UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Vítor Peixoto Menezes

RAY TRACING EM TEMPO REAL USANDO HIERARCHICAL SCENE SIGNED DISTANCE FIELDS

Santa Maria, RS
2019

Vítor Peixoto Menezes

RAY TRACING EM TEMPO REAL USANDO HIERARCHICAL SCENE SIGNED DISTANCE FIELDS

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação, Área de Concentração em 1.03.03.05-7, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de
Mestre em Ciência da Computação.

ORIENTADOR: Prof. Cesar Tadeu Pozzer

Santa Maria, RS
2019

Menezes, Vitor Peixoto
Ray Tracing em Tempo Real Usando Hierarchical Scene Signed Distance Fields / Vitor Peixoto Menezes.- 2019.
56 p.; 30 cm

Orientador: Cesar Tadeu Pozzer
Dissertação (mestrado) - Universidade Federal de Santa Maria, Centro de Tecnologia, Programa de Pós-Graduação em Ciência da Computação , RS, 2019

1. Rendering 2. Signed Functions 3. Ray tracing I. Pozzer, Cesar Tadeu II. Título.

Sistema de geração automática de ficha catalográfica da UFSM. Dados fornecidos pelo autor(a). Sob supervisão da Direção da Divisão de Processos Técnicos da Biblioteca Central. Bibliotecária responsável Paula Schoenfeldt Patta CRB 10/1728.

End. Eletr.: vmenezes@inf.ufsm.br

Vítor Peixoto Menezes

RAY TRACING EM TEMPO REAL USANDO HIERARCHICAL SCENE SIGNED DISTANCE FIELDS

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação, Área de Concentração em 1.03.03.05-7, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de
Mestre em Ciência da Computação.

Aprovado em 25 de março de 2019:

Cesar Tadeu Pozzer, Dr. (UFSM)
(Presidente/Orientador)

Benhur de Oliveira Stein, Dr. (UFSM)

João Luiz Dihl Comba, Dr. (UFRGS)

Santa Maria, RS
2019

RESUMO

RAY TRACING EM TEMPO REAL USANDO HIERARCHICAL SCENE SIGNED DISTANCE FIELDS

AUTOR: Vítor Peixoto Menezes
ORIENTADOR: Cesar Tadeu Pozzer

O *ray tracing* é uma importante técnica para a obtenção de imagens foto-realísticas. O algoritmo básico é bem conhecido e consiste em traçar raios a partir da posição de uma câmera sintética calculando a cor dos objetos colididos com os raios. O número de raios varia, mas normalmente é proporcional a quantidade de pixeis, o que eleva a complexidade da técnica por se tratar de um grande número de colisões entre raios e objetos. Portanto, métodos de aceleração são necessários. Recentemente, foram propostas soluções que mudam o modo de representação de um objeto, fato inédito até então. Estas soluções utilizam o *signed distance fields* (SDF) para representar um objeto. Deste modo, usando uma abordagem hierarquizada de *signed distance fields* este trabalho apresenta uma nova maneira de representar uma cena estática que é capaz de diminuir o tempo de execução da geração da imagem. O algoritmo introduz um esquema que subdivide a cena por meio de *hash* espacial e *octree* em uma etapa de pré-processamento e armazena-os em disco para uso posterior. Inicialmente, cada malha trigonométrica da cena deve ser transformado para uma representação *signed distance fields* e após isso, a hierarquia de toda cena deve ser criada. Ao iniciar o modo de renderização, a estrutura previamente calculada é carregada do disco e utilizada para renderizar a cena baseando-se nos algoritmos *sphere tracing* e *signed distance fields*.

Palavras-chave: Rendering. Signed Functions. Ray tracing.

ABSTRACT

REAL TIME RAY TRACING USAING HIERARCHICAL SCENE SIGNED DISTANCE FIELDS

AUTHOR: Vítor Peixoto Menezes
ADVISOR: Cesar Tadeu Pozzer

Ray tracing is an important technique for obtaining photo-realistic images. The basic algorithm is well known and consists of tracing rays from a synthetic camera position by calculating the color of objects colliding with the rays. The number of rays varies, but it is usually proportional to pixel amount, which raises the complexity of ray tracing technique because it deals with a large number of collisions between rays and objects. Therefore, acceleration methods are required. Recently, proposed solutions changed the way that an object is represented. These solutions use the signed distance field (SDF) to represent an object. Thus, using a hierarchical signed distance fields approach, our work presents a new manner of representing a static scene that is capable of decreasing the image generation time. The algorithm introduces a pre-processing scheme that subdivides the scene and stores it on disk for later use. Initially, each object must be transformed to a signed distance field representation and after that, the hierarchy of the scene must be created and saved to disk. During rendering, the previously generated structure is loaded from disk and used to calculate the ray tracing.

Keywords: Rendering. Signed Functions. Ray tracing.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

SDF	Signed Distance Field
BVH	Bounding Volume Hierarchy
OBB	Oriented Bounding Box
GPU	Graphics Processing Unit
$AABB$	Axis-Aligned Bounding Box
$HLSL$	High Level Shading Language
CPU	Central Processing Unit

SUMÁRIO

1 INTRODUÇÃO

O *ray tracing* é um algoritmo de síntese de imagens capaz de produzir cenas com alta fidelidade, porém com um alto custo computacional. O algoritmo básico, apresentado por Whitted (1979), é bem conhecido e consiste em traçar raios a partir da posição de uma câmera sintética calculando recursivamente a cor dos pixeis referente à colisão dos raios lançados. O resultado desse algoritmo é uma imagem capaz de simular automaticamente efeitos ópticos como reflexão, refração e dispersão. Uma análise mais detalhada do algoritmo é apresentada no Capítulo 3.3.3. Com o alto custo computacional do algoritmo, diversas técnicas foram propostas para acelerar seu processamento. No Capítulo 2 é apresentado as técnicas de aceleração do *ray tracing* que são utilizadas até hoje.

Diversos trabalhos na área de computação gráfica apresentam o uso do paradigma matemático *signed functions* como meio de otimizar o *ray tracing* (CRASSIN et al., 2009; JAMRIŠKA, 2010), originalmente proposto por Inigo (2008). Uma determinada área ao redor de um objeto que usa *signed functions* para representar sua geometria é chamada de *signed distance field* (SDF). O SDF modifica completamente a maneira em que uma malha é representada ao determinar a distância de um dado ponto P dentro da área amostrada à superfície Ω, com o sinal determinando se P está dentro ou fora de Ω. Combinando essa representação com o algoritmo de *ray marching*, é possível encontrar a superfície mais próxima de maneira intrínseca. O algoritmo de *ray marching* utiliza um ponto inicial e uma direção lançando um raio de tamanho calculado por uma função e itera em N passos até alcançar um limite estipulado (N máximo ou distância total máxima). Nesta dissertação, tais técnicas são utilizadas como base do cálculo da imagem final e são apresentadas nos Capítulos 3.1 e 3.3.1 respectivamente.

A precisão do SDF de um objeto impacta diretamente na qualidade de renderização e, caso feito de maneira grosseira, utilizará uma grande quantidade de memória. A construção do SDF de uma primitiva básica (esfera, cubo, toro, entre outros) pode ser facilmente reproduzida pela fórmula matemática de sua geometria (QUILEZ, 2008a), porém, também é necessário poder representar objetos de malhas genéricas. Assim, diversas maneiras eficientes de criar um SDF para objetos de malhas genéricas foram propostas (KOSCHIER; DEUL; BENDER, 2016; JAMRIŠKA, 2010; FRISKEN et al., 2000). Neste trabalho, o SDF de malhas genéricas é mapeado em uma estrutura *hash* no entorno do objeto, onde os triângulos são distribuídos na estrutura de acordo com seu volume no espaço. A geração do SDF de malhas genéricas é apresentado no Capítulo 5.1.

Para utilizar SDFs na renderização de uma cena grande e complexa, deve-se ser capaz de analisar qualquer ponto do mundo e obter informação da superfície mais próxima. Para isso, é utilizado uma estrutura que mapeia os objetos para serem selecionados para o cálculo de distância. A estrutura é gerada a partir do pré-processamento do espaço

usando técnicas de subdivisão espacial. Assim, a abordagem proposta visa criar uma estrutura que engloba todos os SDFs dos objetos da cena, organizando-os de uma maneira que facilite sua utilização e guardando-a em disco. Após isso, a estrutura é enviada para a GPU para ser utilizada pelo algoritmo de *ray tracing* gerando uma imagem que representa a cena com um custo computacional menor comparado ao estado da arte.

1.1 OBJETIVOS

O principal objetivo desta dissertação é propor e implementar uma nova abordagem otimizada de cálculo de intersecção entre raio e malhas em GPU. Mais especificamente:

- Avaliar soluções modernas de subdivisão espacial.
- Estudar abordagens otimizadas de renderização.
- Avaliar soluções modernas de cálculo de intersecção entre raio e malha.
- Avaliar o uso de SDF e suas aplicações para cálculo de intersecção.
- Propor e implementar uma solução escalável em nível de cena para cálculo de intersecção baseada em GPU.
- Avaliar vantagens e desvantagens da mudança de paradigma de representação de objetos.
- Implementar um sistema de *ray tracing* baseado na estrutura proposta, a fim de verificar performance.

1.2 CONTRIBUIÇÕES

A principal contribuição dessa pesquisa é uma solução baseada em GPU para o cálculo otimizado de intersecção entre raio e geometria utilizando SDF para representar objetos. Demonstra-se como associar técnicas de subdivisão espacial com a distribuição de SDFs de acordo com o espaço de influência do mesmo. O algoritmo pré-processa informações e armazena-as em uma estrutura de aceleração que melhora a performance do cálculo de intersecção. A solução leva em consideração uma ampla gama de tópicos e suas aplicações para o cálculo intersecção.

1.3 ORGANIZAÇÃO DA DISSERTAÇÃO

A dissertação está organizada da seguinte maneira: o Capítulo 2 apresenta trabalhos relacionados com os assuntos abordados neste trabalho. Revisa-se os conceitos básicos relacionados às técnicas utilizadas no Capítulo 3. O Capítulo 4 apresenta a arquitetura da abordagem proposta, detalhando as estruturas de dados utilizadas. O algoritmo de criação da estrutura e o de renderização estão explicados no Capítulo 5. O Capítulo 6 mostra os resultados obtidos em relação ao desempenho e à qualidade visual. Finalmente, no Capítulo 7, discutimos a solução proposta e fornecemos descrições para futuras direções de pesquisa e melhorias de desempenho.

2 TRABALHOS RELACIONADOS

Neste capítulo são apresentado os trabalhos que trazem otimizações para o cálculo do *ray tracing* que influenciaram no desenvolvimento do trabalho. Está organizado em duas sessões, o Capítulo 2.1 que aborda estruturas planares e no Capítulo 2.2 que apresenta as estruturas hierárquicas.

2.1 ESTRUTURAS PLANARES

Estruturas planares são as estruturas de dados mais simples de otimização do *ray tracing*. Existem duas separações comuns de estruturas: a primeira define propriedades exclusivamente dos objetos; e a segunda subdivide a cena em regiões menores, reduzindo a região a ser analisada.

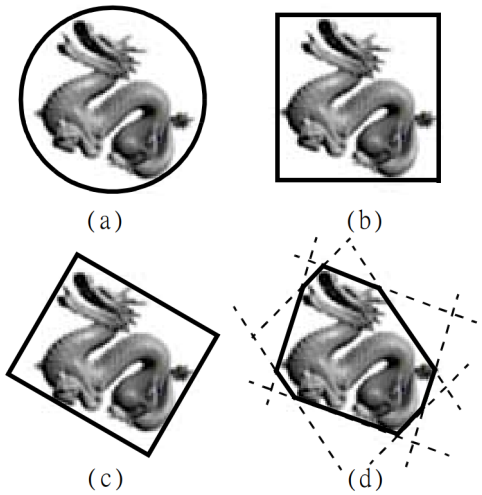
2.1.1 Bounding Volumes

Os *bouding volumes* definem propriedades exclusivas do objeto, mais especificamente visam englobar o espaço na qual a malha do objeto está presente. O motivo para utilizar um *bounding volume* ao redor do objeto é para reduzir o número de testes de intersecções raio-objeto. Durante o passo de análise da colisões de um raio com objetos, caso não exista colisão com nenhuma *bounding volume*, o raio também não colide com os objeto. Por esse motivo, os *bounding volumes* normalmente têm um formato simples mas que envolve completamente o objeto.

Encontrar o *bounding volume* ótimo de malhas trigonométricas genéricas não é um problema trivial (WEGHORST; HOOPER; GREENBERG, 1984). Whitted (WHITTED, 1979) utiliza *bounding volumes* com formato de esfera para cada objeto por sua simplicidade de representação e para a facilitar o cálculo de intersecção. Antes de sua utilização, os *ray tracers* utilizavam quase todo o tempo processando intersecções entre raio e objetos (APPEL, 1968), demonstrando que os *bounding volumes* aceleram o *ray tracing* básico.

Há diversos tipos de *bounding volumes*. A Figura 2.1 identifica os *bounding volumes* mais utilizados para melhorar o desempenho do cálculo de intersecção entre raio-objeto (GLASSNER, 1989). Eles são: esfera, *axis-align bouding box* (AABB), *oriented bouding box* (OBB), e *slabs*. O dragão representado consiste em 1132830 triângulos. O método força-bruta para determinar se o raio intersecta com o dragão é verificar se o raio colidiu com cada um dos triângulos da geometria. Se o raio não colide com nenhum dos triângulos, conclui-se que o raio não colide com o dragão.

Figura 2.1 – *Bounding volumes* mais utilizados: (a) esfera, (b) *axis-aligned bouding box* (AABB), (c) *oriented bouding box* (OBB), (d) four *slabs*.



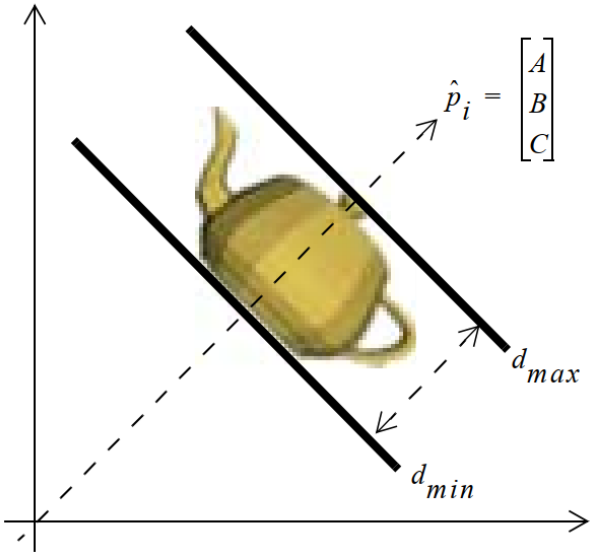
A esfera é a geometria mais rápida e fácil para verificar intersecção com um raio. O raio da esfera é calculado usando a distância máxima do centro do objeto até o vértice mais distante desse ponto possibilitando essa estrutura ser calculada linearmente com complexidade O(k). Whitted (WHITTED, 1979) escolheu esferas para sua abordagem por esse motivo. A principal desvantagem de utilizar esferas é porque normalmente a representação não engloba o objeto compactamente, apresentando grandes espaços vazios. Porém, para um maior ganho em performance é de extrema importância utilizar abordagens que englobam o objeto compactamente.

Para solucionar o problema de grandes espaços vazios ocasionados pelo uso de esferas para representar o volume do objeto, Kaya e Kajiya (KAY; KAJIYA, 1986) propõem o uso de *slabs*. A Figura 2.2 apresenta um *teapot* encapsulado por um *slab* definido por dois planos paralelos. Dado um objeto o no espaço tridimensional e um plano com o vetor normal $\binom{A}{B}{C}$, o *slab* para o objeto o é a região entre dois planos definidos pela função implícita $Ax + By + Cz - d = 0$, onde $d = d_{min}$ ou d_{max} são as distâncias positivas ou negativas do plano à partir da origem.

