade de geração de OBBs é a PCA bounded, com até 81% do volume, e em terceiro a variante Quaternion, com até 91% do volume.

Essa característica se repete quando é estuada a porcentagem de frequência na qual a OBB gerada para um nó possui qualidade superior à AABB de controle. Embora exista grande variância entre as cenas testadas, a PCA unbounded apresenta melhor porcentagem de geração de OBBs de qualidade superior no Bunny, com 97%, e pior porcentagens Pirate ship, com 59%, o balanço de qualidade continua favorecendo a variante PCA unbounded, seguido de PCA bounded, com 76% no Bunny e Quaternion, com 71%. Sobre diferentes graus de qualidade, os algoritmos de construção de OBBs em cascata representam um balanço entre a qualidade de geração de volumes e eficiência de construção.

4.3 Eficiência Dos Algoritmos de Construção de BVHs

O gráfico 11 ilustra o tempo de construção da BVH de cada cena, focando na proporção desse tempo dedicado a cada passe.

O passe de construção dos volumes apresenta o maior impacto sobre o tempo de construção da BVH. Ele representa 88-97% do tempo gasto na construção da BVH na CPU e 75-82% do tempo de construção da BVH na GPU. O Segundo maior tempo é dedicado à ordenação dos códigos. O tempo de construção cresce proporcional a quantidade de primitivas. Métodos de construção de BVH que fazem uso do paralelismo da GPU oferecem os menores tempos de construção.

A tabela 5 apresenta o tempo execução de cada passe, focando no valor do tempo de cada passe executado em CPU em comparação ao tempo do respectivo passe na GPU. Ela também apresenta o tempo de execução das 3 variantes do passe de computação de volumes em CPU e em GPU.

Os passes executados em GPU possuem cronometragem consistentemente menor em comparação aos respectivos passes executados em CPU. A execução do passe de computação de volumes na GPU apresenta o maior potencial na redução do tempo de construção da BVH. Ele gasta, em media, 15 vezes menos tempo que o mesmo passe executado na CPU. O segundo maior potencial de aceleração vem do passe de computação de nós da árvore na GPU. Ele é em média 12 vezes mais rápido. Por último o passe de computação de códigos, com aceleração de 1.25 vezes.

A variante PCA unbounded é muitas vezes o algoritmo mais caro dentre as opções de algoritmos de computação de volumes na CPU. Ela é em média 63% mais cara que a variante PCA bounded e em média 22% mais cara que a variante Quaternion, seguindo os exemplos usados. Por fazer uso de memória não delimitada, entretanto, é esperado que o custo de execução da variante PCA unbounded aumente de forma exponencial, enquanto as demais variantes se mantenham lineares.

É constatado, como esperado, que a computação de volumes por PCA com uso não delimitado de memória na GPU é inviável devido às limitações do hardware paralelo. Dado uma cena suficientemente grande, a variante PCA unbounded é incapaz de rodar na GPU e trava a máquina.

A variante Quaternions é, em média, 36% mais cara que a variante PCA bounded Em contraste, ela é por pouco a variante mais rápida para computação em GPU, ganhando em média

Tabela 3 – Lista de cenas usadas neste trabalho ordenadas por número de primitivas.

Cena	Numero de malhas	Número de vértices	Numero de primitivas
Breakfast room	240	961 508	1 347 596
Buddha	1	549 409	1 087 474
Dragon	1	438 976	871 306
Sponza	393	209 610	262 267
Pirate ship	29	364 973	155 689
Bunny	1	147 040	144 046
Sibenik	1 087	155 179	75 284

4% de eficiência em relação ao algoritmo de PCA de memória delimitada devido a característica vetorial da plataforma.

4.4 Robustez e Desempenho do *Path Tracer*

Cenas com uma maior razão entre o número de malhas em relação ao número de primitivas tendem a apresentar melhor desempenho em relação a cenas com malhas grandes. De acordo com experimentação, cenas com um maior número de malhas tendem a se beneficiar mais do o maior grau de paralelismo atribuído a características de localidade de dados e menor fator de granularidade.

Em contrapartida, algoritmos paralelos processando cenas com um grande número de primitivas tendem a ficar limitados por barramento de memória por conta do grande número de leituras de atributos de vértices.

Algoritmos geométricos são sujeitos a instabilidade numérica quando executados em máquinas de precisão finita. Mesmo em baixa probabilidade, erros tendem a se acumular em situações em cenários que demandam um grande número de computações. Figuras 14, 15, 16, 17, 18 e 19 mostram a robustez do *path tracer* estocástico fazendo uso da BVH de dois níveis proposta em cenas complexas exibindo múltiplas amostras e varias reflexões de raios.

Tabela 4 – Essa série de tabelas mostra a razão (coluna / linha) entre os volumes totais gerados pelas AABBs e OBBs das BVHs construídas para cada cena (**1a,2a,3a**). Ela reflete as 3 variantes de construção de OBBs apresentadas neste trabalho. As tabelas também mostram em porcentagem a frequência com que uma OBB de cada variante gerada para um nó possui melhor qualidade, menor volume, que AABBs geradas para o mesmo nó (**1b,2b,3b**).

Bunny BVH única

1a	**AABB**	**OBB PCA ubnd**	**OBB PCA bnd**	**OBB Qtrn**
AABB	1.00000	1.09792	1.00944	1.00615
OBB PCA ubnd	0.91081	1.00000	0.91941	0.91641
OBB PCA bnd	0.99065	1.08765	1.00000	0.99674
OBB qtrn	0.99389	1.09121	1.00327	1.00000

1b	**AABB**	**OBB PCA ubnd**	**OBB PCA bnd**	**OBB Qtrn**
soma volumes	19.5222	17.7811	19.3396	19.4029
OBB < AABB %	-	97.2488%	76.0658%	71.6519%

Sponza BVH única

2a	**AABB**	**OBB PCA ubnd**	**OBB PCA bnd**	**OBB Qtrn**
AABB	1.00000	1.02018	1.01363	1.00388
OBB PCA ubnd	0.98022	1.00000	0.99358	0.98402
OBB PCA bnd	0.98655	1.00646	1.00000	0.99038
OBB qtrn	0.99614	1.01624	1.00972	1.00000

2b	**AABB**	**OBB PCA ubnd**	**OBB PCA bnd**	**OBB Qtrn**
soma volumes	5.33077	5.22532	5.25907	5.31018
OBB < AABB %	-	77.9693%	54.0539%	52.8521%

Sponza BVH de dois níveis

3a	**AABB**	**OBB PCA ubnd**	**OBB PCA bnd**	**OBB Qtrn**
AABB	1.00000	1.24495	1.04650	1.04148
OBB PCA ubnd	0.80325	1.00000	0.84060	0.83657
OBB PCA bnd	0.95556	1.18963	1.00000	0.99520
OBB qtrn	0.96017	1.19536	1.00482	1.00000

3b	**AABB**	**OBB PCA ubnd**	**OBB PCA bnd**	**OBB Qtrn**
soma volumes	2.50726	2.50691	2.50693	2.50693
OBB < AABB % top	-	-	10.7143%	13.7755%
OBB < AABB % bot	-	94.2567%	60.4837%	60.3462%

