Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Monografia Final
Trabalho de Formatura Supervisionado
MAC0499

Renderização em Tempo Real Utilizando *Ray Marching* e sua Aplicação em Jogos Digitais

Pedro Tonini Rosenberg Schneider
Orientador: Prof. Dr. Márcio Lobo Netto

São Paulo
Dezembro de 2024

Agradecimentos

Ao Prof. Dr. Márcio Lobo Netto por sua orientação durante o ano de desenvolvimento dessa monografia. Ao César Gasparini Fernandes por sua indispensável ajuda na revisão e melhoria deste trabalho. À Clara Yuki Sano por sua ajuda, apoio e paciência nos momentos mais difíceis. E a meus pais, Raquel Tonini Rosenberg Schneider e Victor Queiroz Schneider, irmãos, Clara Rosenberg Schneider, Laura Rosenberg Schneider e Lucas Tonini Rosenberg Schneider, avós, Aide Maria Tonini Rozemberg, Maria Carmem Queiroz Schneider e José Luiz Serrano Schneider, e toda a família por sempre me incentivarem a estudar, aprender e melhorar como pessoa. Esse trabalho nunca teria sido concluído sem o apoio de vocês.

Resumo

Pedro Tonini Rosenberg Schneider. **Renderização em Tempo Real Utilizando *Ray Marching* e sua Aplicação em Jogos Digitais**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, 2024.

Este trabalho buscou investigar a viabilidade do uso da técnica de renderização *Ray Marching*, com foco no algoritmo *Sphere Tracing*, para aplicações interativas em tempo real, como jogos digitais. O principal objetivo foi avaliar os benefícios e as limitações dessa abordagem em comparação com os métodos tradicionais, como rasterização e *Ray Tracing*, amplamente utilizados na indústria de jogos. Para tal, foi desenvolvido um renderizador como prova de conceito utilizando a técnica de *Ray Marching* acelerado por *Sphere Tracing*. A metodologia adotada envolveu uma revisão teórica das principais técnicas de renderização, explorando os fundamentos do *Ray Marching*, suas variantes (como marcha uniforme e *Sphere Tracing*) e conceitos relacionados, como funções de distância e operações de manipulação de domínios. O trabalho incluiu a implementação de algoritmos em linguagens como Slang e Rust para a definição de formas primitivas, transformações geométricas, materiais e modelos 3D. Os resultados obtidos demonstraram que o *Ray Marching*, embora apresente limitações em termos de desempenho em cenas modeladas tradicionalmente com malhas de triângulos, oferece vantagens significativas no campo criativo, permitindo a renderização eficiente de superfícies implícitas e estruturas complexas, como fractais. Além disso, foi constatado que a técnica é capaz de simular efeitos avançados de iluminação, como sombras suaves e reflexões, com maior controle sobre os detalhes da cena. Conclui-se que o *Ray Marching*, em particular o *Sphere Tracing*, é uma abordagem promissora para aplicações que buscam inovação visual e flexibilidade criativa. No entanto, seu uso em cenários comerciais ainda depende de avanços no desenvolvimento de *hardware* dedicado e integração com os padrões atuais da indústria. Este trabalho contribuiu para o entendimento e a exploração de novas possibilidades na renderização gráfica, destacando o potencial do *Ray Marching* como uma alternativa inovadora para o desenvolvimento de jogos digitais.

Palavras-chave: Ray Marching, Sphere Tracing, renderização em tempo real, jogos digitais, superfícies implícitas.

Abstract

Pedro Tonini Rosenberg Schneider. **Real Time Rendering Using Ray Marching and its Application in Digital Games**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, 2024.

This study aimed to investigate the feasibility of using the Ray Marching rendering technique, focusing on the Sphere Tracing algorithm, for real-time interactive applications such as digital games. The main objective was to evaluate the benefits and limitations of this approach compared to traditional methods like rasterization and Ray Tracing, which are widely used in the gaming industry. To this end, a renderer was developed as a proof of concept using the Ray Marching technique accelerated by Sphere Tracing. The adopted methodology involved a theoretical review of the main rendering techniques, exploring the fundamentals of Ray Marching, its variants (such as uniform marching and Sphere Tracing), and related concepts, including distance functions and domain manipulation operations. The work included the implementation of algorithms in languages like Slang and Rust for defining primitive shapes, geometric transformations, materials, and 3D models. The results showed that Ray Marching, although limited in performance for traditionally modeled scenes with triangle models, offers significant advantages in the creative field, enabling the efficient rendering of implicit surfaces and complex structures, such as fractals. Additionally, it was observed that the technique is capable of simulating advanced lighting effects, such as soft shadows and reflections, with greater control over scene details. It is concluded that Ray Marching, particularly Sphere Tracing, is a promising approach for applications seeking visual innovation and creative flexibility. However, its use in commercial scenarios still depends on advances in the development of dedicated hardware and integration with current industry standards. This study contributed to the understanding and exploration of new possibilities in graphical rendering, highlighting the potential of Ray Marching as an innovative alternative for digital game development.

Keywords: Ray Marching, Sphere Tracing, real-time rendering, digital games, implicit surfaces.