Para definir um *bounding volume* no espaço tridimensional, é necessário pelo menos três *slabs* (um para cada eixo), gerando uma malha convexa. Para evitar o processamento de *slabs* ótimo para cada objeto, Kay e Kajiya (KAY; KAJIYA, 1986) pré-selecionam sete amostras de vetores normais, enquanto Klosowski et al. (KLOSOWSKI et al., 1998) utilizam treze direções globais. As direções pré-selecionadas dos *slabs* são fixas independente do objeto. Essa abordagem melhor encapsula o objeto comparado com esferas, porém o custo do teste de intersecção entre o raio e malha convexa é alto. Além disso, calcular os valores dos vetores normais para cada objeto não é trivial.

Uma abordagem intermediária entre um volume compacto e facilidade de computação é a *axis-aligned bouding box* (AABB), descrita por Youssef (YOUSSEF, 1986) e Haine

Figura 2.2 – Representação de um *slab* representado pela origem do objeto, vetor normal e as distâncias positivas e negativas à partir da origem.



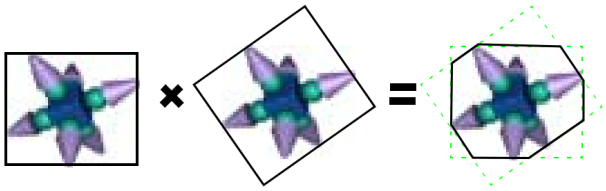
(HAINES; GREENBERG, 1986). Essa abordagem não gera um volume tão compacto quando o *slab*, porém o custo computacional da AABB é mais barato em termos de tempo e espaço. Caso um volume compacto seja necessário, uma *oriented bouding box* (OBB) pode ser utilizada. Ao contrário da AABB, a orientação da OBB depende da orientação do objeto. OBBs são extensamente utilizadas em *ray tracing* (ARVO; KIRK, 1989) e detecção de colisão (ZACHMANN, 1995).

A OBB encapsula o objeto mais compacto comparado com AABB, porém com um custo extra de transformação para cada teste de raio-volume. Para calcular a AABB de um objeto com k vértices, basta encontrar os valores máximos e mínimos em cada coordenada de cada eixo com complexidade $O(k)$. A OBB de volume mínimo não é trivial de calcular. O'Rourke (O'ROURKE, 1985) apresenta um algoritmo com complexidade $O(k^3)$ para computar a OBB com volume mínimo para k vértices na $\mathbb{R}^3$. Barequet e Har-Peled (BAREQUET; HAR-PELED, 2001) melhoraram o desempenho provando que existe um algoritmo que obtém a aproximação da OBB com volume mínimo com complexidade $O(k \log^2 k)$. Uma versão randomizada do algoritmo resolve o problema em complexidade $O(k \log k)$. Outros tipos de *bounding volumes* como cones (SAMET, 1990), prisma (BAREQUET et al., 1996) e *cheesecake* (KAJIYA, 1983) podem ser utilizado em casos especiais. A maior parte desses *bounding volumes* podem ser aproximados usando algoritmos heurísticos em complexidade $O(k)$.

É possível utilizar mais de uma abordagem de *bounding volumes* para representar o objeto. Uma estrutura híbrida normalmente envolve melhor o objeto. Assim, menos intersecções raio-objeto são necessárias. Kaya e Kajiya (KAY; KAJIYA, 1986) descrevem uma maneira de combinar uma AABB e uma OBB em conjunto. O objeto fica contido na intersecção desses dois *bounding volumes*. Porém, o custo do cálculo de intersecção raio-

bounding volume aumenta proporcionalmente a complexidade da geometria da mesma. A Figura 2.3 apresenta um objeto no interior de ambas AABB e OBB.

Figura 2.3 – Estrutura híbrida obtida através da intersecção de uma AABB com uma OBB.



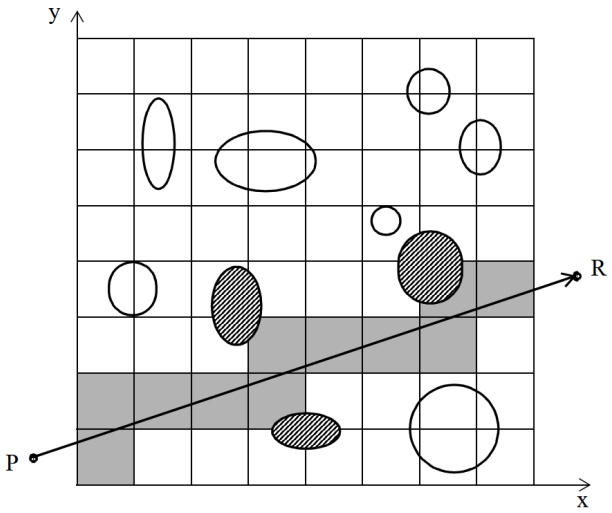
2.1.2 Grids Uniformes

Abordagens que subdividem o espaço para *ray tracing* são normalmente implementadas utilizando *grids* uniformes. Ao dividir a cena em N_x, N_y e N_z intervalos nos três eixos, a cena tridimensional é repartida em $N_x \times N_y \times N_z$ células alinhadas com os eixos. Para facilitar a análise de complexidade, normalmente é assumido que $N_x = N_y = N_z = N$ (CAZALS; DRETTAKIS; PUECH, 1995; CAZALS; PUECH, 1997). O espaço, então, é particionado em células em formato de cubo alinhadas com os eixos globais. Dividir a cena em formato de *grid* é simples e eficiente e a escolha do tamanho da *grid* é um fator importante que pode afetar a velocidade do *ray tracing*.

Utilizar *grids* uniformes como subdivisão espacial para *ray tracing* foi proposto por Fujimoto e Iwata (FUJIMOTO; IWATA, 1985) como uma alternativa mais eficiente que *octree*. A ideia básica é eliminar a busca no interior do nó da *octree* (Capítulo 2.2.4). Cada célula representa um *voxel*. O cálculo de intersecção de raio-objeto é somente feito quando há objetos referenciados pelo *voxel*. A Figura 2.4 ilustra uma *grid* uniforme em duas dimensões, dividindo a cena em 8×8 *voxels*, com dez objetos representados pelos elipses. Um raio R originando do ponto p atravessa a cena sem colidir com nenhum objeto. As células hachuradas representam *voxels* que foram intersectadas pelo raio. Somente objetos que intersectam os *voxels* hachurados devem realizar o teste de intersecção raio-objeto. Nesse exemplo, somente três dos dez objetos realizam o teste de intersecção raio-objeto.

Fujimoto et al. (FUJIMOTO; IWATA, 1985; FUJIMOTO; TANAKA; IWATA, 1986) chama a *grid* uniforme de SEADS (*Spatially Enumerated Auxiliary Data Structure*). É utilizado um lista tridimensional para mapear os correspondentes *voxels* da cena. No estágio de pré-processamento, as informações sobre os objetos da cena são armazenados no elemento da lista correspondente ao *voxel* em que o objeto está contido. Essa estrutura é independente do formato e topologia dos objetos e baseando-se somente na resolução da *grid* uniforme. Porém, encontrar a resolução ideal da *grid* não é uma tarefa trivial e é um tópico de pesquisa até hoje.

Figura 2.4 – Representação de uma *grid* uniforme em duas dimensões. As elipses representam objetos. Os objetos hachurados representam objetos onde é necessário calcular a intersecção raio-objeto. Células hachuradas representam células intersectadas pelo raio.



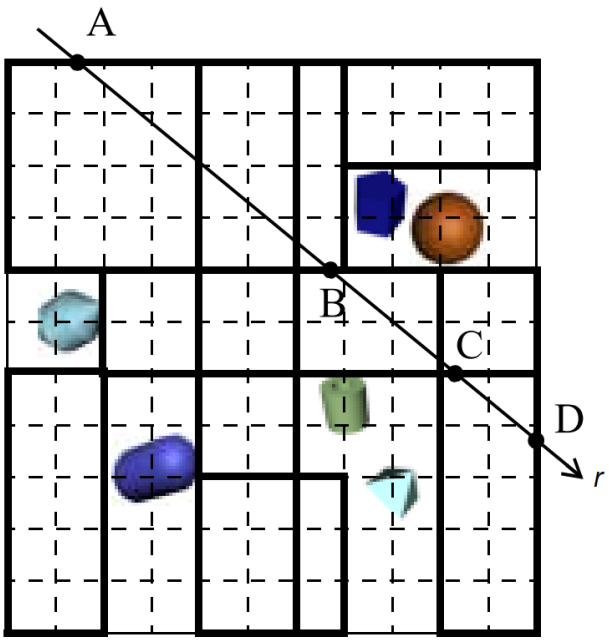
Uma estrutura de dados similar que também utiliza um lista tridimensional para armazenar informações dos objetos é proposta por Yagel et al. (YAGEL; COHEN; KAUFMAN, 1992). A resolução da *grid* escolhida é igual a resolução da cena, ou seja cada célula tem resolução $N \times N \times N$ onde N é a menor unidade de distância. No estágio de pré-processamento, toda geometria é convertida em uma representação discreta. Cada elemento do lista representa um *voxel* da mesma maneira que um lista bidimensional do raster representa uma imagem. Como cada *voxel* é a menor unidade possível, somente um único objeto é permitido em cada *voxel*, eliminando a necessidade de armazenar uma lista de objetos por *voxel*. Além disso, cada *voxel* contém informações da superfície contida, como normal, textura, etc. Entretanto, o uso de memória dessa estrutura é muito superior ao se comparar com outras abordagens. Assim, Yagel et al. assumem que o uso de memória não será problema no futuro e considera somente a velocidade de cálculo de intersecção.

Outra abordagem que utiliza *voxels* vazios é proposto por Devillers (DEVILLERS, 1989). Durante o pré-processamento, é construída uma lista de AABBs chamadas de *macro-regions*. Cada *macro-region* é a combinação de AABBs vazias máxima, ilustrado na Figura 2.5. Um raio é capaz de ignorar diversos *voxels* vazios ao utilizar informações das *macro-regions* cruzando a cena com um número de intersecções reduzidos.

2.2 ESTRUTURAS HIERÁRQUICAS

As estruturas hierárquicas particionam hierarquicamente os objetos, a fim de facilitar cálculo de intersecção com raio. Um raio deve além de avaliar o espaço tridimensional,

Figura 2.5 – Representação do caminho onde um raio r cruza uma *grid* (linhas pretas pontilhadas) com adição de *macro-regions* (linhas pretas mais aparentes) identificando intersecções entre *macro-regions* representado pelas letras maiúsculas.

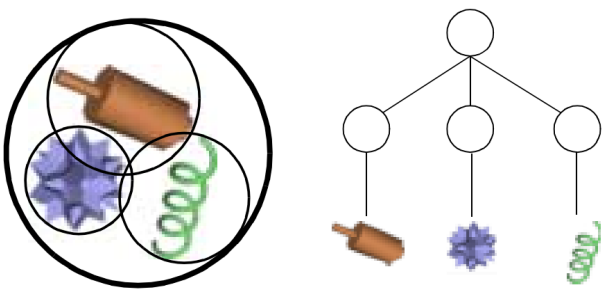


deve também avaliar verticalmente a estrutura.

2.2.1 Bounding Volume Hierarchies

Uma *Bounding Volume Hierarchy* (BVH) é uma árvore. Cada nó intermediário da árvore também é uma *bounding volume* que envolve todos os volumes dos nós em níveis inferiores. Um nó folha indexa um objeto. A Figura 2.6 exemplifica uma BVH de dois níveis. Na esquerda, cada objeto é envolto por uma esfera. A esfera maior engloba todas as esferas menores, podendo ser identificada como pai e as três esferas menores seus filhos. A estrutura conceitual da árvore é ilustrada na direita.

Figura 2.6 – Representação de uma BVH de dois níveis. Cada nó intermediário é também uma *bounding volume*. Cada nó folha identifica um objeto.



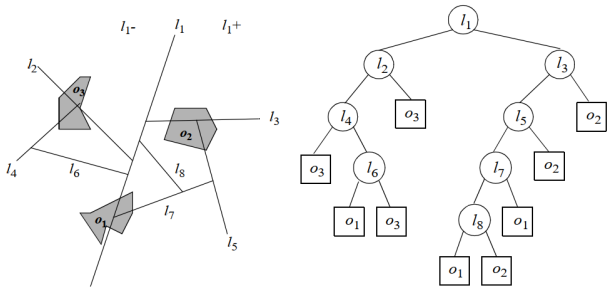
Para calcular intersecção, deve-se manter uma lista de objetos atingidos pelo raio

pois múltiplos *bounding volumes* podem se sobrepôr. Após encontrar todos os objetos atingidos pelo raio, escolhe-se o mais próximo da lista. Fuchs et al. (JR, 2008) sugerem manter a lista com um tamanho máximo de oito para não impactar o desempenho. Embora a performance da BVH seja inferior quando comparado com *grids* uniformes, esse método é bastante utilizado para raios de sombra (HAINES, 1991). Nesse caso, somente é necessário encontrar se há algum objeto entre a fonte luminosa e o ponto sendo avaliado. Caso positivo, conclui-se que esse ponto recebe sombra.

2.2.2 BSP-trees

A árvore de particionamento binário de espaço (*BSP-tree*) foi originalmente proposta por Fuchs et al. (FUCHS; KEDEM; NAYLOR, 1980) para determinar as superfícies visíveis de uma cena composta por um conjunto de polígonos. A abordagem ordena os polígonos da cena de acordo com a proximidade com a câmera. A estrutura de uma *BSP-tree* para *ray tracing* geralmente subdivide a cena em duas partes divididas por um plano na direita e outro na esquerda de acordo com a Figura 2.7. Esse passo de subdivisão se repete até que limites estipulados sejam alcançados, como um valor limite ou nível máximo. Cada região gerada pela subdivisão da BSP-tree é um poliedro convexo.

Figura 2.7 – Exemplo de uma *BSP-tree* gerada à partir de uma cena bidimensional. A cena é subdividida por linhas em duas regiões, positiva e negativa subsequentemente.

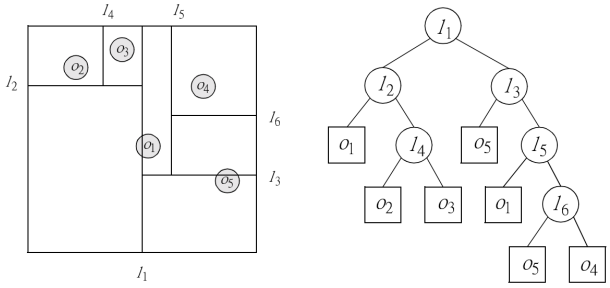


As *BSP-trees* são amplamente utilizadas em diversas áreas como remoção de superfícies não-visíveis (BERG et al., 1994; JAMES, 1999; MURALI, 1999), detecção de colisão (NAYLOR, 1992), planejamento de movimento (WADE, 2001), *ray casting* (DELFOSSE; HEWITT; MÉRIAUX, 1994; AR; CHAZELLE; TAL, 2000) e jogos como DOOM e Quake (DOOM, 2000). Aplicações para renderização utilizando *ray tracing* normalmente utilizam *axis-aligned BSP-trees* para facilitar teste de intersecção raio-caixa (WOO, 1990; AKENINE-MOLLER; HAINES; HOFFMAN, 2018).

2.2.3 k-D trees

A k-D *tree* foi introduzida por Bentley (BENTLEY, 1975) como uma árvore para busca binária em espaços multidimensionais associativos. O símbolo k identifica a dimensionalidade do espaço de busca. As k-D *trees* são um caso especial da *BSP-tree*. A principal diferença entre as duas é que a k-D *tree* restringe a direção do plano de subdivisão. Os planos que subdividem uma *BSP-trees* podem ser orientados arbitrariamente, enquanto a k-D *tree* deve ser alinhados com os eixos do mundo (Figura 2.8). Assim, esta estrutura de dados é bastante utilizada para resolver problemas de busca em k dimensões ortogonais(AGARWAL; ERICKSON et al., 1999; AGARWAL, 1997; BENTLEY; FRIEDMAN, 1979).

Figura 2.8 – Exemplo de uma k-D *tree* gerada a partir de uma cena bidimensional. A cena é subdividida por linhas ortogonais em duas regiões, positiva e negativa subsequentemente.