Tabela 5 – Tempos de execução de cada passe na construção da BVH de cada cena na CPU e na GPU, incluindo as 3 variações do passe de computação de volumes; Computação de códigos$_1$, ordenação de códigos$_2$, computação de nós$_3$, e computação de volumes$_4$ em suas 3 variantes; PCA bounded$_{4a}$, Quaternion$_{4b}$ e PCA unbounded$_{4c}$

Cena	**Cmp cod**$_1$	**Ord cod**$_2$	**Cmp nós**$_3$	**PCA bnd**$_{4a}$	**Qtrn**$_{4b}$	**PCA ubnd**$_{4c}$
Buddha						
CPU	17.94ms	164.10ms	35.90ms	405.53ms	652.74ms	805.71ms
GPU	15.97ms	-	2.66ms	63.66ms	63.35ms	>1min
Dragon						
CPU	14.40ms	100.80ms	28.42ms	316.26ms	520.10ms	626.26ms
GPU	13.14ms	-	2.08ms	50.90ms	50.04ms	>1min
Sponza						
CPU	4.27ms	23.62ms	8.52ms	756.01ms	490.58ms	723.99ms
GPU	3.81ms	-	0.64ms	14.14ms	13.40ms	>1min
Pirate Ship						
CPU	2.53ms	12.81ms	5.27ms	68.21ms	97.43ms	119.64ms
GPU	1.73ms	-	0.41ms	9.37ms	9.01ms	>1min
Bunny						
CPU	2.37ms	12.51ms	4.36ms	51.90ms	84.76ms	90.27ms
GPU	1.90ms	-	0.36ms	7.67ms	7.23ms	>1min
Sibenik						
CPU	1.26ms	6.44ms	2.50ms	42.96ms	53.53ms	60.60ms
GPU	0.85ms	-	0.21ms	5.04ms	4.70ms	>1min

Figura 11 – Análise comparativa do tempo de construção da BVH em CPU e em GPU entre cenas de tempos de construção similares.

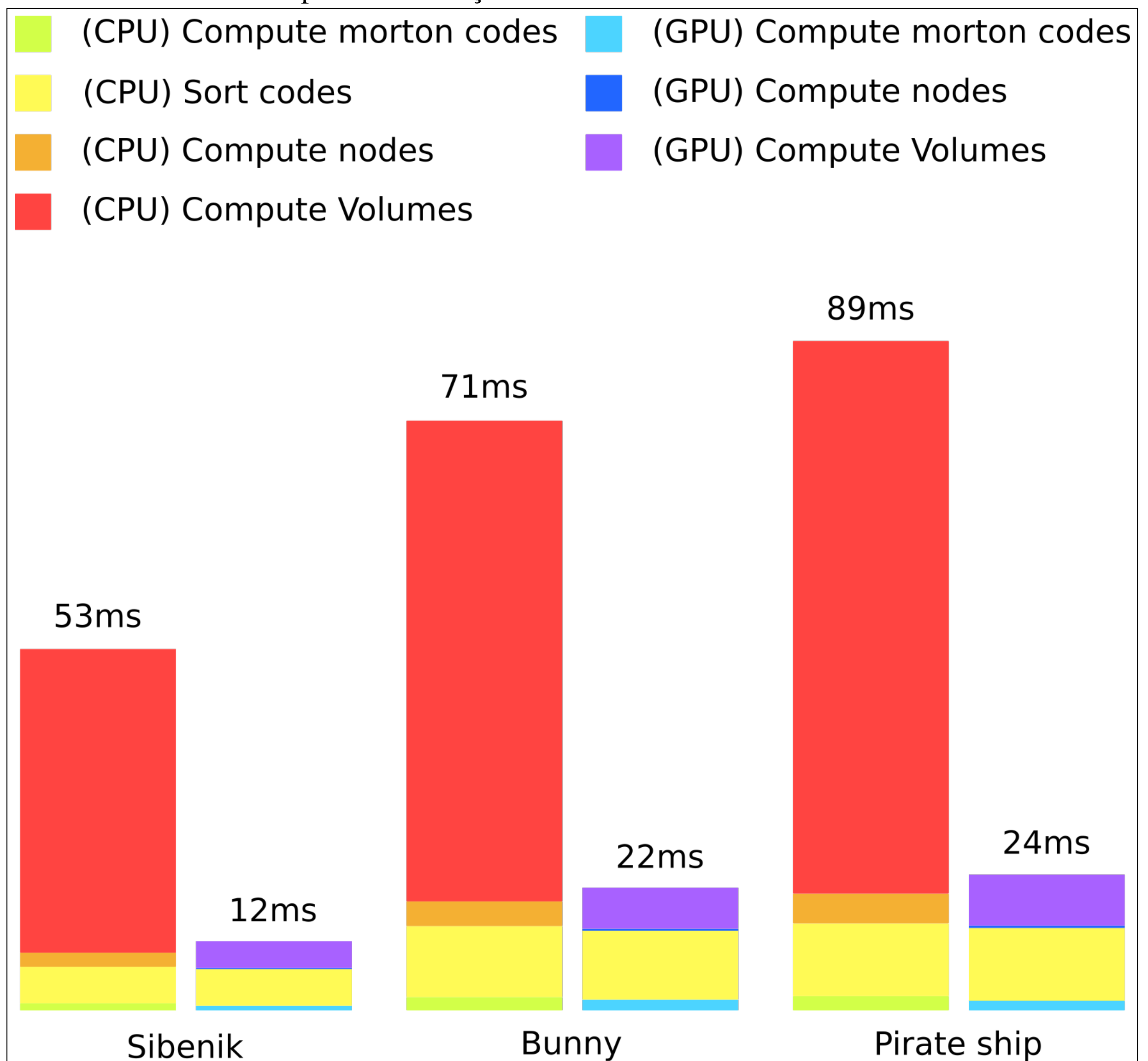


Figura 12 – *Heatmap* mostrando o custo de percurso da BVH de diferentes malhas usando AABBs à esquerda e OBBs à direita. O gradiente representa o número de interseções com BVs realizadas pelo raio durante o percurso.

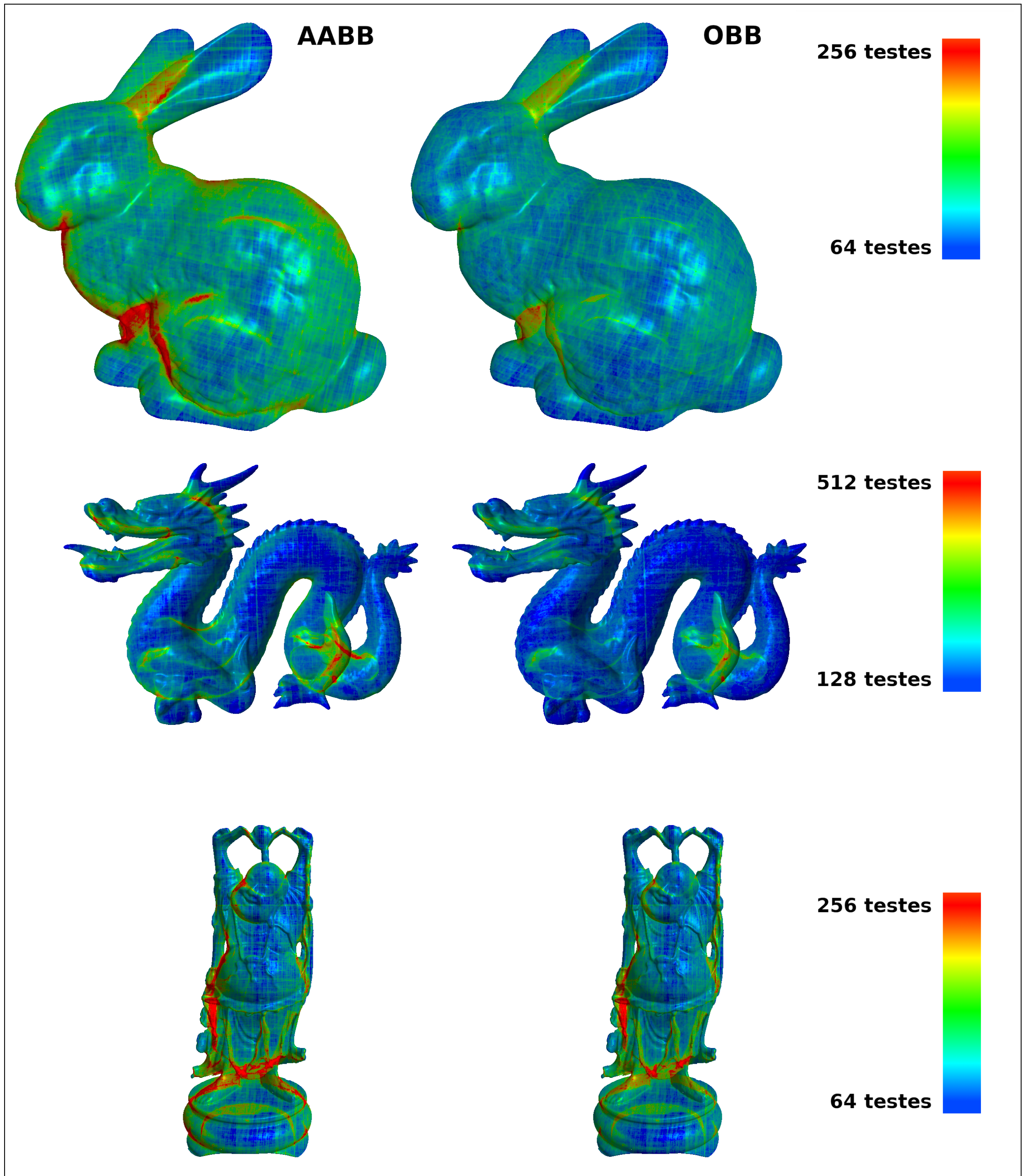


Figura 13 – *Heatmap* mostrando o custo de percurso em uma cena contendo múltiplas malhas. Acima, a cena construída usando apenas uma BVH. Abaixo, a cena construída usando uma BVH de dois níveis. O gradiente representa o número de interseções com BVs realizadas pelo raio durante o percurso.

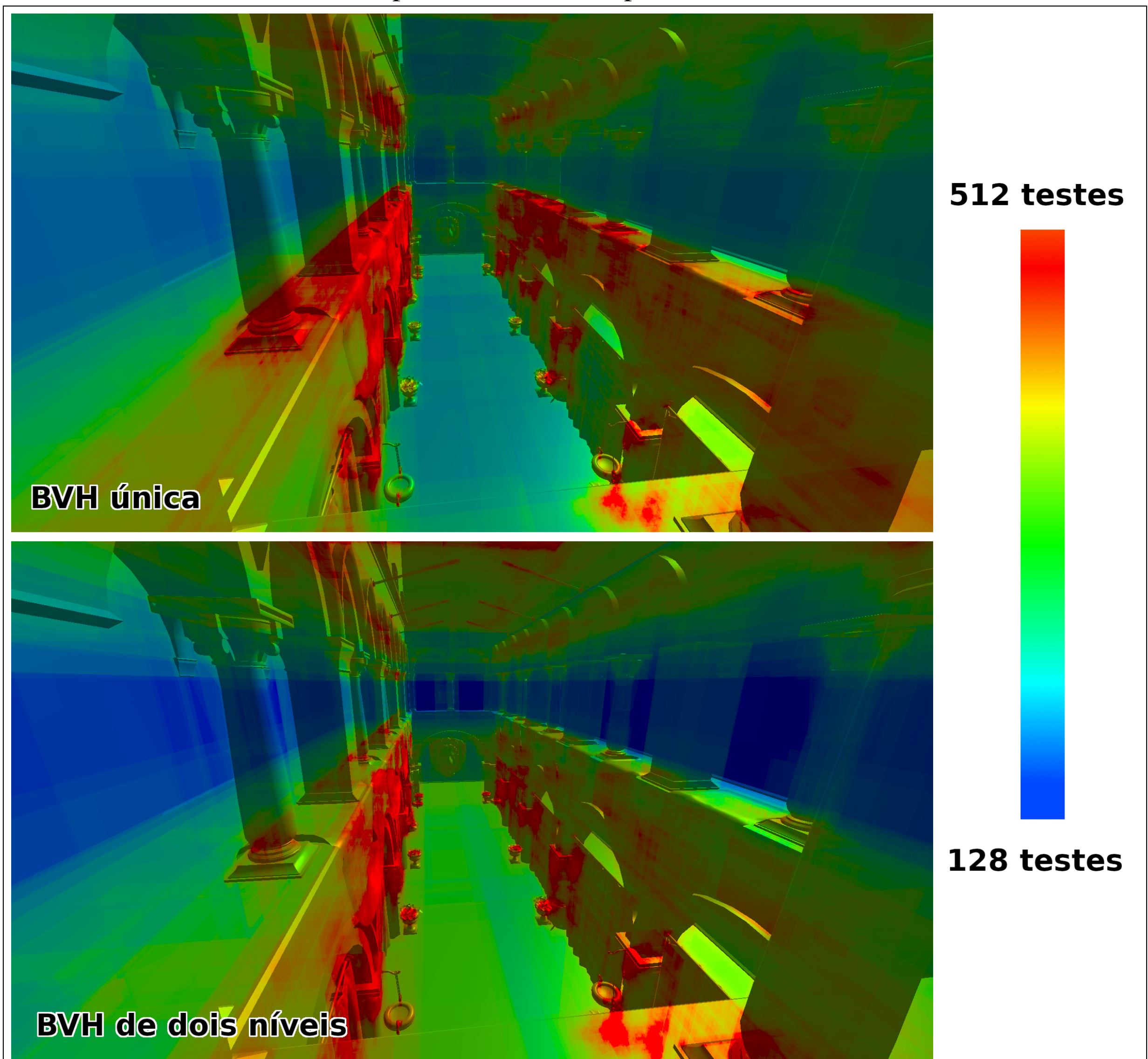


Figura 14 – Cena *Breakfast room* renderizada em 5m0s à 1920x1080, 24 amostras por *pixel* e 2 reflexões

Figura 15 – Cena *Buddha* renderizada em 14s à 1920x1080, 6 amostras por *pixel* e 2 reflexões