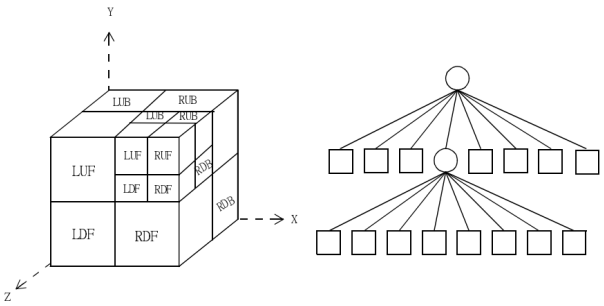


2.2.4 Octrees

Uma *octree* consiste em subdividir o espaço tridimensional em oito octantes - também chamados de célula, nó ou caixa - recursivamente até satisfazer as condições de parada. As condições de parada normalmente utilizadas são tamanho mínimo de célula, nível máximo de profundidade ou quantidade máxima de objetos. Caso alguma das condições não sejam satisfeitas, a *octree* continua a subdivisão. A abordagem tradicional somente armazena os objetos nos nós em profundidade máxima (GLASSNER, 1984), também chamados de nós folha. A Figura 2.9 apresenta uma *octree* de dois níveis. Cada octante está nomeado com uma das seguintes siglas LUF, LUB, LDF, LDB, RUB, RDF e RDB que representam a posição do nó relativa ao nível anterior. Cada letra da sigla tem o seguinte significado: esquerda (L), direita (R), cima (U), baixo (D), frente (F) e traz (B).

Em *octrees* tradicionais, cada nó é representado geometricamente por um cubo. Um nó não-folha é subdividido no ponto médio p referente ao centro do nó (GLASSNER, 1984; SANDOR, 1985; PENG; ZHU; LIANG, 1987; SAMET, 1989; LEVOY, 1990; SPACKMAN; WILLIS, 1991; ARONOV; FORTUNE, 1999; REVELLES; URENA; LASTRA, 2000).

Figura 2.9 – Representação de uma octree de dois níveis.



Os oito octantes resultantes são geometricamente de mesmas dimensões. Essa abordagem tem a vantagem da facilidade de implementação. Além disso, também assume que os objetos estão distribuídos uniformemente pela cena. Caso os objetos estejam espalhados, dividir pela mediana dos objetos é mais eficiente para cálculo de intersecção. Assim, MacDonald e Booth (MACDONALD; BOOTH, 1990) demonstram que o ponto de subdivisão ótimo está entre o centro do nó e a mediana dos objetos. *Octrees* são bastante utilizadas em diversas situações para o cálculo do *ray tracing* pois a estrutura se adapta naturalmente com à geometria complexa da cena.

2.3 RAY TRACING HÍBRIDO

Renderização híbrida utiliza as técnicas de rasterização e *ray tracing* em conjunto para otimizar o cálculo da cor de cada pixel (BECK et al., 2005; HERTEL; HORMANN; WESTERMANN, 2009; CABELEIRA, 2010; SABINO et al., 2012). Essa estratégia é normalmente utilizada para renderização *offline*. Essas abordagens utilizam o *ray tracing* para calcular as características que a rasterização não consegue representar fielmente, como sombras detalhadas, transparência, reflexão, entre outros. A taxa de *frames* atingida nesses estudos impossibilita sua utilização em tempo real como em jogos, onde uma taxa de *frames* estável é necessária.

Trabalhos mais recentes adicionam heurísticas para agilizar o *ray tracing* (SABINO et al., 2012; SABINO; PAGLIOSA, 2012; ANDRADE; SABINO; CLUA, 2014; ANDRADE et al., 2014). Essas abordagens definem heurísticas que escolhem em tempo real os elementos que devem ser renderizados utilizando *ray tracing*. A seleção é baseada em quatro condições: o poder de processamento disponível, a posição atual e movimentos recentes da câmera, as propriedades de cada elemento e os elementos selecionados anteriormente. Assim, é possível reduzir o impacto do *ray tracing* na renderização de uma cena virtual ao selecionar precisamente quais objetos têm maior influencia na qualidade da imagem.

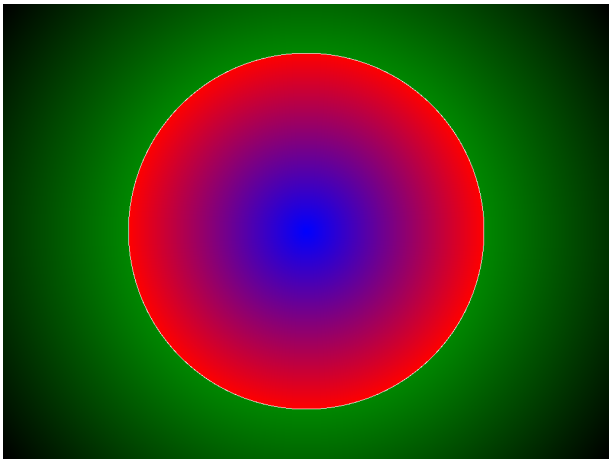
3 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo é apresentado uma revisão aprofundada das técnicas fundamentais e conceitos essenciais utilizados na formulação desta dissertação.

3.1 SIGNED DISTANCE FIELD

Os SDFs são bastante adequados na detecção de colisão e extensamente utilizados no campo da física. O SDF pré-processa um campo de distância ao redor de uma malha (Figura 3.1), o que permite a rápida e precisa consulta da distância entre dois objetos que possivelmente irão colidir. Além disso, o gradiente do SDF, que é definido pelo menor caminho até a superfície, pode ser usado como a normal de contato para reação à colisão. Bridson et al. (2005) e Fuhrmann et al. (2003) usam SDFs para calcular colisão entre tecido e corpo rígido. *Haptic rendering* (Renderização usando controles de realidade virtual), apresentado por Barbič e James (BARBIČ; JAMES, 2008), usam dados providos de uma representação SDF. Colisões com corpos rígidos são detectados pelos métodos de Kaufman et al. (2007), Glondu et al. (2012) e Xu e Barbič (XU; BARBIČ, 2014). Além disso, Xu et al. (2014) apresentam uma solução para detecção de colisão contínua para corpos rígidos. Neste trabalho, cria-se a representação da cena usando SDFs para agilizar o cálculo de colisão do *ray tracing*. Porém, a abordagem proposta também pode ser aplicada em outras áreas que se beneficiam do paradigma de SDF.

Figura 3.1 – Representação 2D de um *signed distance field*. A cor preta representa distâncias afastadas da superfície e verde distâncias próximas. A cor branca representa exatamente a superfície. As cores vermelha e azul representam a parte interna do círculo, proporcional a distância até a borda.



Desde a introdução de SDFs na computação gráfica por Rosenfeld e Pfaltz (ROSENFELD; PFALTZ, 1966), muitos métodos foram apresentados para melhorar a performance e a exatidão de *signed functions* baseados em malhas (por exemplo Sanchez et al. (2013)). Porém, o tempo de computação ainda é longo para simulação interativa ou *haptic rendering*. Com isso, aproximações do *signed distance function* usando um SDF pré-computado são geralmente suficientemente precisas. Devido ao fato de que a discretizações de cenas complexas consomem muita memória, vários métodos focados na redução do consumo de memória foram desenvolvidos. Um dos métodos mais utilizados atualmente é as *adaptively sampled distance fields* (ADFs) introduzida por Frisken et al. (2000). ADFs constroem um SDF em formato de octree. Durante sua construção, uma célula pai da octree é subdividida em células filhas enquanto a diferença da distância aproximada da célula pai para a célula filha esteja acima de um determinado limite. Segue-se que os ADFs necessitam de muitas células filho em regiões onde a discretização trilinear não representa adequadamente a distância até a superfície. Tanto o tempo de construção quanto o consumo de memória das ADFs foram aperfeiçoados por Perry e Frisken (PERRY; FRISKEN, 2001). Recentemente, Liu e Kim (LIU; KIM, 2014) apresentaram um método para computar ADFs na GPU. Outra abordagem que reduz o consumo de memória é discretizar somente uma faixa estreita perto da superfície do objeto como proposto por Bærentzen (BÆRENTZEN, 2002) e Erleben e Dohlmann (ERLEBEN; DOHLMANN, 2008).

Uma abordagem híbrida para malhas entre avaliação exata e pré-computada do SDF é apresentado por Huang et al. (2001). Essa abordagem trabalha com uma *grid* regular onde cada célula armazena dados sobre todos os triângulos que influenciam os valores de distância da célula. Como resultado, valores exatos de distância torna-se disponíveis para todas as células. Enquanto esse método desacopla com sucesso o tamanho da *grid* com sua acurácia, torna-se difícil encontrar a proporção correta entre o tamanho da *grid* e o número de triângulos por célula. Outras abordagens adicionam à *grid* dados adicionais que representam características agudas como quinas ou bordas sem subdividir a *grid* desnecessariamente. Por exemplo, Ju et al. (2002) armazena as próprias funções e suas derivadas na *grid*, Qu et al. (2004) armazena uma *grid* curvilinear adicional e Bærentzen (BÆRENTZEN, 2005) usa um ponto adicional além da *grid*. Um eficiente esquema de cache de subdivisão espacial para dados volumétricos na *grid* é proposto por Museth (MUSETH, 2013). Essa abordagem é específica para um grande conjunto de dados esparsos com um domínio de pelo menos 81923 células. Para ser capaz de manipular um grande conjunto de dados essa abordagem usa uma estrutura similar a uma árvore B+ para encontrar células com dados válidos. Em vez de subdividir o espaço em hexaedros, Wu e Kobbelt (WU; KOBBELT, 2003) propõem usar uma partição binária de espaço (BSP-tree) onde o campo de distância dentro da célula é aproximado por uma função linear. A principal vantagem desse método comparado com *grid* é o ajuste de planos divididos para a geometria. Em contraste com esquemas de subdivisão espacial, Jones (JONES,

2004) transforma o campo de distância com uma transformação vetorial de distância e um preditor definido que é capaz de usar compressão de entropia no campo. Por reduzir os dados SDF para um campo de altura 2D projetado em uma geometria intermediária, a proposta de Otaduy et al. (2004) e Moustakas et al. (2007) visa reduzir o consumo de memória utilizado pelo SDF. O principal problema dessas abordagens é definir uma geometria intermediária apropriada.

Mitchell et al. (2015) apresentam um SDF que suporta mais de um valor onde múltiplas células podem ocupar o mesmo volume espacial. Isso permite representar características *non-manifold* que não podem ser representadas pelas representações padrões baseado em *grid*. Como tal, a abordagem é ortogonal aos métodos mencionados acima para representar mais detalhes com menos memória. Volumes de contato é uma abordagem baseada em captura de imagens proposta por Faure et al. (2008) e Allard et al. (2010). É uma alternativa que captura a geometria de contato detalhada do objeto. Para isso, é necessário uma amostragem em alta resolução para manipulação de contato precisa.

3.2 DIAGRAMA DE VORONOI

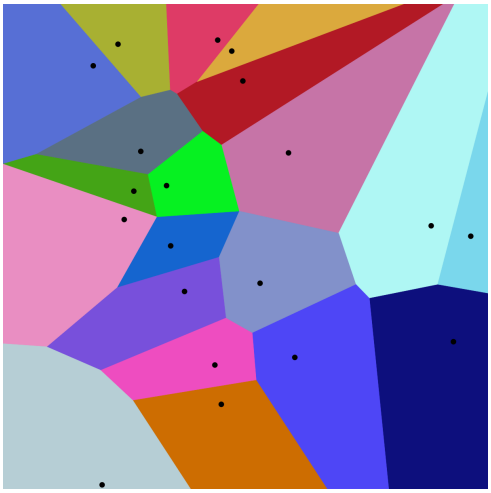
O Diagrama de Voronoi visa decompor um espaço de N dimensões e é gerado baseado na distância euclidiana entre os elementos de um determinado conjunto de objetos gerando células contendo somente um objeto, como visto na Figura 3.2. Essas células, também chamados de sítios ou geradores, representam um conjunto de pontos no dado espaço o qual a distância de quaisquer pontos dentro do sítio é menor para o objeto em seu interior comparado a quaisquer outros. Diagramas de Voronoi podem ser encontrados em diversos campos da ciência e tecnologia, até mesmo na arte, tendo inúmeras aplicações práticas e teóricas.

Como a estrutura proposta visa mapear objetos mais próximos a dado ponto no espaço dentro de um nó, o Diagrama de Voronoi representa exatamente a região de influência de cada objeto possibilitando o mapeamento preciso. Este trabalho utiliza uma aproximação do diagrama com mesma propriedade para gerar a estrutura proposta.

3.3 RENDERIZAÇÃO

Uma cena que usa *distance fields* pode ser renderizada usando algoritmos simples. A cor final de cada pixel é calculada ao encontrar a intersecção dos raios lançados a partir da câmera com superfícies da cena. Os pontos resultantes podem ser usados para aproximar a normal da superfície e a distância da câmera, que são essenciais para cálculo

Figura 3.2 – Representação 2D de um diagrama de Voronoi de pontos infinitesimais.



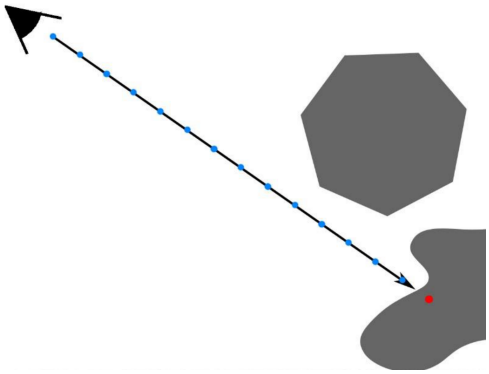
de iluminação. Algoritmos otimizados para o cálculo de intersecção com superfícies têm sido vastamente pesquisados nos últimos anos. As duas técnicas mais usadas atualmente são *ray marching* e *sphere tracing* e são apresentadas neste capítulo. Essas abordagens mudam completamente o paradigma padrão de representação de malhas trigonométricas utilizado nas últimas décadas e são intimamente ligadas com o algoritmo *ray tracing*.

3.3.1 Algoritmo Ray Marching

A técnica *ray marching* é similar ao *ray tracing* com algumas modificações adicionais para melhorar a performance de colisão do raio lançado com superfícies na cena. Assim, essa técnica visa subdividir um raio em N passos modificando-os de maneira interativa à medida que é amostrado (Figura 3.3). Normalmente, essa técnica é utilizada quando a superfície é bem refinada ou de alta complexidade facilitando operar sobre a mesma. As abordagens, por exemplo, do *sphere tracing* (Sessão 3.3.2) e *path tracing* se baseiam nessa estratégia.

Normalmente, essa técnica utiliza somente objetos cujas geometrias são superfícies intrínsecas ou cenas simplificadas para demonstração de efeitos de iluminação. Neste trabalho, cada passo do *ray marching* é modificado usando o Scene SDF para o cálculo do tamanho do mesmo permitindo cenas de maior complexidade e quantidade de objetos enquanto mantém a qualidade gráfica. A informação da superfície do objeto pode ser utilizada para calcular outras características da superfície como vetor normal. O Algoritmo 3.1 apresenta o pseudocódigo básico do *ray marching*. A função $SDF()$ representa o cálculo de distância. As letras O e D representam a origem e a direção do raio respectivamente e a letra s representa um tamanho fixo e predeterminado do passo.

Figura 3.3 – Representação do *ray marching* para cálculo de intersecção.



Algoritmo 3.1: Pseudocódigo do algoritmo ray marching.

```
For each pixel:
   samplingPoint = O
      while(SDF(samplingPoint) > 0)
         samplingPoint += s * D
```

Em implementações práticas, mais condições são adicionadas ao laço principal. Um exemplo é manter um registro da distância percorrida e acrescentar um critério de parada caso a distância seja maior que um valor pré-determinado. Outra adição comum é determinar um número máximo de passos, garantindo que o laço termine caso o raio não colida com nenhum objeto. Esse algoritmo pode ser facilmente paralelizado pela GPU, uma vez que os cálculos devem ser feitos para cada pixel independentemente.