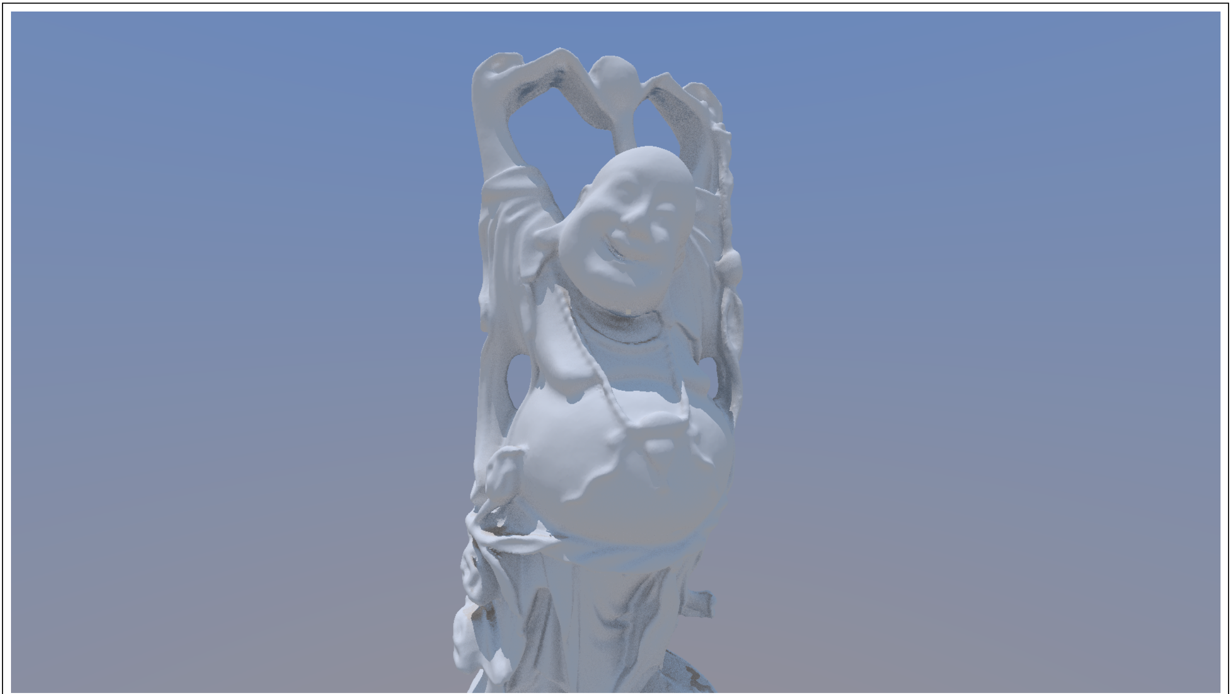


Figura 16 – Cena *Dragon* renderizada em 24s à 1920x1080, 16 amostras por *pixel* e 2 reflexões

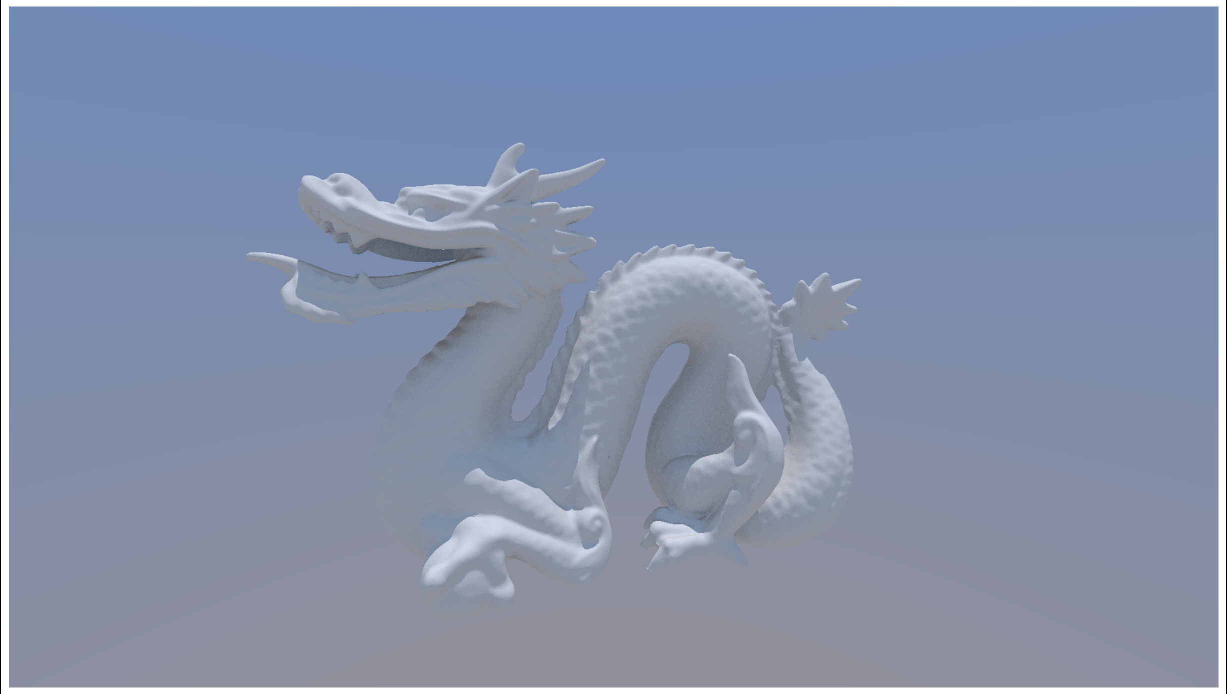


Figura 17 – Cena *Sponza* renderizada em 2m12s à 1920x1080, 24 amostras por *pixel* e 2 reflexões

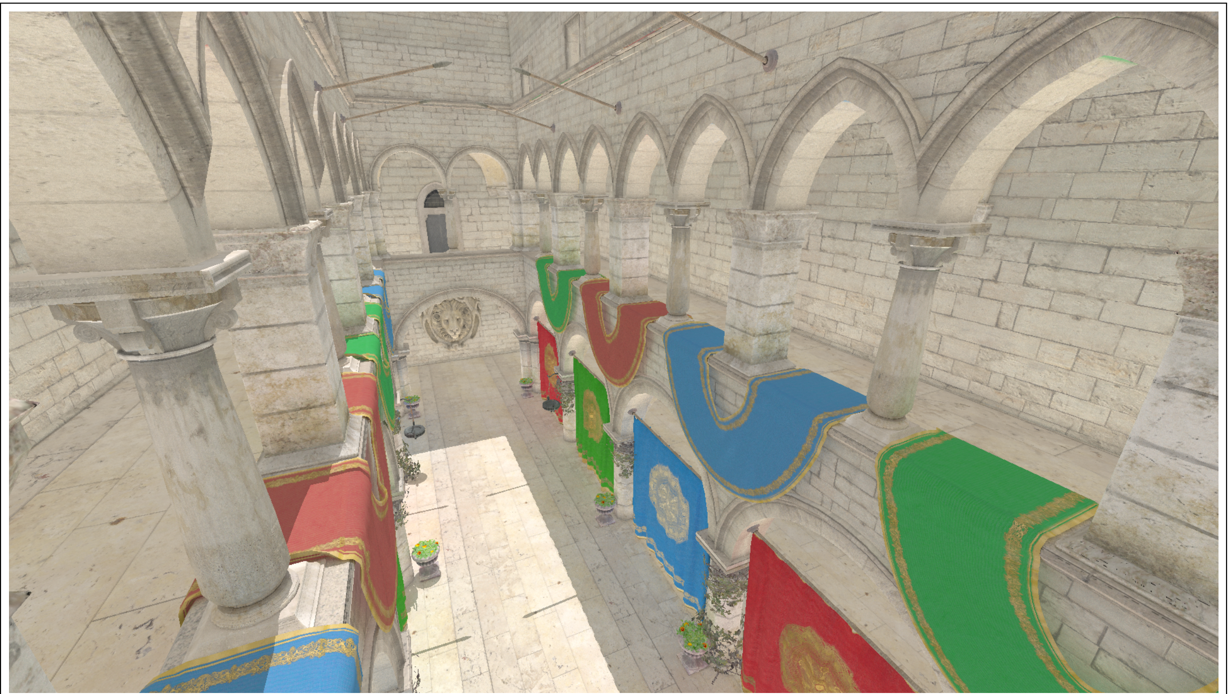


Figura 18 – Cena *Pirate Ship* renderizada em 2m17s à 1920x1080, 24 amostras por *pixel* e 2 reflexões

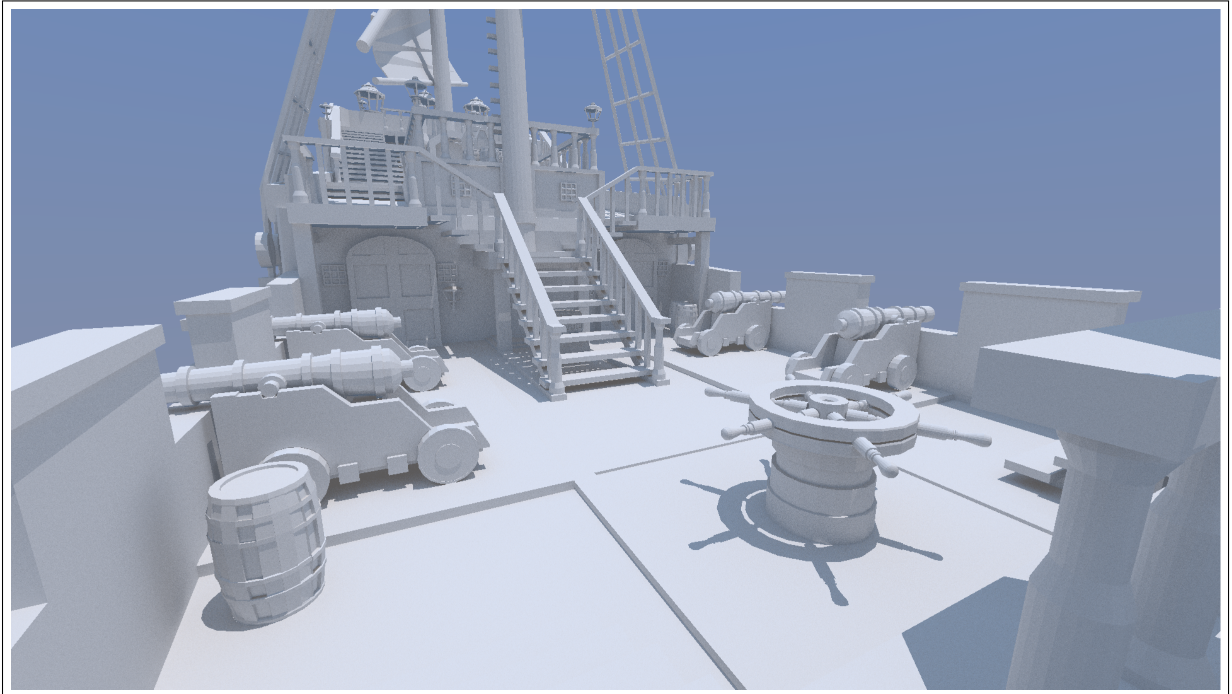
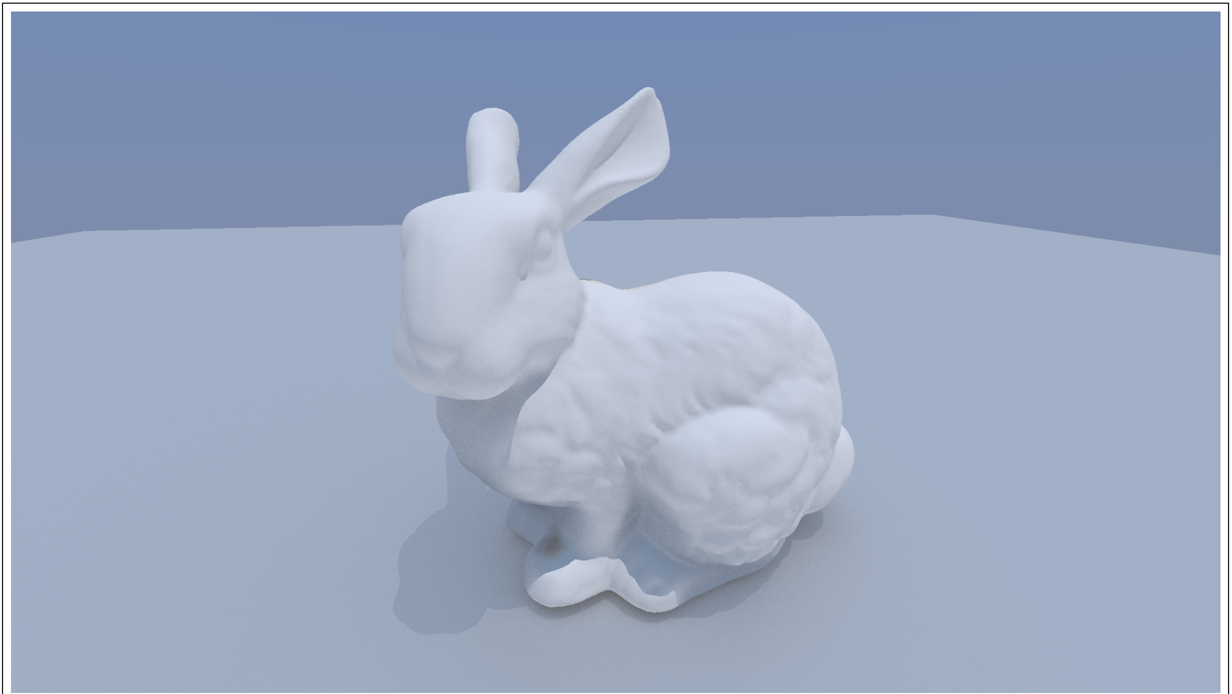


Figura 19 – Cena *Bunny* renderizada em 15s à 1920x1080, 24 amostras por *pixel* e 2 reflexões



5 CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho foram apresentados algoritmos para a representação e computação de OBBs, bem como um algoritmo de interseção de raio com OBB. Foi elaborado um método paralelo de dados para a construção de BVHs para malhas e instâncias. Foram propostos diferentes algoritmos paralelos para a computação da hierarquia de volumes da BVH. Foram apresentados a modelagem e algoritmos paralelos de percurso da BVH de dois níveis, para o nível de baixo e o nível de cima. Finalmente, foi apresentado a implementação desses algoritmos em uma *pipeline* hibrida de *ray tracing* estocástico executada na GPU.

O algoritmo de geração de BVs proposto neste trabalho possibilita a construção de OBBs justas por meio da extração de elementos de escala, orientação e translação de um conjunto de vertices de uma malha. Tais informações são usadas em aplicações de computação gráfica, como jogos e visualização, na forma de matrizes de transformação.