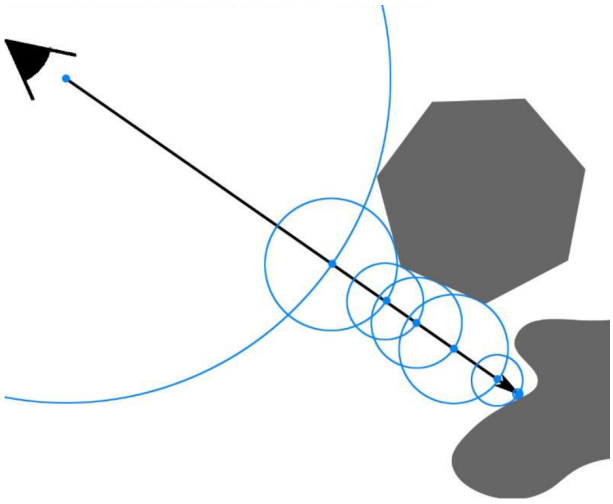
O algoritmo convencional do *ray marching* tem múltiplas desvantagens. A principal é o tamanho fixo do passo, onde raios podem atravessar objetos com dimensões pequenas caso o tamanho do passo seja muito grande. Muitas variações do algoritmo foram propostas para resolver esse problema. Por exemplo determinar se o ponto avaliado está no interior de um objeto e caso positivo retornar a distância com sinal negativa. Com o sinal invertido, o passo percorre o caminho inverso. Além disso, o tamanho do passo é reduzido possibilitando encontrar a superfície. Esse processo pode ser repetido um número arbitrário de vezes para aproximar cada vez mais do ponto de intersecção.

Essas adições não resolvem problemas onde há a necessidade de amostrar uma grande quantidade de objetos de maneira precisa. Com isso o algoritmo de *sphere tracing* foi desenvolvido como uma extensão do *ray marching* para solucionar essas dificuldades usando as propriedades nativas dos *distance fields*. *Ray marching* ainda é o estado-da-arte para cálculo de intersecção, como por exemplo calcular a superfície em terrenos procedurais baseados em *heightmap*.

3.3.2 Algoritmo Sphere Tracing

Como mencionado anteriormente, a principal desvantagem do algoritmo *ray marching* é o tamanho fixo de cada passo. Esse problema foi solucionado pelo *sphere tracing* proposto por Hart (HART, 1996) onde o tamanho do passo é ajustado dinamicamente ao longo do caminho. O tamanho de cada passo é dado pela distância mais próxima de quaisquer superfície. Atribuindo esse valor para o tamanho do passo do raio fará com que o mesmo nunca avance para o interior de um objeto. Além disso, grandes espaços vazios são analisados rapidamente. O pseudocódigo apresentado na Figura 3.4 exibe o algoritmo básico do *sphere tracing*.

Figura 3.4 – Representação do *sphere tracing* para cálculo intersecção.



```
For each pixel:
    samplingPoint = O
    step = SDF(samplingPoint)
    while(step > 0)
        step = SDF(samplingPoint)
        samplingPoint += step * D
```

Da mesma maneira que a correção do tamanho do raio impede que o raio avance mais do que necessário, também evita que objetos pequenos sejam ignorados, uma vez que o tamanho do passo se ajusta de acordo com a distância da superfície. O algoritmo tem esse nome pois a distância calculada pode ser visualizada como o raio de uma esfera com origem na posição do passo. A principal desvantagem dessa técnica está no fato que o número de passos não pode ser pré-determinado pois é altamente dependente da origem e direção do raio. Caso um raio cruze paralelamente a um objeto, o tamanho do passo será reduzido significativamente podendo impactar negativamente na performance. Assim, um número máximo de passos e uma distância mínima do passo devem ser definidos.

Em muitos casos, o ponto calculado pelo *sphere tracing* é somente uma estimativa do ponto matematicamente correto de intersecção, que pode ser obtido analisando a geometria daquela região. O tamanho do passo é dependente do valor calculado na função de distância na posição atual. Isso resulta em um diferente número de iterações para intersectar objetos, e pode se tornar problemático quando um raio passa perto de um objeto, sem intersectá-lo. Como definido pelo algoritmo, o tamanho do passo se torna pequeno, aumentando o número de passos necessários. Assim, alguns artefatos podem

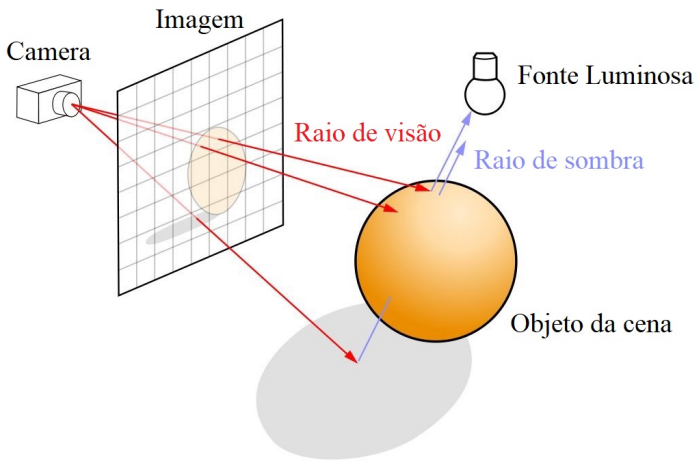
surgir caso o número exceda o valor previamente estipulado, e podem ser visíveis especialmente quando a câmera está olhando paralelamente a uma superfície. Esses artefatos podem ser reduzidos ao incrementar o tamanho mínimo do passo, porém com impacto na performance.

3.3.3 Algoritmo Ray Tracing

O algoritmo original do *ray tracing* é recursivo e consiste em projetar, à partir do observador, raios para cada um dos pixeis da câmera. Esses raios irão ou não intersectar com objetos na cena para formar a imagem em análise (Figura 3.5). Se um determinado raio não intersectar nenhum objeto no seu trajeto, é atribuído ao pixel a cor do fundo da cena. Caso o raio intersectar um objeto, é necessário determinar a cor do pixel correspondente. Para isso, calcula-se a iluminação no ponto de acordo com a cena. Essa região pode ser iluminada diretamente por fontes luminosas, por reflexão de outro objeto, por luz refratada transmitida através de outro objeto ou pode ser uma combinação de mais do que uma destas formas de iluminação, que é a situação mais comum.

Para determinar a intensidade da luz que atinge o ponto em análise o algoritmo recorre a três tipos de raios secundários: raios refletidos, refratados e de sombra. Cada um destes raios possui características e objetivos diferentes que podem facilmente ser identificados pelo seu nome. A partir da intersecção destes raios secundários com objetos e de acordo com a informação que cada um deles transporta, o algoritmo calcula a influência da luz e qual sua contribuição para a área sendo avaliado.

Figura 3.5 – Representação do processo dos raios lançados do algoritmo *ray tracing*.



Com isso, um único raio pode ser refletido e refratado indefinidamente, subdividindo-se em vários raios secundários, formando assim uma árvore complexa de raios de acordo com a composição da cena. Para possibilitar o processamento destas informações, é ne-

cessário estabelecer limites máximo de subdivisões. Um dos métodos mais utilizados para terminar a recursão dos raios secundários acontece quando a contribuição retornada pelos raios secundários para um determinado pixel se torna inferior a um determinado valor α previamente definido ou quando um raio primário não intersecta nenhum objeto dentro dos limites da cena, assim, atribuindo a cor de fundo ao pixel em análise.

As sombras são verificadas através de raios secundários que são lançados a partir do ponto de intersecção do raio primário com o objeto, em direção a fonte luminosa. Se no seu trajeto o raio intersectar outro objeto antes de chegar a fonte luminosa, é porque o ponto em análise se encontra na sombra, se não, é porque recebe luz direta.

3.3.4 Gerenciamento de Material

Informações sobre a superfície do objeto são necessárias para mapear uma textura específica ou parâmetros de luz para um determinado pixel na renderização usando SDF. Para isso, é necessário recuperar informações específicas do material do objeto que intersectou com o raio. Isso é feito armazenando o id do último objeto cuja distância foi suficiente para que a intersecção ocorra. Esse id representa um índice na estrutura de informações arbitrárias dos objetos, como texturas, uvs (caso malha genérica), informações sobre superfície, entre outros.

Como o resultado da cor do pixel é calculado em uma única passada, utilizar diferentes funções de iluminação para diferentes materiais podem causar impacto na performance. Calcular informações de diferentes materiais é feito à partir do id, porém a função que calcula da cor do pixel não pode mudar de forma dinâmica de acordo com o material selecionado. Isso ocorre porque as linguagens para GPU (como *glsl* e *hlsl*) são básicas e não suportam ponteiros ou polimorfismo. Para solucionar esse problema, pode ser utilizado um *switch*, chamando as funções de iluminação respectivas do material, com impacto na performance.

3.3.5 Mapeamento de Textura

Mapeamento de textura têm sido usado por mais de 20 anos para aumentar a fidelidade visual dos triângulos renderizados. Esse processo inicia mapeando uma textura 2D em uma malha 3D, onde é atribuído para cada vértice da malha uma coordenada de textura (coordenada uv). Durante a renderização de cada *frame*, o valor uv dos triângulos sendo analisados são interpolados para cada fragmento no *fragment shader*, possibilitando a renderização de objetos texturizados.

O mapeamento de textura uv não pode ser utilizado usando a técnica SDF para

representar um objeto, uma vez que o SDF somente contém informação de distância e não de posição de vértice impossibilitando mapeamentos e interpolações. Assim, a solução de mapeamento uv para geometrias genéricas varia de acordo com a abordagem escolhida (*ray marching*, *sphere tracing* e *path tracing*) e com a abordagem utilizada para gerar o SDF da malha genérica. Para geometrias intrínsecas, é possível criar coordenadas uv para objetos utilizando sua fórmula matemática, por exemplo o Algoritmo 3.2 exemplifica o mapeamento uv de uma esfera.

Algoritmo 3.2: Mapeamento uv de uma esfera.

```
u = atan2(normal.x, normal.z) / (2 * pi) + 0.5;
v = normal.y * 0.5 + 0.5;
```

3.3.6 Normal da Superfície

A normal da superfície é essencial para os algoritmos de iluminação. No modelo convencional de renderização de malhas, a normal de cada vértice é pré-processada ou calculada pelo produto vetorial entre os vértices de um triângulo. Porém, essa abordagem é inviável ao utilizar renderização por *distance fields*, uma vez que normalmente não há informação alguma sobre a geometria. Com isso, é necessário aproximar a normal da superfície usando o gradiente da superfície (HART, 1996) como visto no Algoritmo 3.3.

Algoritmo 3.3: Cálculo de normal à partir do gradiente da superfície.

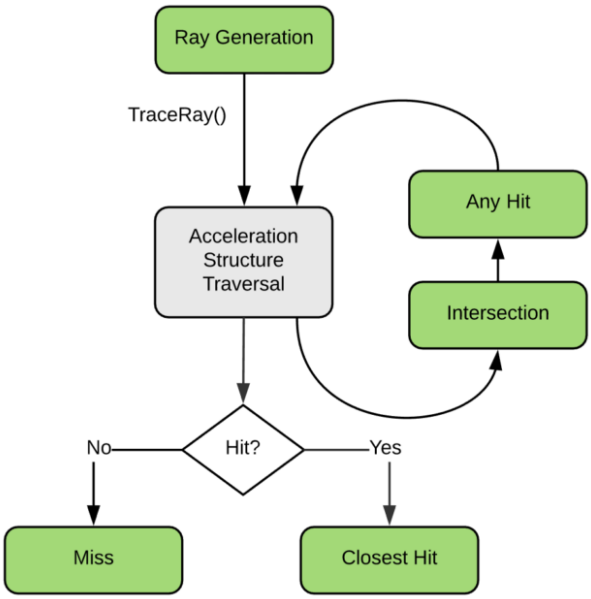
```
Nx = SDF(pos.x − ε, pos.y, pos.z) − SDF(pos.x + ε, pos.y, pos.z)
Ny = SDF(pos.x, pos.y − ε, pos.z) − SDF(pos.x, pos.y + ε, pos.z)
Nz = SDF(pos.x, pos.y, pos.z − ε) − SDF(pos.x, pos.y, pos.z + ε)
```

Para aproximar a normal é necessário calcular a distância da superfície pelo menos seis vezes, duas vezes para cada eixo. Pode-se utilizar mais amostras para aumentar a precisão, porém com um custo de performance proporcional a complexidade de calcular esse valor. Para fins visuais, seis amostras mostram-se suficientes enquanto mantém o baixo custo computacional. A constante ε determina o *offset* do ponto sendo avaliado e, incrementando esse valor resulta em uma suavização nas normais em objetos rugosos.

3.3.7 NVIDIA RTX Ray Tracing

A NVIDIA recentemente lançou da API NVIDIA RTX. É uma tecnologia de alta performance que é alimentada pelas GPUs da NVIDIA de arquitetura VOLTA e suas sucessoras. Além disso, a Microsoft anunciou integração do *ray tracing* diretamente na API do DirectX. A estrutura base da RTX contém os *shaders* dos objetos que serão executados no *dispatch* do *ray tracing*, estruturas de aceleração para acelerar o *ray tracing* e tabelas que definem as relações entre a geometria dos objetos e suas propriedades (texturas, constantes, etc). Além disso, é introduzido um novo pipeline (Figura 3.6).

Figura 3.6 – Representação do pipeline gráfico da API NVIDIA RTX.



O primeiro passo é gerar o raio, procurar por intersecções e chamar o *shader* de *Miss* ou *Closest Hit*. Assim, os testes de colisão usam a estrutura de aceleração e avaliam quando o raio colide com o *bouding volumes* de um objeto e, caso positivo, um *shader* de intersecção determina se intersectou com a geometria. Caso negativo, continua com o percurso do raio até finalizar a recursão. Ao terminar sua computação, o raio tem como resposta ou *Miss* ou *Closest Hit*, chamando o respectivo *shader*. O *Closest Hit* é onde avalia-se o material, texturas, constantes, etc.

A API é construída de maneira que para executar cada estágio basta chamar algumas funções respectivas de cada um. Semanticamente, cada *thread* da GPU é responsável por um raio por vez e não podem se comunicar durante o processo. Pela falta de suporte da API com as ferramentas utilizadas para a implementação deste trabalho - além de não ter a disponibilidade de hardware compatível com a tecnologia, não foi possível comparar sua performance com a solução proposta.

4 ARQUITETURA PROPOSTA

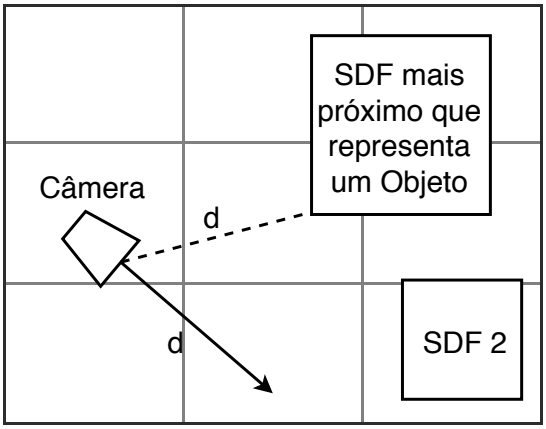
4.1 VISÃO GERAL

A estrutura de dados proposta nesta dissertação visa minimizar o impacto do cálculo de intersecção dos raios do algoritmo *ray tracing* em GPU com objetos da cena utilizando uma estrutura de subdivisão espacial pré-processada em CPU baseada nas estruturas de *signed distance fields*, *hash* espacial e *octree*. A subdivisão inicia com uma repartição elementar do espaço tridimensional (*hash* espacial) em nós para separar objetos relevantes dentro de um determinado espaço. Após isso, é subdividido cada nó do *hash* em uma *octree* que refina mais detalhadamente os objetos de um nó. As malhas dos objetos são completamente substituídos pelas suas representações usando SDF.

A estrutura de dados do *Scene* SDF é pré-processada e armazenada em disco, agilizando futuras execuções. O processo de geração da estrutura inicia pelo usuário selecionando a região de interesse, a qual define quais objetos serão incluídos no cálculo, e o tamanho do nó do *hash*. Em seguida, os objetos selecionados são processados em uma estrutura aceleradora para a criação do *Scene* SDF (Sessão 4.2). Após isso, utilizando a estrutura gerada no passo anterior, executa-se um processo de três etapas onde é gerada as estruturas de *hash*, das *octrees* e dos objetos (Sessão 4.3) até que a região selecionada seja inteiramente processada, ou seja, o processo de mapeamento da cena esteja concluído. A escolha da região de interesse e o tamanho do nó do *hash* estão estritamente relacionadas com o desempenho do algoritmo de *ray tracing* e devem ser atribuídos especificamente para cada configuração de cena. Como resultado, cada nó do *Scene* SDF referencia todos os objetos que influenciam o espaço do mesmo enquanto a estrutura distribui todos os objetos de forma igualitária impedindo a concentração em um único nó da *octree*/*hash*.

O algoritmo de *ray tracing* proposto neste trabalho é baseado na técnica de *sphere tracing*, ou seja, cada raio é subdividido em N passos (Sessão 3.3.2). O tamanho de cada passo é calculado com o auxílio da estrutura *Scene* SDF enviada à GPU. Para realizar o cálculo de distância (Figura 4.1) é necessário somente uma posição no espaço referente ao passo do raio. A imagem final é gerada assim que a análise de todos os raios é finalizada (Sessão 5.3).

Figura 4.1 – Ilustração do cálculo de distância d entre um ponto no mundo ao objeto mais próximo. A *grid* representa o nível mais baixo da *octree* em um nó do *hash*.



4.2 ESTRUTURA ACELERADORA

A estrutura aceleradora organiza e calcula informações dos objetos que visam maximizar a precisão e o desempenho da criação do *Scene* SDF. Essa estrutura calcula a área de influência do objeto, ou seja, o intervalo no espaço tridimensional mais promissor para o cálculo da cor do pixel sendo avaliado relativo ao objeto. Esse espaço tridimensional é uma malha tridimensional, a qual define estritamente essa propriedade e que pode ser gerada utilizando um Diagrama de Voronoi 3D com todos os objetos como entrada, porém com um alto custo computacional. Por isso, foi utilizado uma aproximação simplificada que visa calcular o valor máximo de todos os eixos positivos e negativos da malha criada à partir do Diagrama de Voronoi e utilizando esses valores é gerado uma AABB que representa influência do objeto.

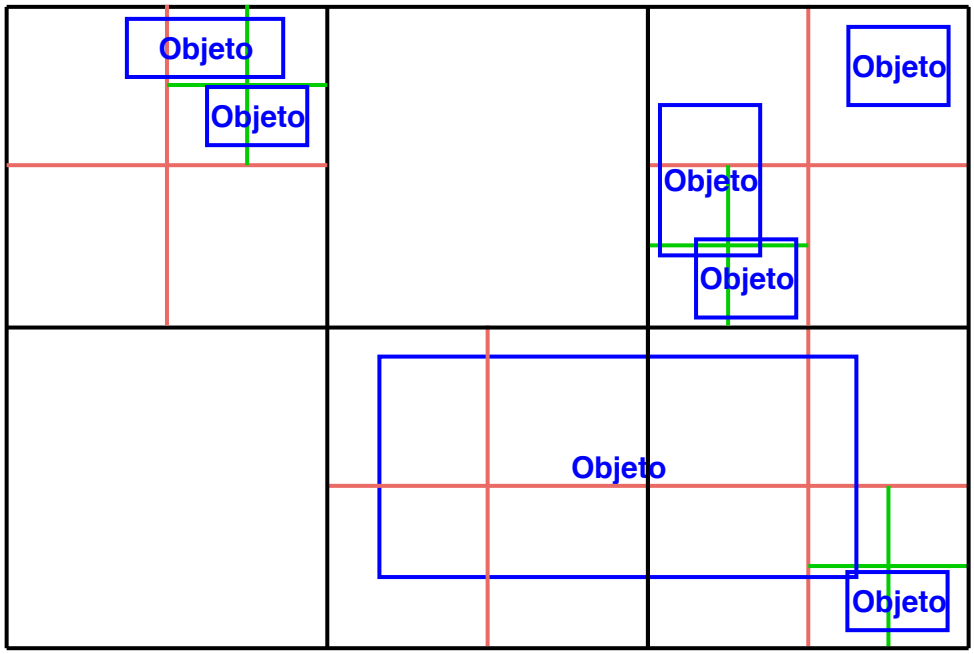
Para calcular os dimensões da AABB leva-se em conta os objetos no entorno do objeto analisado. Primeiramente, faz-se uma análise em todos os eixos positivos e negativos verificando os objetos mais próximos naquela direção. Após isso, é possível iterar pelas arestas entre esses objetos e, assim, aproximar para o vértice de maior distância do centro. Posteriormente, essa estrutura irá auxiliar na distribuição dos objetos pelas *octrees*.

4.3 HASH, OCTREE E OBJETOS

A hierarquia de estruturas que armazena os objetos inicia com o uso de um *hash* tridimensional gerado a partir da cena onde cada célula armazena uma *octree* (Figura 4.2). Como visto no Capítulo 2.1.2, o *hash* espacial tem performance superior comparado a *octree*, porém a estrutura não se adéqua a geometria da cena. Com isso, cenas com grandes espaços vazios utilizam uma maior quantidade de recursos, impossibilitando representá-la

eficientemente. A combinação das estruturas *hash* e *octree* é necessária pois melhora a capacidade de representação de cenas ao introduzir a *octree* que se adéqua a geometria mais facilmente. O *hash* possibilita reduzir o laço de descida na *octree*, melhorando o desempenho da busca pela geometria mais próxima. O *hash* é armazenado em um *array* de inteiros de uma dimensão onde cada valor do *array* referencia a raiz de uma *octree*.

Figura 4.2 – Representação 2D da estrutura. A *grid* preta representa o *hash*. As *grids* vermelha e verde representam o primeiro nível e o segundo nível da *octree* respectivamente. Em azul, estão representado os objetos.



Cada *octree* recebe as estruturas aceleradores geradas anteriormente à partir dos objetos para iniciar sua criação. A estrutura aceleradora simplifica a geometria dos objetos para uma AABB que representa área de influência do mesmo. Assim, a *octree* utiliza dessa informação para se subdividir de acordo com a quantidade de AABBs presentes em seu volume, até um número mínimo de objetos O e nível máximo N. A *octree* é armazenada em um *array* de uma dimensão da estrutura $OctreeBufferStruct$ (Algoritmo 4.1) onde cada elemento do *array* representa um nó da *octree*. O conteúdo dessa estrutura contém informações do nó, índices referentes aos objetos do nó e índices para sub-nós. Após a indexação dos objetos na estrutura Scene SDF, a estrutura aceleradora é destruída.

Algoritmo 4.1: Estrutura da Octree em C#.

```
public struct OctreeBufferStruct
{
   public int index;
   public int indexOfvalues;
   public int countOfvalues;
   public fixed int childs[8];
}
```

O valor $index$ é o índice da própria *octree* no nível superior, onde o valor 0 representa o canto inferior esquerdo do volume e o valor 7 o canto superior direito do volume. Esse valor é usado para calcular em tempo de execução qual o centro da *octree*. O valor $indexOfValues$ é o índice no *array* de objetos do primeiro objeto da *octree*. O valor $countOfValues$ representa a quantidade de valores disponíveis no *array* de objetos. O valor $childs[8]$ é o índice para as *octrees* no próximo nível.

Caso todos os objetos na cena sejam representados por uma ou mais geometrias implícitas, esses podem ser armazenados em um *array* de *float3* que representa as suas respectivas posições no mundo e um *array* auxiliar de inteiros que faz a conexão entre a *octree* e o *array* de posições. Esse *array* auxiliar é necessário uma vez que elimina a necessidade de duplicar os valores de posição que compartilham mais de um nó, assim, economizando memória ao se duplicar inteiros ao invés de *float3*. Caso os objetos sejam representados por uma estrutura de SDF gerada a partir de uma geometria genérica, o último nível da *octree* deve indexar a estrutura criada a partir dos objetos naquele espaço.

5 IMPLEMENTAÇÃO

5.1 GERAÇÃO DO SDF

A estrutura proposta nesta dissertação busca encontrar o SDF do objeto mais próximo de um ponto no espaço avaliado minimizando cálculos de colisão enquanto o SDF calcula a distância desse ponto avaliado com a superfície do mesmo. Assim, é necessário uma representação precisa da superfície para que o cálculo de distância condiga com as necessidades do cálculo de colisão. A geração do SDF de uma geometria implícita (esfera, cubo, cápsula, pirâmide, etc.) pode ser calculada utilizando sua fórmula matemática. Por exemplo, o cálculo do SDF da esfera e do cubo podem visualizados nos Algoritmos 5.1 e 5.2 respectivamente.

Algoritmo 5.1: SDF de uma esfera.

```
float SphereSDF(float3 spherePos, float3 currentPos, float radius)
{
    return length(spherePos - currentPos) - radius;
}
```

Algoritmo 5.2: SDF de um cubo.

```
float BoxSDF(float3 boxMin, float3 boxMax, float3 position)
{
    float dx = max(max(boxMin.x - p.x, 0), position.x - boxMax.x);
    float dy = max(max(boxMin.y - p.y, 0), position.y - boxMax.y);
    float dz = max(max(boxMin.z - p.z, 0), position.z - boxMax.z);

    return sqrt(dx * dx + dy * dy + dz * dz);
}
```

Analisando o cálculo do SDF de geometrias implícitas, é possível calcular a distância até a superfície do objeto apenas utilizando sua posição. As primitivas são estruturas essenciais para SDFs, uma vez que é possível combinar múltiplos SDF para representar apenas um objeto. Para isso, são utilizadas operações Booleanas como união (Algoritmo 5.3), subtração e intersecção. Essas operações são úteis uma vez que preservam a continuidade de Lipschitz, ou seja, preservando corretamente os gradientes e as distâncias dentro do domínio.

Algoritmo 5.3: Algoritmo de união de dois SDF.

```
float union( float distance1, float distance2 )
{
   return min(distance1,distance2);
}
```

Os SDFs de geometrias genéricas necessitam representar a malha com alta precisão. Para isso, geralmente baseiam-se em estruturas de subdivisão espacial como *hash* espaciais (MUSETH, 2013), *octrees* (FRISKEN; PERRY, 2006) ou árvores BVH (GUNTHER et al., 2007). Neste trabalho, o SDF de malhas genéricas é pré-processado e armazenado em disco de acordo com o Algoritmo 5.4. O espaço ao redor do objeto é discretizado em nós ou *voxels* utilizando *hash* espacial onde cada nó armazena os triângulos que são relevantes para o cálculo de distância naquela região. A geração do *hash* é realizados em dois passos para determinar se um triângulo deve ser armazenado em um nó. O primeiro passo é calcular todos os triângulos que estão contidos no nó de dimensões $N \times M \times L$ e, caso positivo, também calcular todos os triângulos que estão à uma distância d da célula onde $d < \frac{max(N,M,L)}{2}$. Os triângulos dentro dessa distância podem estar mais próximos de regiões dentro do *voxel* do que triângulos em seu interior. O segundo passo é analisar todos os nós vazios (que não possuem triângulos em seu interior) e determinar o triângulo que tenha a menor distância do nó. Utilizando essa abordagem, as informações de vértice, UV, normais e triângulos da geometria também são armazenadas. Com isso, é possível calcular informações sobre a superfície, evitando a necessidade de aproximações dessas informações, como por exemplo a normal.

Algoritmo 5.4: Pseudo-código de geração do SDF de uma malha genérica.

```
For each voxel v:
   For each triangle t:
      if (Intersects(v,t))
         Triangles.Add(t)
         continue
      d = Distance(v, t)
      if (d < max(M,N,L) / 2)
         CloseToNode.Add(t)
      if (d < minDist)
         minDist = d
         closestTriangle = t
   if (IntersectNode.Count == 0)
      Triangles.Add(closestTriangle)
   else
      Triangles.Add(CloseToNode)
```

5.2 ENVIO DE INFORMAÇÕES PARA GPU

Para enviar as estruturas calculadas para a GPU, é necessário gerar *buffers*, preenchê-los, referenciá-los na GPU para finalmente enviar. As estruturas dos *buffers* têm que ser similar a estrutura dos dados, por exemplo caso tenha três variáveis do tipo inteiro e duas do tipo *float*, a estrutura do *buffer* também deve ter o mesmo número e tipos de variáveis além de estarem dispostos na mesma ordem. Além das estruturas calculadas, informações como resolução da tela, posição da luz e da câmera também são necessárias para o cálculo de renderização. Essas também são enviadas através de *buffers* dinâmicos. Como as estruturas pré-calculadas foram geradas com intuito de serem enviadas para a GPU, basta criar uma estrutura similar em HLSL, preencher os *buffers* com a estrutura calculada e enviar para GPU. A Tabela 5.1 exibe um exemplo da representação da estrutura da *octree* em GPU. As estruturas que armazenam os objetos são enviadas apenas uma vez para a GPU logo após a sua criação, uma vez que são estruturas estáticas. Em contrapartida, informações dinâmicas, como posição da câmera, são enviadas uma vez a cada frame.

A abordagem proposta consiste de dez *buffers*. A estrutura *Scene* SDF utiliza os quatro primeiros e os demais são referentes ao SDF das malhas genéricas dos objetos da cena. O primeiro é referente ao *hash* espacial que subdivide toda a cena em nós ou *voxels*. O segundo é o *buffer* referente às *octrees*, onde há uma para cada nó do *hash* espacial. O terceiro indexa os objetos em cada nó folha da *octree* no *buffer* objetos. O quarto *buffer* representa os objetos, onde cada valor identifica um objeto diferente. O quinto representa o *hash* dos objetos, onde o espaço ao redor do objeto é subdividido em nós ou *voxels* que identifica os triângulos mais próximos naquele espaço no *array* de triângulos. O sexto *buffer* representa o *array* de triângulos que referenciam os vértices no *array* de vértices, onde diferentes malhas estão concatenadas no mesmo *array*. O oitavo, nono e décimo *buffer* representam os vértices, normais e uvs das malhas genéricas dos objetos respectivamente.

Tabela 5.1 – Comparação da implementação entre a estrutura da *octree* em C# e HLSL.

```
//C#
public struct OctreeBufferStruct
{
    public int index;
    public int indexOfvalues;
    public int countOfvalues;
    public fixed int childs[8];
}
```

```
//HLSL
struct OctreeBufferStruct
{
    int index;
    int indexOfValues;
    int countOfValues;
    int childs[8];
};
```

5.3 RECURSÃO BASE

O algoritmo de renderização é programado usando HLSL para *Compute Shader*. O cálculo é paralelizado de acordo com a quantidade de pixeis da resolução, ou seja, todos os pixeis são calculados paralelamente. O cálculo de cada pixel baseia-se na contribuição do raio lançado para a imagem final e na quantidade de raios calculados, de acordo com o Algoritmo 5.5. A função $Trace$ calcula a interação do raio com a cena, podendo ou não refletir de acordo com as propriedades do material do objeto. Caso o raio intersecte um material reflexivo, a função $Trace$ calcula posição de contato, reflete o raio, reduz a intensidade do raio e retorna a cor referente ao objeto. A intensidade do raio é representada pela variável $rayEnergy$, que representa o quanto aquele raio contribui para a cor final.

Algoritmo 5.5: Recursão base do processo de geração de imagem usando princípios do algoritmo sphere tracing.

```
int count = 0
while(length(rayEnergy) < ℰ && count < N)
   result += rayEnergy * TraceRay(rayPos, rayEnergy)
   count++
```

A função $Trace$ calcula a trajetória de um raio baseado no algoritmo *sphere tracing* descrita no Capítulo 3. De forma análoga, o tamanho do raio é ajustado dinamicamente ao longo do caminho e seu tamanho é igual a distância da superfície mais próxima. Assim, a estrutura Scene SDF auxilia o cálculo da superfície mais próxima ao subdividir a cena e as malhas dos objetos em estruturas que facilitam o cálculo de intersecção raio-superfície. O algoritmo é similar ao da Figura 3.4, onde a função SDF é substituída pela função de intersecção de superfície descrita nos Capítulos subsequentes.