A representação de OBBs por uma matriz4×4 a faz candidata à aceleração SIMD na GPU. Além disso, elas podem servir de entrada para nas sessões subsequentes da *pipeline* para a atualização de OBBs em conjunto com a matriz de transformação das instâncias. A possibilidade de aplicar transformações lineares às OBBs faz o algoritmo proposto mais generalista que outras abordagens tradicionais uma vez que ele oferece suporte para ainda mais possibilidades de transformações lineares incluindo cisalhamento.

A revindicação de que a representação da OBB por uma matriz de transformação 4×4 é eficiente existe na perspectiva em que tal estrutura de dados possui bom alinhamento de memória de 128 bits da GPU. Porém, pode-se argumentar que tal estrutura ainda utilize uma quantidade muito alta de memória.

Trabalhos futuros podem considerar a possibilidade de representação da parte rotacional e de escala uniforme em um único *quaternion* e um adicional 128 bits para a parte translacional. Essa representação reduz em metade o traço de memória da estrutura de representação da OBB, porém introduz a limitação de descartar escalas não uniformes.

A computação da OBB justa mínima para uma dada malha possui uma ordem de complexidade $O(n^3)$ (LARSSON; KäLLBERG, 2011). A escolha do algoritmo de (GOTTSCHALK, 2000) para a computação da OBB justa por meio de PCA permite a modelagem de algoritmos paralelos de construção de BVHs que possam competir com os algoritmos que usam AABBs (LAUTERBACH *et al.*, 2009). Porém, o algoritmo proposto pode gerar *loose-fitting* OBBs e é especialmente susceptível quando opera em malhas simétricas. Algoritmos alternativos,

como os de (LARSSON; KäLLBERG, 2011), podem potencialmente melhorar a eficiência da BVH gerada, uma vez que estes algoritmos rodam em tempo linear e geram OBBs de melhor qualidade.

O proposto de interseção de raio-OBB se baseia na transformação afim do raio para o espaço local da AABB codificada pela matriz da OBB. O algoritmo de interseção faz uso do teorema de eixos separadores e a eficiência de instruções vetoriais, oferecendo melhor eficiência de descarte em comparação a métodos existentes que usam AABB na maioria das BVHs.

O método construção de hierarquias proposto oferece complexidade linear no tempo de construção de BVHs. Sua divisão em múltiplos passes permite a combinação de diferentes algoritmos. O passe de ordenação de primitivas, por exemplo foi implementado em CPU enquanto os demais passes têm implementação em CPU e em GPU. O método de construção de BVHs é capaz de explorar o paralelismo para atingir melhor performance em GPU. Dentre todos os passes testados, os desempenho dos passes executados em GPU possuem tempo de execução menor em relação aos respectivos passes executados em CPU.

O particionamento da cena usando *Morton codes* oferece alto grau de paralelismo, porém gera uma BVH de qualidade inferior. A divisão da geometria em uma grade regular não leva em consideração a distribuição da malha no espaço e o algoritmo ignora a forma da primitiva em sua heurística de particionamento.

Foram apresentados diferentes algoritmos para a computação de BVs para a hierarquia. Cada algoritmo apresenta diferentes graus de qualidade de BVs geradas bem como diferentes graus de granularidade e de desempenho entre si, e entre as diferentes plataformas de execução. Nessa perspectiva, os algoritmos de construção de OBBs em cascata representam um balanço entre a qualidade de geração de volumes e eficiência de construção.

A proposta de uma BVH de dois níveis particiona a cena em um nível lógico, gerando uma hierarquia de instâncias, no nível de cima e uma lista de hierarquias para cada malha no nível de baixo. O nível de cima é gerado a partir do instanciamento e dá início ao algoritmo de percurso da BVH de dois níveis. A construção do nível de cima baseado na heurística de coerência espacial entre objetos de uma cena confere à BVH uma qualidade superior em comparação ao uso de uma BVH comum e também oferece a possibilidade de atualização de cenas dinâmicas a um baixo custo.

A estrutura da BVH de dois níveis permite a modularizadão de diferentes partes da cena. A hierarquia de cada malha pode ser atualizada separadamente, ou em paralelo, sem

a necessidade de sincronização ou barreiras de memória. A atualização seletiva confere um alto grau de balanceamento de desempenho para à estrutura proposta. Diferentes algoritmos de construção de hierarquias podem existir em conjunto na mesma cena. Essa flexibilidade torna a BVH de dois níveis a torna uma opção atrativa para uso em motores gráficos de diferentes especialidades, seja o foco renderização em tempo real ou renderização foto realista *offline*.

Foi apresentado um modelo de *pipeline* hibrida para *ray tracing* estocástico com aceleração gráfica na GPU. Ela combina a *pipeline* de rasterização com a *pipeline compute* do hardware paralelo moderno para a geração de cenas foto realistas usando materiais PBR com efeitos de GI.

A técnica de *deferred rendering* é usada durante a etapa de rasterização para a geração do *g-buffer* que contém informações geométricas e de material da cena em espaço de tela. As informações contidas no *g-buffer* são equivalentes às informações geradas pela simulação de raios primários. O uso do *raster* elimina a necessidade de computação desses raios e proporciona métodos de eficientes filtragem de textura anisotrópica que contribuem para a qualidade da imagem.

O *path tracer* estocástico é executado a partir das informações contidas no *g-buffer*. Ele possui alto grau de cooperação, grupos de processos paralelos compartilham a computação de amostras de cada *pixel* em resolução total da imagem.

A geração de raios usa um algoritmo de geração de números pseudo aleatórios em tempo de execução o qual é simples porém eficiente. A técnica de estratificação proporciona uma melhor distribuição de amostras para algoritmos de geração de números pseudo aleatórios. *Importance sampling* altera as propriedades de distribuição de um conjunto de amostras baseado em modelos probabilísticos a fim de acelerar convergência em métodos estocásticos.

O transporte de luz envolve o percurso da BVH de dois níveis. A técnica de *cone tracing* é utilizada na interseção com a geometria da cena para fins de computação de filtragem isotrópica trilinear de texturas.

O cálculo de iluminação computa a GI da cena por meio da simulação de sombras, *ambient occlusion* e iluminação indireta da a luz ambiente e reflexões difusas.

REFERÊNCIAS

AKENINE-MöLLER, T.; NILSSON, J.; ANDERSSON, M.; BARRÉ-BRISEBOIS, C.; TOTH, R.; KARRAS, T. Texture level of detail strategies for real-time ray tracing. In: **Ray Tracing Gems**. Apress, 2019. p. 321–345. Disponível em: https://doi.org/10.1007/978-1-4842-4427-2_20. Acesso em: 13 jun. 2019.

AMANATIDES, J. Ray tracing with cones. **SIGGRAPH Comput. Graph.**, Association for Computing Machinery, New York, NY, USA, v. 18, n. 3, p. 129–135, jan. 1984. ISSN 0097-8930. Disponível em: https://doi.org/10.1145/964965.808589. Acesso em: 13 jun. 2019.

ANTWERPEN, D. van; SEIBERT, D.; KELLER, A. A simple load-balancing scheme with high scaling efficiency. In: **Ray Tracing Gems**. Apress, 2019. p. 127–133. Disponível em: https://doi.org/10.1007/978-1-4842-4427-2_10. Acesso em: 13 jun. 2019.