5.4 INTERSECÇÃO COM SUPERFÍCIES

O cálculo de intersecção com superfícies inicia ao calcular a intersecção do raio origem com a AABB do *hash* da cena. Assim, com o passo do raio posicionado dentro do *hash*, inicia-se a busca na estrutura *Scene* SDF calculando o nó do *hash* baseando-se na posição qual se encontra o passo, de acordo com o Algoritmo 5.6 escrito em HLSL.

Algoritmo 5.6: Cálculo da célula do hash à partir de uma posição no mundo.

```
index = clampToInt(position − gridMin, 0, gridSize);
```

O dado armazenado no *array* do *hash* é um inteiro referente ao índice da *octree* possibilitando, assim, analisar a *octree* e recuperar os índices dos objetos naquela região. Para isso, um laço avalia a *octree* até que o índice dos objetos seja válido, como representado no Algoritmo 5.7. É necessário ressaltar que todos os níveis mais inferiores da *octree* têm pelo menos um objeto mapeado.

Algoritmo 5.7: Busca na octree, descendo até o nível mais inferior.

```
while(index > -1)
    node = octreeBuffer[index]
    child_index = FindIndex(rayPos)
    index = node.childs[child_index]
```

O resultado da busca são dois inteiros: um representando o índice inicial dos SDFs do nó e o segundo a quantidade de SDFs, possibilitando assim a iteração entre os mesmos. Após isso, basta avaliar individualmente as estruturas de cada SDFs. A abordagem proposta implementa o SDF das malhas genéricas dos objetos utilizando a estrutura *hash*, onde cada célula armazena os possíveis triângulos mais próximos do ponto sendo analisado. Cada valor retornado pela busca da *octree* indexa uma estrutura que contém informações do objeto como indexação da malha, informações do *hash* do SDF, matrizes de transformações, entre outros. Com essas informações, podemos utilizar o Algoritmo 5.6 para encontrar o nó do *hash* do SDF na qual o ponto avaliado se encontra. Como cada nó contém informações dos triângulos referente aquela região, é possível calcular as intersecção com os triângulos indexados e encontrar a distância mais próxima da superfície. Após isso, basta continuar os passos da recursão calculando as contribuições dos raios até as condições de paradas serem satisfeitas. Com isso, é possível encontrar todas as superfícies mapeadas na estrutura criada e calcular a iluminação em todos os pontos avaliados.

6 RESULTADOS

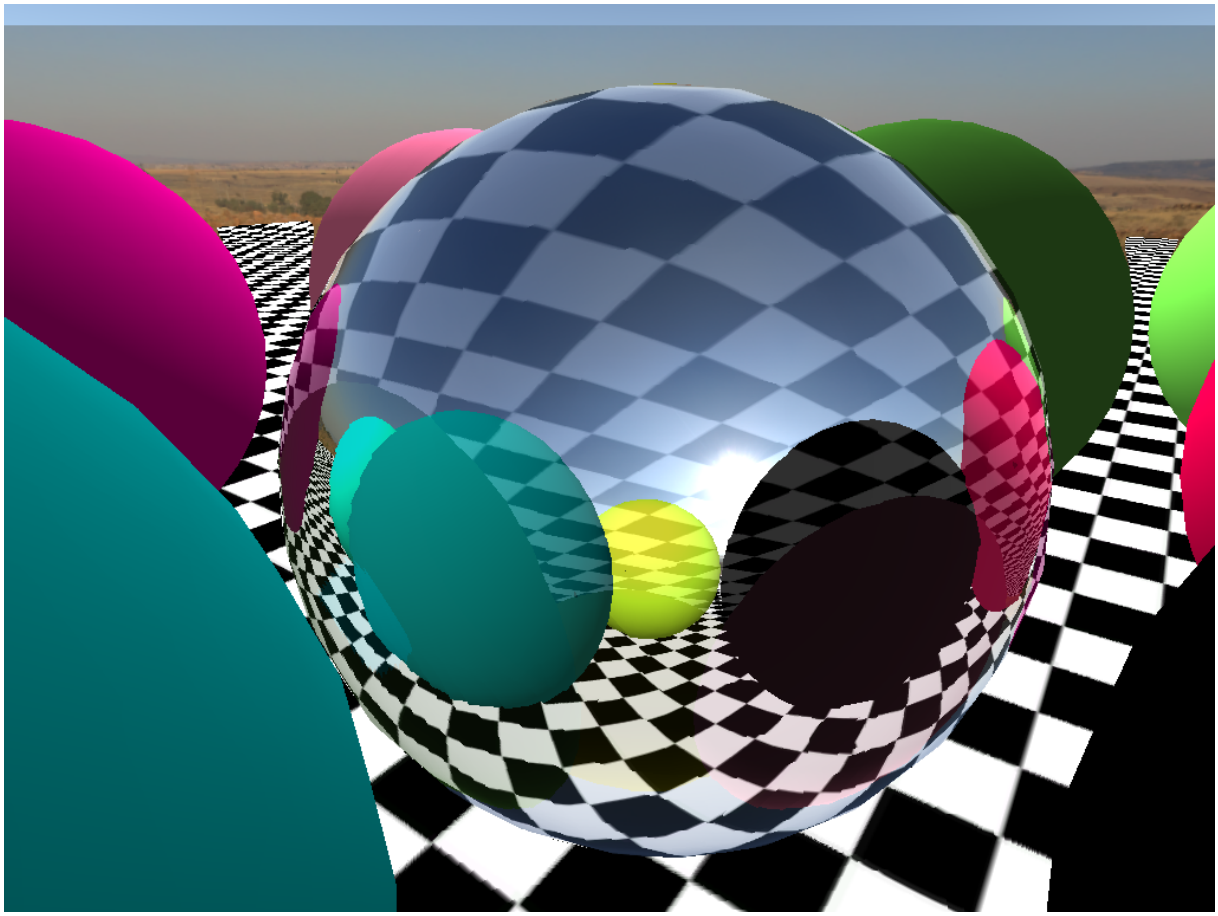
A técnica proposta foi implementada em C# utilizando a *engine* de jogos *Unity* e compute *shader* escrito em HLSL. Embora tenha-se conseguido implementar com sucesso todo o processo de renderização, algumas limitações foram encontradas em todo o desenvolvimento como, por exemplo, muitas funcionalidades padrões da *Unity* não podem ser desativadas, impactando na performance.

O hardware em que o sistema foi avaliado consiste de uma CPU Intel Core i5-4460 de 3.2GHz com 8GB de RAM e uma GPU Nvidia GTX 1070 (8GB VRAM) e os testes foram realizados utilizando resolução 720p (1280x720). Para comprovar a técnica apresentada, foram testados diversos cenários onde é avaliado a performance de diversas configurações das resolução do *hash* e da *octree*. Além disso, diversos objetos com diferentes números de triângulos foram testados e a solução proposta foi comparado com o *ray tracing* tradicional. A solução proposta engloba todas as qualidades gráficas apresentadas pelo algoritmo de *ray tracing*, como reflexão e refração, enquanto melhora a performance. As Figuras 6.1, 6.2, 6.3 e 6.4 apresentam o resultado do *ray tracing* em tempo real de uma cena composta de diferentes configurações de objetos. Para fins visuais, foi adicionado uma *skybox* a partir uma textura panorâmica utilizando amostragem esférica. O padrão de refração utilizado baseia-se na lei de Snell, que define a relação entre o ângulo de incidência e o de refração entre diferentes meios isotrópicos.

O desempenho está diretamente relacionado com a quantidade de passos ao se calcular intersecção com as geometrias. O cálculo de intersecção inicia com dois laços, e é de extrema importância otimizá-los pois é indicado minimizar o uso de branchs em GPU. O primeiro laço é de avaliação da *octree* e o segundo é o laço entre todos os objetos resultantes da busca na *octree*. Profundidades de árvore mais altas exigirão mais passos para percorrer e, portanto, impactando negativamente na performance. No entanto, ao se reduzir a profundidade da árvore, aumenta-se o número de objetos por nó sobrecarregando o segundo laço e também impactando negativamente na performance. Assim, pode-se diminuir a profundidade da *octree* aumentando a granularidade do *hash*, onde cada nó é uma *octree* de tamanho reduzido com tempo de busca O(1). O resultado da busca da *octree* são os SDFs referentes aquela região. O cálculo de intersecção deve também avaliar as estruturas encontradas pela busca da *octree* para recuperar a distância até a superfície. Sendo assim, é preferível que a abordagem escolhida para representação da malha genérica minimize o tempo de avaliação.

Com isso em mente, os parâmetros escolhidos para a geração da estrutura Scene SDF são altamente dependentes da cena. Observou-se que para as cenas teste, que são compostas de três mil objetos condensados num espaço de $140 \times 100 \times 140$, um *hash* com nós de tamanho $10 \times 10 \times 10$ distribuiu de maneira eficiente os objetos da cena na estrutura,

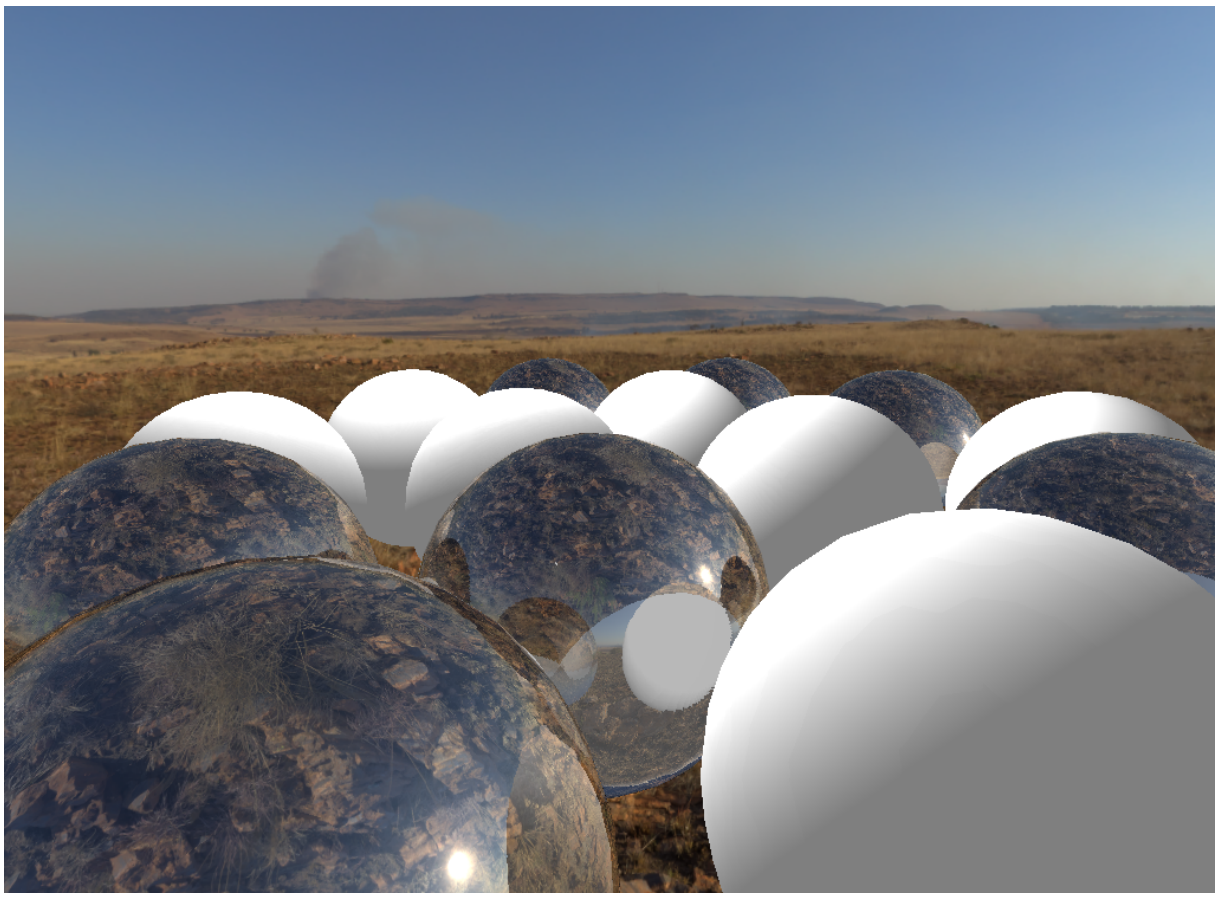
Figura 6.1 – Imagens do resultado do *ray tracing* em tempo real de uma cena composta de nove esferas opacas de diferentes albedos ao arredor de uma esfera reflexiva e refrativa postas no topo de um tabuleiro de xadrez.



evitando sobrecarga em um dos laços. O desempenho de cada configuração é apresentado nas Tabelas 6.1 e 6.2. Ao analisar as tabelas, pode-se observar que a estrutura *hash* é mais eficiente que a *octree*, porém utilizar somente o *hash* limita o tamanho e a configuração da cena. Isso ocorre pois grandes espaços vazios são subdivididos igualmente pelo *hash*, enquanto a *octree* se ajusta automaticamente a geometria da cena. Evidentemente, pode-se notar uma diferença de pelo menos 10% na performance nos casos mais extremos comparando as duas configurações e isso está relacionado à diferença no tempo de resposta de ambas as estruturas.

Para validar a performance da solução proposta, foi implementado o *ray tracing* original baseado em colisão com malhas trigonométricas. Essa abordagem utiliza otimizações simples como o pré-requisito de colidir com uma AABB para iniciar o cálculo de colisão com a geometria. Porém, foi necessário remover reflexão e refração somente da implementação do *ray tracing* original. Para isso, foram montados quatro cenários com diferentes números de objetos com diferentes complexidades e comparado o tempo neces-

Figura 6.2 – Imagens do resultado do *ray tracing* em tempo real de uma cena composta de cinco esferas reflexivas e refrativas e cinco esferas opacas, ilustrando a convergência de diversos raios reflexivos e refrativos.



sário para renderizar cada *frame* e a estabilidade do FPS (Tabela 6.3). Assim, percebeu-se uma diferença de pelo menos 20 vezes. Claramente pode-se observar as vantagens na performance ao utilizar a solução proposta nesta dissertação para o cálculo de intersecção.

As estruturas de *hash* e *octree* podem ser visualizadas nas figuras 6.6 e 6.7 respectivamente. O *hash* compreende todos os objetos e a *octree* é respectiva a um nó do *hash*. A Figura 6.5 apresenta um objeto criado à partir de múltiplas operações de união, intersecção e subtração de fórmulas matemáticas de geometrias implícitas, além disso as coordenadas uv também foram geradas à partir de sua fórmula matemática. Essa geometria é composta de sete elipsóides e nove cápsulas.

Figura 6.3 – Imagens do resultado do *ray tracing* em tempo real de uma cena composta por quatro diferentes malhas trigonométricas: o *bunny* (4968 triângulos), o *teapot* (4032 triângulos), o monkey (968 triângulos) e a esfera (768 triângulos). Ao todo são vinte e dois objetos de diferentes materiais.