BALDWIN, D.; WEBER, M. Fast ray-triangle intersections by coordinate transformation. **Journal of Computer Graphics Techniques (JCGT)**, v. 5, n. 3, p. 39–49, September 2016. ISSN 2331-7418. Disponível em: http://jcgt.org/published/0005/03/03/. Acesso em: 13 jun. 2019.

BARR, A. H. GLOBAL AND LOCAL DEFORMATIONS OF SOLID PRIMITIVES. In: **Readings in Computer Vision**. Elsevier, 1987. p. 661–670. Disponível em: https://doi.org/10.1016/b978-0-08-051581-6.50064-7. Acesso em: 13 jun. 2019.

BARRÉ-BRISEBOIS, C.; HALÉN, H.; WIHLIDAL, G.; LAURITZEN, A.; BEKKERS, J.; STACHOWIAK, T.; ANDERSSON, J. Hybrid rendering for real-time ray tracing. In: **Ray Tracing Gems**. Apress, 2019. p. 437–473. Disponível em: https://doi.org/10.1007/978-1-4842-4427-2_25. Acesso em: 13 jun. 2019.

BENTHIN, C.; WOOP, S.; WALD, I.; ÁFRA, A. T. Improved two-level bvhs using partial re-braiding. In: **Proceedings of High Performance Graphics**. New York, NY, USA: ACM, 2017. (HPG '17), p. 7:1–7:8. ISBN 978-1-4503-5101-0. Disponível em: http://doi.acm.org/10.1145/3105762.3105776. Acesso em: 13 jun. 2019.

BLELLOCH, G. E. **Vector Models for Data-Parallel Computing**. Cambridge, MA, USA: MIT Press, 1990. ISBN 026202313X.

DAVIDOVIČ, T.; ENGELHARDT, T.; GEORGIEV, I.; SLUSALLEK, P.; DACHSBACHER, C. 3d rasterization: A bridge between rasterization and ray casting. In: **Proceedings of Graphics Interface 2012**. Toronto, Ont., Canada, Canada: Canadian Information Processing Society, 2012. (GI '12), p. 201–208. ISBN 978-1-4503-1420-6. Disponível em: http://dl.acm.org/citation.cfm?id=2305276.2305310. Acesso em: 13 jun. 2019.

DOYLE, M. J.; FOWLER, C.; MANZKE, M. A hardware unit for fast sah-optimised bvh construction. **ACM Trans. Graph.**, ACM, New York, NY, USA, v. 32, n. 4, p. 139:1–139:10, jul. 2013. ISSN 0730-0301. Disponível em: http://doi.acm.org/10.1145/2461912.2462025. Acesso em: 13 jun. 2019.

DU, P.; KIM, Y. J.; YOON, S. E. Tss bvhs: Tetrahedron swept sphere bvhs for ray tracing subdivision surfaces. **Comput. Graph. Forum**, The Eurographs Association & John Wiley & Sons, Ltd., Chichester, UK, v. 35, n. 7, p. 279–288, out. 2016. ISSN 0167-7055. Disponível em: https://doi.org/10.1111/cgf.13025. Acesso em: 13 jun. 2019.

ERICSON, C. Merging two obbs. In: **Real-Time Collision Detection**. USA: CRC Press, Inc., 2004. cap. 6.5.3, p. 279. ISBN 1558607323.

GOTTSCHALK, S. A. **Collision Queries Using Oriented Bounding Boxes**. Tese (Doutorado), 2000. AAI9993311.

GRüNSCHLOSS, L.; STICH, M.; NAWAZ, S.; KELLER, A. Msbvh: An efficient acceleration data structure for ray traced motion blur. In: **Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics**. New York, NY, USA: ACM, 2011. (HPG '11), p. 65–70. ISBN 978-1-4503-0896-0. Disponível em: http://doi.acm.org/10.1145/2018323.2018334. Acesso em: 13 jun. 2019.

HALTON, J. H. Algorithm 247: Radical-inverse quasi-random point sequence. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 7, n. 12, p. 701–702, dez. 1964. ISSN 0001-0782. Disponível em: https://doi.org/10.1145/355588.365104. Acesso em: 13 jun. 2019.

IGEHY, H. Tracing ray differentials. In: **Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques**. USA: ACM Press/Addison-Wesley Publishing Co., 1999. (SIGGRAPH '99), p. 179–186. ISBN 0201485605. Disponível em: https://doi.org/10.1145/311535.311555. Acesso em: 13 jun. 2019.

KAJIYA, J. T. The rendering equation. **SIGGRAPH Comput. Graph.**, Association for Computing Machinery, New York, NY, USA, v. 20, n. 4, p. 143–150, ago. 1986. ISSN 0097-8930. Disponível em: https://doi.org/10.1145/15886.15902. Acesso em: 13 jun. 2019.

KARRAS, T. Maximizing parallelism in the construction of bvhs, octrees, and k-d trees. In: **Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics**. Goslar Germany, Germany: Eurographics Association, 2012. (EGGH-HPG'12), p. 33–37. ISBN 978-3-905674-41-5. Disponível em: https://doi.org/10.2312/EGGH/HPG12/033-037. Acesso em: 13 jun. 2019.

KARRAS, T. **Thinking Parallel, Part I: Collision Detection on the GPU**. 2012. Disponível em: https://devblogs.nvidia.com/thinking-parallel-part-i-collision-detection-gpu/. Acesso em: 14 dec. 2019.

KARRAS, T. **Thinking Parallel, Part III: Tree Construction on the GPU**. 2012. Disponível em: https://devblogs.nvidia.com/thinking-parallel-part-iii-tree-construction-gpu/. Acesso em: 14 dec. 2019.

KARRAS, T.; AILA, T. Fast parallel construction of high-quality bounding volume hierarchies. In: **Proceedings of the 5th High-Performance Graphics Conference**. New York, NY, USA: ACM, 2013. (HPG '13), p. 89–99. ISBN 978-1-4503-2135-8. Disponível em: http://doi.acm.org/10.1145/2492045.2492055. Acesso em: 13 jun. 2019.

KAY, T. L.; KAJIYA, J. T. Ray tracing complex scenes. In: **Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques**. New York, NY, USA: ACM, 1986. (SIGGRAPH '86), p. 269–278. ISBN 0-89791-196-2. Disponível em: http://doi.acm.org/10.1145/15922.15916. Acesso em: 13 jun. 2019.

KELLER, A.; VIITANEN, T.; BARRÉ-BRISEBOIS, C.; SCHIED, C.; MCGUIRE, M. Are we done with ray tracing? In: **ACM SIGGRAPH 2019 Courses**. [*S. l.*]: ACM, 2019. p. 1–381.

LARSEN, M.; MEREDITH, J. S.; NAVRATIL, P. A.; CHILDS, H. Ray tracing within a data parallel framework. In: **2015 IEEE Pacific Visualization Symposium (PacificVis)**. IEEE, 2015. Disponível em: https://doi.org/10.1109/pacificvis.2015.7156388. Acesso em: 13 jun. 2019.

LARSSON, T.; KäLLBERG, L. Fast computation of tight-fitting oriented bounding boxes. In: **Game Engine Gems 2**. 1st. ed. [*S. l.*]: A. K. Peters, Ltd., 2011. cap. 1, p. 3–20. ISBN 1568814372, 9781568814377.