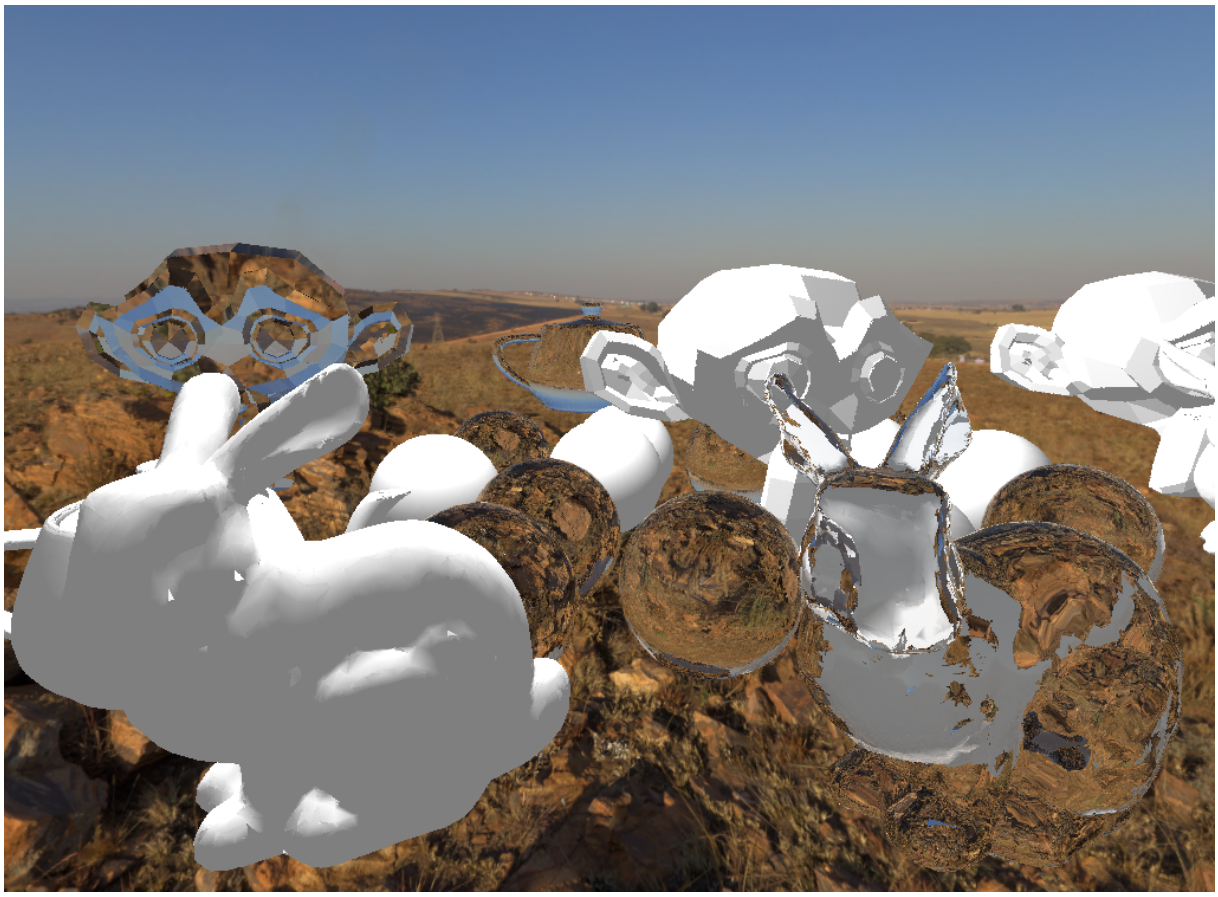


Tabela 6.1 – Comparação da performance de diversas malhas utilizando somente hash, onde cada célula tem um tamanho de $10 \times 10 \times 10$ e *octree* com dimensão mínima do nó de 1 unidade.

	1 Objeto	**10 Objetos**	**100 Objetos**	**1000 Objetos**
Bunny (4968 triângulos)	380 FPS	100 FPS	28 FPS	19 FPS
Teapot (4032 triângulos)	390 FPS	105 FPS	29 FPS	20 FPS
Monkey (968 triângulos)	450 FPS	112 FPS	33 FPS	25 FPS
Sphere (768 triângulos)	455 FPS	118 FPS	33 FPS	27 FPS

Figura 6.4 – Imagens do resultado do *ray tracing* em tempo real de uma cena composta por quatro diferentes malhas trigonométricas: o *bunny* (4968 triângulos), o *teapot* (4032 triângulos), o monkey (968 triângulos) e a esfera (768 triângulos). Ao todo são vinte e dois objetos de diferentes materiais.

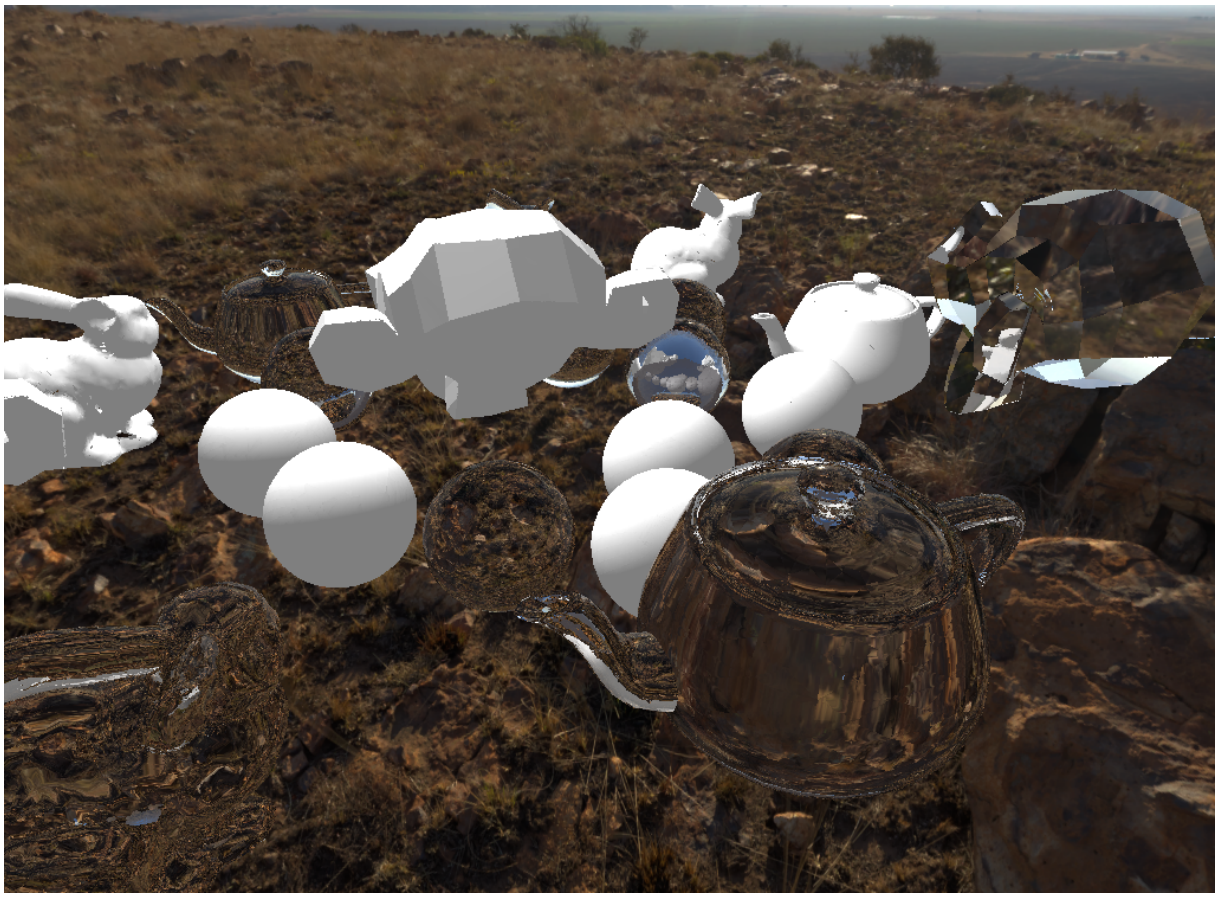


Tabela 6.2 – Comparação da performance de diversas malhas utilizando somente *hash*, onde cada célula tem um tamanho de $1 \times 1 \times 1$, sem a utilizar *octree*.

	1 Objeto	**10 Objetos**	**100 Objetos**	**1000 Objetos**
Bunny (4968 triângulos)	395 FPS	110 FPS	31 FPS	23 FPS
Teapot (4032 triângulos)	403 FPS	111 FPS	33 FPS	23 FPS
Monkey (968 triângulos)	461 FPS	117 FPS	35 FPS	27 FPS
Sphere (768 triângulos)	472 FPS	124 FPS	36 FPS	29 FPS

Figura 6.5 – Exemplo da utilização de operações como união, intersecção e subtração de fórmulas matemáticas de geometrias implícitas para gerar um único SDF com geração de coordenadas uv.

Tabela 6.3 – Comparação da performance entre diferentes malhas com diferentes complexidades usando *ray tracing* tradicional.

	1 Objeto	**10 Objetos**	**100 Objetos**	**1000 Objetos**
Bunny (4968 triângulos)	9 FPS	1 FPS	<1 FPS	<1 FPS
Teapot (4032 triângulos)	10 FPS	1 FPS	<1 FPS	<1 FPS
Monkey (968 triângulos)	60 FPS	3 FPS	1 FPS	<1 FPS
Sphere (768 triângulos)	65 FPS	6 FPS	2 FPS	<1 FPS

Figura 6.6 – Estrutura do *hash* com resolução de $14 \times 10 \times 14$ e cada célula com tamanho de 10 unidades em cada eixo, criada à partir da posição dos objetos no mundo tridimensional.

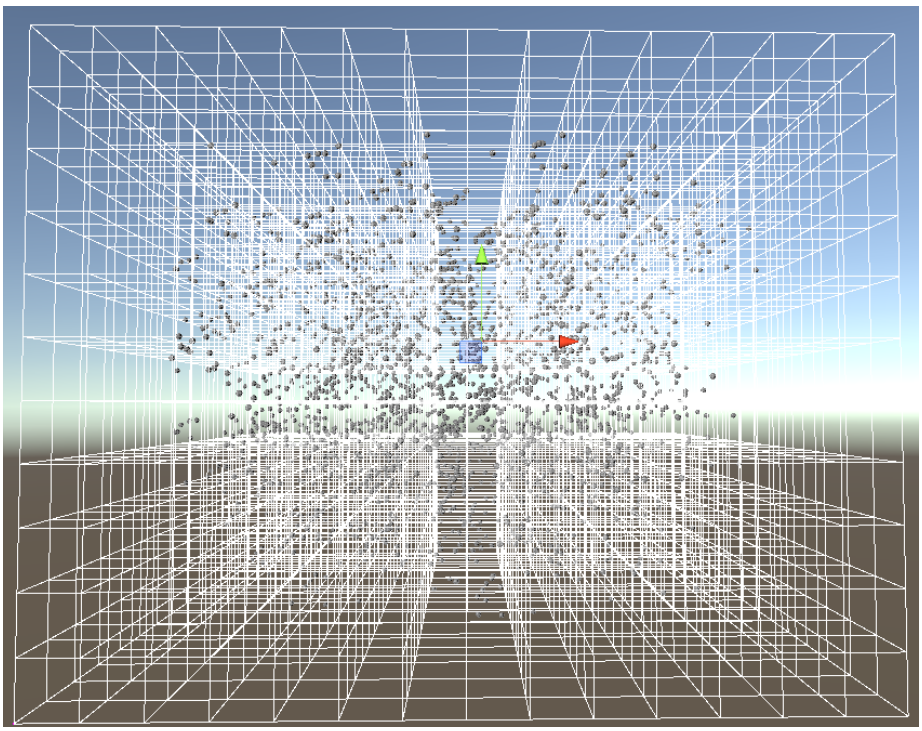
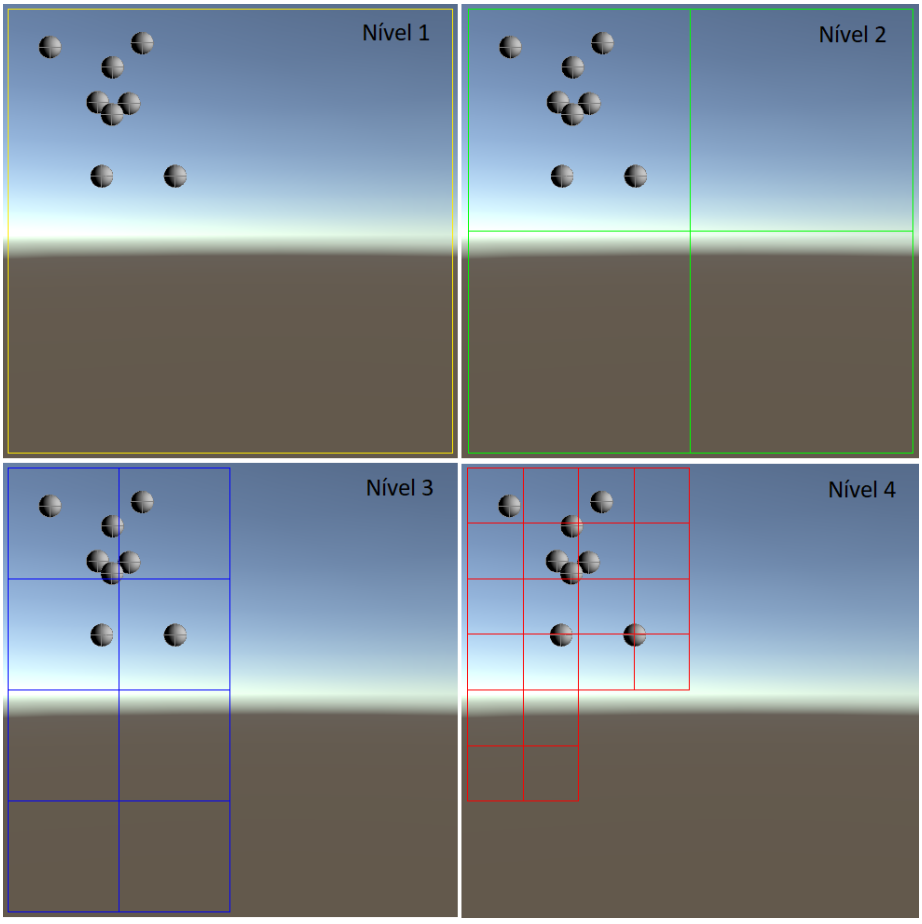


Figura 6.7 – Diversos níveis da *octree* gerada em um nó do *hash* à partir da área de influência dos objetos. O quadrado amarelo representa o nível mais alto da *octree*. Os quadrados verdes, azuis e vermelhos representam a subdivisão dos quadrados do nível anterior, de acordo com a área de influência dos objetos.



7 CONCLUSÃO

Este trabalho teve como principal objetivo a concepção de um novo método de representação de cenas baseado no paradigma matemático SDF e subdivisão espacial. Para este fim, foi realizada uma extensa pesquisa sobre diversos cenários nas quais a utilização desse paradigma agiliza o processamento. A proposta apresentada é a primeira que utiliza uma solução global que substitui totalmente a maneira de representar uma cena, característica que tem a capacidade de substituir o pipeline gráfico atual usado para renderizar cenas tridimensionais. Deste modo, a abordagem proposta visou maximizar a eficiência desde a criação do SDF dos objetos até o uso da estrutura criada para geração da imagem pelo algoritmo de *ray tracing*.

Os resultados obtidos mostram o potencial da solução criada. Ao verificar a comparação de performance da abordagem proposta com o *ray tracing* tradicional no Capítulo 6, pode-se notar um ganho de desempenho de pelo menos 20 vezes enquanto a qualidade visual mantém-se. Deste modo, a principal contribuição deste trabalho foi uma nova maneira de representar uma cena e seus objetos de forma global e que tem a capacidade de agilizar diversos algoritmos como *ray tracing* (JAMRIŠKA, 2010), reconstrução de superfície (TAUBIN, 2012), modelagem geométrica (FRISKEN; PERRY, 2006), detecção de colisão (MITCHELL et al., 2015), entre outros. O método apresentado gerencia múltiplas instâncias de objetos organizados em uma estrutura global que representa a cena armazenando diversas propriedades escolhidas referente a cada objeto agilizando o cálculo de intersecção de um raio com a cena.

7.1 TRABALHOS FUTUROS

Embora os resultados mostrem que a abordagem proposta é capaz de renderizar com eficiência cenas com um grande número de objetos com um alto ganho de performance. A área de influência dos objetos é aproximada e é maior do que necessário resultando em objetos presentes em nós onde não há sua influência, causando um impacto negativo na performance ao adicionar mais objetos ao *loop*. Uma possibilidade para a realização de trabalhos futuros é desenvolver métodos para gerar a área de influência precisa de cada objeto e verificar seu impacto na performance e no tempo de pré-processamento. Outra possibilidade de trabalhos futuros é portar a solução para um ambiente externo ao Unity3D, como OpenGL, DirectX ou Vulkan. Outra possibilidade para trabalhos futuros é utilizar a representação proposta para implementar uma solução global de física.

REFERÊNCIAS BIBLIOGRÁFICAS

AGARWAL, P. **Handbook of Discrete and Computational Geometry, chapter Range Searching**. [S.l.]: CRC Press LLC, Boca Raton, FL, 1997.

AGARWAL, P. K.; ERICKSON, J. et al. Geometric range searching and its relatives. **Contemporary Mathematics**, Providence, RI: American Mathematical Society, v. 223, p. 1–56, 1999.

AKENINE-MOLLER, T.; HAINES, E.; HOFFMAN, N. **Real-time rendering**. [S.l.]: AK Peters/CRC Press, 2018.

ALLARD, J. et al. Volume contact constraints at arbitrary resolution. In: ACM. **ACM Transactions on Graphics (TOG)**. [S.l.], 2010. v. 29, n. 4, p. 82.

ANDRADE, P. et al. A heuristic approach to render ray tracing effects in real time for first-person games. **SBC Journal on Interactive Systems**, v. 5, n. 1, p. 26–33, 2014.

ANDRADE, P.; SABINO, T.; CLUA, E. Towards a heuristic based real time hybrid rendering a strategy to improve real time rendering quality using heuristics and ray tracing. In: IEEE. **2014 International Conference on Computer Vision Theory and Applications (VISAPP)**. [S.l.], 2014. v. 3, p. 12–21.

APPEL, A. Some techniques for shading machine renderings of solids. In: ACM. **Proceedings of the April 30–May 2, 1968, spring joint computer conference**. [S.l.], 1968. p. 37–45.

AR, S.; CHAZELLE, B.; TAL, A. Self-customized bsp trees for collision detection. **Computational Geometry**, Elsevier, v. 15, n. 1-3, p. 91–102, 2000.

ARONOV, B.; FORTUNE, S. Approximating minimum-weight triangulations in three dimensions. **Discrete & Computational Geometry**, Springer, v. 21, n. 4, p. 527–549, 1999.

ARVO, J.; KIRK, D. A survey of ray tracing acceleration techniques. **An introduction to ray tracing**, p. 201–262, 1989.

BÆRENTZEN, J. A. Manipulation of volumetric solids with applications to sculpting. In: CITESEER. [S.l.], 2002.

____. Robust generation of signed distance fields from triangle meshes. In: IEEE. **Volume Graphics, 2005. Fourth International Workshop on**. [S.l.], 2005. p. 167–239.

BARBIČ, J.; JAMES, D. L. Six-dof haptic rendering of contact between geometrically complex reduced deformable models. **IEEE Transactions on Haptics**, IEEE, v. 1, n. 1, 2008.

BAREQUET, G. et al. Boxtree: A hierarchical representation for surfaces in 3d. In: WILEY ONLINE LIBRARY. **Computer Graphics Forum**. [S.l.], 1996. v. 15, n. 3, p. 387–396.

BAREQUET, G.; HAR-PELED, S. Efficiently approximating the minimum-volume bounding box of a point set in three dimensions. **Journal of Algorithms**, Elsevier, v. 38, n. 1, p. 91–109, 2001.

BECK, S. et al. Cpu-gpu hybrid real time ray tracing framework. Citeseer, 2005.

BENTLEY, J. L. Multidimensional binary search trees used for associative searching. **Communications of the ACM**, ACM, v. 18, n. 9, p. 509–517, 1975.

BENTLEY, J. L.; FRIEDMAN, J. H. Data structures for range searching. **ACM Computing Surveys (CSUR)**, ACM, v. 11, n. 4, p. 397–409, 1979.

BERG, M. de et al. Efficient ray shooting and hidden surface removal. **Algorithmica**, Springer, v. 12, n. 1, p. 30–53, 1994.

BRIDSON, R.; MARINO, S.; FEDKIW, R. Simulation of clothing with folds and wrinkles. In: ACM. **ACM SIGGRAPH 2005 Courses**. [S.l.], 2005. p. 3.

CABELEIRA, J. Combining rasterization and ray tracing techniques to approximate global illumination in real-time. **Master's Thesis, Engenharia Informática e de Computadores, Universidade Técnica de Lisboa, Portugal**, 2010.

CAZALS, F.; DRETTAKIS, G.; PUECH, C. Filtering, clustering and hierarchy construction: a new solution for ray-tracing complex scenes. In: WILEY ONLINE LIBRARY. **Computer graphics forum**. [S.l.], 1995. v. 14, n. 3, p. 371–382.

CAZALS, F.; PUECH, C. Bucket-like space partitioning data structures with applications to ray-tracing. In: **13th ACM Symposium on Computational Geometry**. [S.l.: s.n.], 1997.

CRASSIN, C. et al. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In: ACM. **Proceedings of the 2009 symposium on Interactive 3D graphics and games**. [S.l.], 2009. p. 15–22.

DELFOSSE, J.; HEWITT, W. T.; MÉRIAUX, M. An investigation of discrete ray-tracing. In: CITESEER. **4th Discrete Geometry in Computer Imagery Conference**. [S.l.], 1994. p. 65–76.

DEVILLERS, O. The macro-regions, an efficient space subdivision structure for ray tracing. In: **Eurographics**. [S.l.: s.n.], 1989. p. 27–38.

DOOM. Doom. 2000.

ERLEBEN, K.; DOHLMANN, H. Signed distance fields using single-pass gpu scan conversion of tetrahedra. In: **Gpu Gems 3**. [S.l.]: Addison-Wesley, 2008. p. 741–763.

FAURE, F. et al. Image-based collision detection and response between arbitrary volume objects. In: EUROGRAPHICS ASSOCIATION. **Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation**. [S.l.], 2008. p. 155–162.

FRISKEN, S. F.; PERRY, R. N. Designing with distance fields. In: ACM. **ACM SIGGRAPH 2006 Courses**. [S.l.], 2006. p. 60–66.

FRISKEN, S. F. et al. Adaptively sampled distance fields: A general representation of shape for computer graphics. In: ACM PRESS/ADDISON-WESLEY PUBLISHING CO. **Proceedings of the 27th annual conference on Computer graphics and interactive techniques**. [S.l.], 2000. p. 249–254.

FUCHS, H.; KEDEM, Z. M.; NAYLOR, B. F. On visible surface generation by a priori tree structures. In: ACM. **ACM Siggraph Computer Graphics**. [S.l.], 1980. v. 14, n. 3, p. 124–133.

FUHRMANN, A.; SOBOTKA, G.; GROSS, C. Distance fields for rapid collision detection in physically based modeling. In: **Proceedings of GraphiCon 2003**. [S.l.: s.n.], 2003. p. 58–65.

FUJIMOTO, A.; IWATA, K. Accelerated ray tracing. In: **Computer Graphics**. [S.l.]: Springer, 1985. p. 41–65.

FUJIMOTO, A.; TANAKA, T.; IWATA, K. Arts: Accelerated ray-tracing system. **IEEE Computer Graphics and Applications**, IEEE, v. 6, n. 4, p. 16–26, 1986.

GLASSNER, A. S. Space subdivision for fast ray tracing. **IEEE Computer Graphics and applications**, IEEE, v. 4, n. 10, p. 15–24, 1984.

____. **An introduction to ray tracing**. [S.l.]: Elsevier, 1989.

GLONDU, L. et al. Efficient collision detection for brittle fracture. In: EUROGRAPHICS ASSOCIATION. **Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation**. [S.l.], 2012. p. 285–294.

GUNTHER, J. et al. Realtime ray tracing on gpu with bvh-based packet traversal. In: IEEE. **2007 IEEE Symposium on Interactive Ray Tracing**. [S.l.], 2007. p. 113–118.

HAINES, E. Efficiency improvements for hierarchy traversal in ray tracing. In: **Graphics Gems II**. [S.l.]: Elsevier, 1991. p. 267–272.

HAINES, E. A.; GREENBERG, D. P. The light buffer: A shadow-testing accelerator. **IEEE Computer Graphics and Applications**, IEEE, v. 6, n. 9, p. 6–16, 1986.

HART, J. C. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. **The Visual Computer**, Springer, v. 12, n. 10, p. 527–545, 1996.

HERTEL, S.; HORMANN, K.; WESTERMANN, R. A hybrid gpu rendering pipeline for alias-free hard shadows. In: CITESEER. **Eurographics (Areas Papers)**. [S.l.], 2009. p. 59–66.

HUANG, J. et al. A complete distance field representation. In: IEEE COMPUTER SOCIETY. **Proceedings of the conference on Visualization'01**. [S.l.], 2001. p. 247–254.

JAMES, A. **Binary space partitioning for accelerated hidden surface removal and rendering of static environments.** 1999. Tese (Doutorado) — University of East Anglia, 1999.

JAMRIŠKA, O. Interactive ray tracing of distance fields. In: **Central European Seminar on Computer Graphics**. [S.l.: s.n.], 2010. v. 2, p. 1–7.

JONES, M. W. Distance field compression. UNION Agency, 2004.

JR, F. S. H. **Computer graphics using open gl**. [S.l.]: Pearson education, 2008.

JU, T. et al. Dual contouring of hermite data. In: ACM. **ACM transactions on graphics (TOG)**. [S.l.], 2002. v. 21, n. 3, p. 339–346.

KAJIYA, J. T. **New techniques for ray tracing procedurally defined objects**. [S.l.]: ACM, 1983. v. 17.

KAUFMAN, D. M.; SUEDA, S.; PAI, D. K. Contact trees: adaptive contact sampling for robust dynamics. In: **SIGGRAPH Sketches**. [S.l.: s.n.], 2007. p. 16.

KAY, T. L.; KAJIYA, J. T. Ray tracing complex scenes. In: ACM. **ACM SIGGRAPH computer graphics**. [S.l.], 1986. v. 20, n. 4, p. 269–278.

KLOSOWSKI, J. T. et al. Efficient collision detection using bounding volume hierarchies of k-dops. **IEEE transactions on Visualization and Computer Graphics**, IEEE, v. 4, n. 1, p. 21–36, 1998.

KOSCHIER, D.; DEUL, C.; BENDER, J. Hierarchical hp-adaptive signed distance fields. In: **Symposium on Computer Animation**. [S.l.: s.n.], 2016. p. 189–198.

LEVOY, M. Efficient ray tracing of volume data. **ACM Transactions on Graphics (TOG)**, ACM, v. 9, n. 3, p. 245–261, 1990.

LIU, F.; KIM, Y. J. Exact and adaptive signed distance fieldscomputation for rigid and deformablemodels on gpus. **IEEE transactions on visualization and computer graphics**, IEEE, v. 20, n. 5, p. 714–725, 2014.

MACDONALD, J. D.; BOOTH, K. S. Heuristics for ray tracing using space subdivision. **The Visual Computer**, Springer, v. 6, n. 3, p. 153–166, 1990.

MITCHELL, N. et al. Non-manifold level sets: A multivalued implicit surface representation with applications to self-collision processing. **ACM Transactions on Graphics (TOG)**, ACM, v. 34, n. 6, p. 247, 2015.

MOUSTAKAS, K.; TZOVARAS, D.; STRINTZIS, M. G. Sq-map: Efficient layered collision detection and haptic rendering. **IEEE Transactions on Visualization & Computer Graphics**, IEEE, n. 1, p. 80–93, 2007.

MURALI, T. **Efficient hidden-surface removal in theory and in practice**. [S.l.]: Brown University, 1999.

MUSETH, K. Vdb: High-resolution sparse volumes with dynamic topology. **ACM Transactions on Graphics (TOG)**, ACM, v. 32, n. 3, p. 27, 2013.

NAYLOR, B. F. Interactive solid geometry via partitioning trees. In: **Proc. Graphics Interface**. [S.l.: s.n.], 1992. v. 92, p. 11–18.

O'ROURKE, J. Finding minimal enclosing boxes. **International journal of computer & information sciences**, Springer, v. 14, n. 3, p. 183–199, 1985.

OTADUY, M. A. et al. Haptic display of interaction between textured models. In: IEEE. **Visualization, 2004. IEEE**. [S.l.], 2004. p. 297–304.

PENG, Q.; ZHU, Y.; LIANG, Y. A fast ray tracing algorithm using space indexing techniques. In: **Proceedings of Eurographics**. [S.l.: s.n.], 1987. v. 87, p. 11–23.

PERRY, R. N.; FRISKEN, S. F. Kizamu: A system for sculpting digital characters. In: ACM. **Proceedings of the 28th annual conference on Computer graphics and interactive techniques**. [S.l.], 2001. p. 47–56.

QU, H. et al. Feature preserving distance fields. In: IEEE. **Volume Visualization and Graphics, 2004 IEEE Symposium on**. [S.l.], 2004. p. 39–46.

QUILEZ, I. Modeling with distance functions. 2008. Acessed 2018-08-03. Disponível em: <http://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm>.

____. Raymarching distance fields. 2008. Acessed 2018-08-03. Disponível em: <http://iquilezles.org/www/articles/raymarchingdf/raymarchingdf.htm>.

REVELLES, J.; URENA, C.; LASTRA, M. An efficient parametric algorithm for octree traversal. Václav Skala-UNION Agency, 2000.

ROSENFELD, A.; PFALTZ, J. L. Sequential operations in digital picture processing. **Journal of the ACM (JACM)**, ACM, v. 13, n. 4, p. 471–494, 1966.

SABINO, P. M. A. T. L.; PAGLIOSA, E. W. C. P. A heuristic to selectively ray trace light effects in real time. 2012.

SABINO, T. L. et al. A hybrid gpu rasterized and ray traced rendering pipeline for real time rendering of per pixel effects. In: SPRINGER. **International Conference on Entertainment Computing**. [S.l.], 2012. p. 292–305.

SAMET, H. Implementing ray tracing with octrees and neighbor finding. **Computers & Graphics**, Elsevier, v. 13, n. 4, p. 445–460, 1989.

____. **The design and analysis of spatial data structures**. [S.l.]: Addison-Wesley Reading, MA, 1990. v. 85.

SANCHEZ, M.; FRYAZINOV, O.; PASKO, A. Efficient evaluation of continuous signed distance to a polygonal mesh. In: ACM. **Proceedings of the 28th Spring Conference on Computer Graphics**. [S.l.], 2013. p. 101–108.

SANDOR, J. Octree data structures and perspective imagery. **Computers & graphics**, Elsevier, v. 9, n. 4, p. 393–405, 1985.

SPACKMAN, J.; WILLIS, P. The smart navigation of a ray through an oct-tree. **Computers & graphics**, Elsevier, v. 15, n. 2, p. 185–194, 1991.

TAUBIN, G. Smooth signed distance surface reconstruction and applications. In: SPRINGER. **Iberoamerican Congress on Pattern Recognition**. [S.l.], 2012. p. 38–45.

WADE, B. **BSP Tree Frequently Asked Questions**. 2001.

WEGHORST, H.; HOOPER, G.; GREENBERG, D. P. Improved computational methods for ray tracing. **ACM Transactions on Graphics (TOG)**, ACM, v. 3, n. 1, p. 52–69, 1984.

WHITTED, T. An improved illumination model for shaded display. In: ACM. **ACM SIGGRAPH Computer Graphics**. [S.l.], 1979. v. 13, n. 2, p. 14.

WOO, A. Fast ray-box intersection. In: ACADEMIC PRESS PROFESSIONAL, INC. **Graphics gems**. [S.l.], 1990. p. 395–396.

WU, J.; KOBBELT, L. Piecewise linear approximation of signed distance fields. In: **VMV**. [S.l.: s.n.], 2003. p. 513–520.

XU, H.; BARBIČ, J. Signed distance fields for polygon soup meshes. In: CANADIAN INFORMATION PROCESSING SOCIETY. **Proceedings of Graphics Interface 2014**. [S.l.], 2014. p. 35–41.

XU, H.; ZHAO, Y.; BARBIC, J. Implicit multibody penalty-baseddistributed contact. **IEEE Transactions on Visualization & Computer Graphics**, IEEE, n. 9, p. 1266–1279, 2014.

YAGEL, R.; COHEN, D.; KAUFMAN, A. Discrete ray tracing. **IEEE Computer graphics and applications**, IEEE, n. 5, p. 19–28, 1992.

YOUSSEF, S. A new algorithm for object oriented ray tracing. **Computer Vision, Graphics, and Image Processing**, Elsevier, v. 34, n. 2, p. 125–137, 1986.

ZACHMANN, G. The boxtree: Exact and fast collision detection of arbitrary polyhedra. In: **First Workshop on Simulation and Interaction in Virtual Environments (SIVE 95)**. [S.l.: s.n.], 1995.