LAUTERBACH, C.; GARLAND, M.; SENGUPTA, S.; LUEBKE, D.; MANOCHA, D. Fast BVH construction on GPUs. **Computer Graphics Forum**, Wiley, v. 28, n. 2, p. 375–384, abr. 2009. Disponível em: https://doi.org/10.1111/j.1467-8659.2009.01377.x. Acesso em: 13 jun. 2019.

MAIA, J. G. R.; VIDAL, C. A.; CAVALCANTE-NETO, J. B. Transformation semantics: An efficient approach for collision detection. In: IEEE. **2006 19th Brazilian Symposium on Computer Graphics and Image Processing**. [*S. l.*], 2006. p. 94–104.

MAJERCIK, A.; CRASSIN, C.; SHIRLEY, P.; MCGUIRE, M. A ray-box intersection algorithm and efficient dynamic voxel rendering. **Journal of Computer Graphics Techniques (JCGT)**, v. 7, n. 3, p. 66–81, September 2018. ISSN 2331-7418. Disponível em: http://jcgt.org/published/0007/03/04/. Acesso em: 13 jun. 2019.

MATTAUSCH, O.; BITTNER, J.; JASPE, A.; GOBBETTI, E.; WIMMER, M.; PAJAROLA, R. Chc+rt: Coherent hierarchical culling for ray tracing. **Computer Graphics Forum**, Wiley, v. 34, n. 2, p. 537–548, may 2015. Disponível em: https://doi.org/10.1111/cgf.12582. Acesso em: 13 jun. 2019.

MERRILL, D.; GRIMSHAW, A. HIGH PERFORMANCE AND SCALABLE RADIX SORTING: A CASE STUDY OF IMPLEMENTING DYNAMIC PARALLELISM FOR GPU COMPUTING. **Parallel Processing Letters**, World Scientific Pub Co Pte Lt, v. 21, n. 02, p. 245–272, jun. 2011. Disponível em: https://doi.org/10.1142/s0129626411000187. Acesso em: 13 jun. 2019.

MERRILL, D. G.; GRIMSHAW, A. S. Revisiting sorting for gpgpu stream architectures. In: **Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques**. New York, NY, USA: ACM, 2010. (PACT ’10), p. 545–546. ISBN 978-1-4503-0178-7. Disponível em: http://doi.acm.org/10.1145/1854273.1854344. Acesso em: 13 jun. 2019.

MüLLER, M.; BENDER, J.; CHENTANEZ, N.; MACKLIN, M. A robust method to extract the rotational part of deformations. In: **Proceedings of the 9th International Conference on Motion in Games**. New York, NY, USA: Association for Computing Machinery, 2016. (MIG ’16), p. 55–60. ISBN 9781450345927. Disponível em: https://doi.org/10.1145/2994258.2994269. Acesso em: 13 jun. 2019.

MöLLER, T.; TRUMBORE, B. Fast, minimum storage ray-triangle intersection. **Journal of Graphics Tools**, Taylor & Francis, v. 2, n. 1, p. 21–28, 1997. Disponível em: https://doi.org/10.1080/10867651.1997.10487468. Acesso em: 13 jun. 2019.

PARKER, S. G.; BIGLER, J.; DIETRICH, A.; FRIEDRICH, H.; HOBEROCK, J.; LUEBKE, D.; MCALLISTER, D.; MCGUIRE, M.; MORLEY, K.; ROBISON, A.; STICH, M. Optix: A general purpose ray tracing engine. **ACM Trans. Graph.**, ACM, New

York, NY, USA, v. 29, n. 4, p. 66:1–66:13, jul. 2010. ISSN 0730-0301. Disponível em: http://doi.acm.org/10.1145/1778765.1778803. Acesso em: 12 jun. 2019.

PHARR, M. Efficient generation of points that satisfy two-dimensional elementary intervals. **Journal of Computer Graphics Techniques (JCGT)**, v. 8, n. 1, p. 56–68, February 2019. ISSN 2331-7418. Disponível em: http://jcgt.org/published/0008/01/04/. Acesso em: 13 jun. 2019.

PHARR, M. On the importance of sampling. In: **Ray Tracing Gems**. Apress, 2019. p. 207–222. Disponível em: https://doi.org/10.1007/978-1-4842-4427-2_15. Acesso em: 13 jun. 2019.

SATISH, N.; HARRIS, M.; GARLAND, M. Designing efficient sorting algorithms for manycore gpus. In: **Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing**. Washington, DC, USA: IEEE Computer Society, 2009. (IPDPS '09), p. 1–10. ISBN 978-1-4244-3751-1. Disponível em: https://doi.org/10.1109/IPDPS.2009.5161005. Acesso em: 13 jun. 2019.

SCHANDER, T. **Deferred Path Tracing By Enscape**. 2017. Disponível em: https://gpuopen.com/deferred-path-tracing-enscape/. Acesso em: 12 fev. 2020.

STICH, M.; FRIEDRICH, H.; DIETRICH, A. Spatial splits in bounding volume hierarchies. In: **Proc. High-Performance Graphics 2009**. [*S. l.*: *s. n.*], 2009.

SUBTIL, N. **Introduction to Real-Time Ray Tracing with Vulkan**. 2018. Disponível em: https://devblogs.nvidia.com/vulkan-raytracing/. Acesso em: 13 jun. 2019.

TATARCHUK, N.; LEFOHN, A. Open problems in real-time rendering. In: **ACM SIGGRAPH 2017 Courses**. New York, NY, USA: ACM, 2017. (SIGGRAPH '17). ISBN 978-1-4503-5014-3. Disponível em: http://doi.acm.org/10.1145/3084873.3127940. Acesso em: 13 jun. 2019.

VIVO, P. G.; LOWE, J. **Generative designs**. 2015. Disponível em: https://thebookofshaders.com/10/. Acesso em: 2019-06-14.

WäCHTER, C.; BINDER, N. A fast and robust method for avoiding self-intersection. In: **Ray Tracing Gems**. Apress, 2019. p. 77–85. Disponível em: https://doi.org/10.1007/978-1-4842-4427-2_6. Acesso em: 13 jun. 2019.

WHITTED, T. An improved illumination model for shaded display. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 23, n. 6, p. 343–349, jun. 1980. ISSN 0001-0782. Disponível em: https://doi.org/10.1145/358876.358882. Acesso em: 13 jun. 2019.

WILLBERGER, T.; MUSTERLE, C.; BERGMANN, S. Deferred hybrid path tracing. In: **Ray Tracing Gems**. Apress, 2019. p. 475–492. Disponível em: https://doi.org/10.1007/978-1-4842-4427-2_26. Acesso em: 13 jun. 2019.

WILLIAMS, A.; BARRUS, S.; MORLEY, R. K.; SHIRLEY, P. An efficient and robust ray-box intersection algorithm. **Journal of graphics tools**, Taylor and Francis, v. 10, n. 1, p. 49–54, 2005.

WYMAN, C.; MARRS, A. Introduction to DirectX raytracing. In: **Ray Tracing Gems**. Apress, 2019. p. 21–47. Disponível em: https://doi.org/10.1007/978-1-4842-4427-2_3. Acesso em: 13 jun. 2019.