Lista de Figuras

Lista de Algoritmos

Sumário

Capítulo 1

Introdução

O propósito principal deste trabalho é a avaliação da viabilidade da utilização da técnica de renderização de *Ray Marching*, particularmente utilizando o algoritmo de *Sphere Tracing*, para a renderização de cenas tridimensionais em aplicações interativas de tempo real, como jogos digitais. Além deste estudo, será desenvolvido uma prova de conceito de um renderizador utilizando a técnica de renderização supracitada como uma demonstração prática da viabilidade do algoritmo. Como o foco do trabalho são aplicações interativas, o renderizador terá funcionalidades para tal, sendo possível controlar aspectos das cenas utilizando entradas do usuário.

1.1 Motivação

Atualmente, a técnica de renderização mais utilizada na criação de jogos digitais em 3D é a rasterização [6]. De acordo com um levantamento realizado pela *Dragonfly*[1], das 56 *Game Engines* 3D mais populares[2], todas utilizam rasterização na renderização de gráficos 3D.

Embora tenha sido desenvolvida durante os anos 80 junto com outras técnicas de renderização, como *Ray Tracing*, a rasterização rapidamente se tornou a técnica mais utilizada no desenvolvimento de jogos digitais por diversos fatores, como sua baixa complexidade computacional, acesso a memória previsível e ganhos de performance advindo de *hardware* gráfico dedicado para rasterização [6].

Atualmente, o *Ray Tracing* é utilizado em conjunto com a rasterização para implementar efeitos específicos dentro das cenas, geralmente relacionados a iluminação. Na Figura 1.1, à esquerda, podem ser vistos efeitos de iluminação adicionais, como reflexões na água e materiais emissivos, que não estão presentes à direita. O *Ray Tracing* neste caso é utilizado apenas para estes efeitos adicionais, sendo o restante da cena renderizada por rasterização.

[1] Disponível em: https://www.dragonflydb.io/game-dev/engines/3d, Acesso em 26 de setembro de 2024.
[2] Ordenadas por número de acessos ao *website* das *Game Engines*.

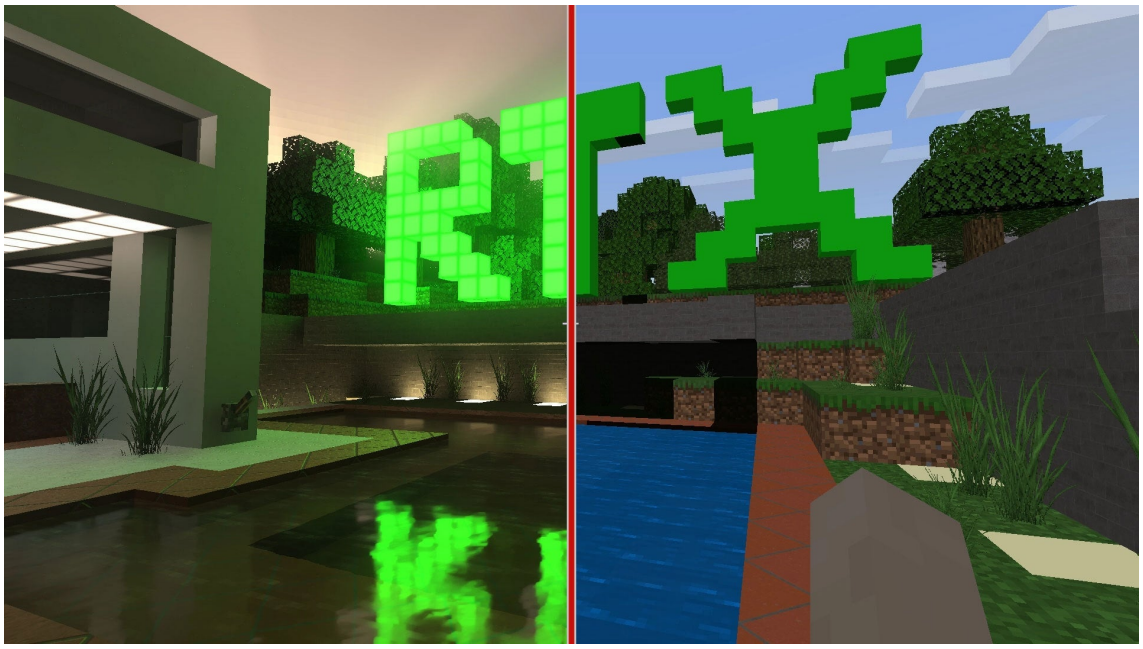

Figura 1.1: Demonstração do uso de *Ray Tracing* no jogo digital *Minecraft*. À esquerda, utilizando *Ray Tracing*, podem ser vistos efeitos de iluminação que não estão presentes à direita, que foi renderizada apenas com rasterização.
Fonte: Mark Knapp, IGN, 2023. Disponível: https://www.ign.com/articles/what-is-ray-tracing

A técnica de *Ray Marching* faz parte do grupo de técnicas de renderização que utilizam raios para simular a propagação da luz. A diferença entre as técnicas baseadas em raios está no algoritmo utilizado para propagar os raios pela cena 3D. Estas distinções serão melhor exploradas no Capítulo 2 desta monografia. *Ray Marching* foi primeiro descrito por John C. Heart, Daniel J. Sandin e Louis H. Kauffman em 1989 [14] em que os autores utilizaram o algoritmo para renderizar fractais tridimensionais, como pode ser visto na Figura 1.2.

Embora tenha sido desenvolvido na mesma época da rasterização, o *Ray Marching* não foi muito utilizado na produção de aplicações interativas, como jogos digitais. Atualmente, a técnica tem ganhado mais atenção na indústria, mas outros algoritmos da família de *Ray Tracing* geralmente são preferidos.

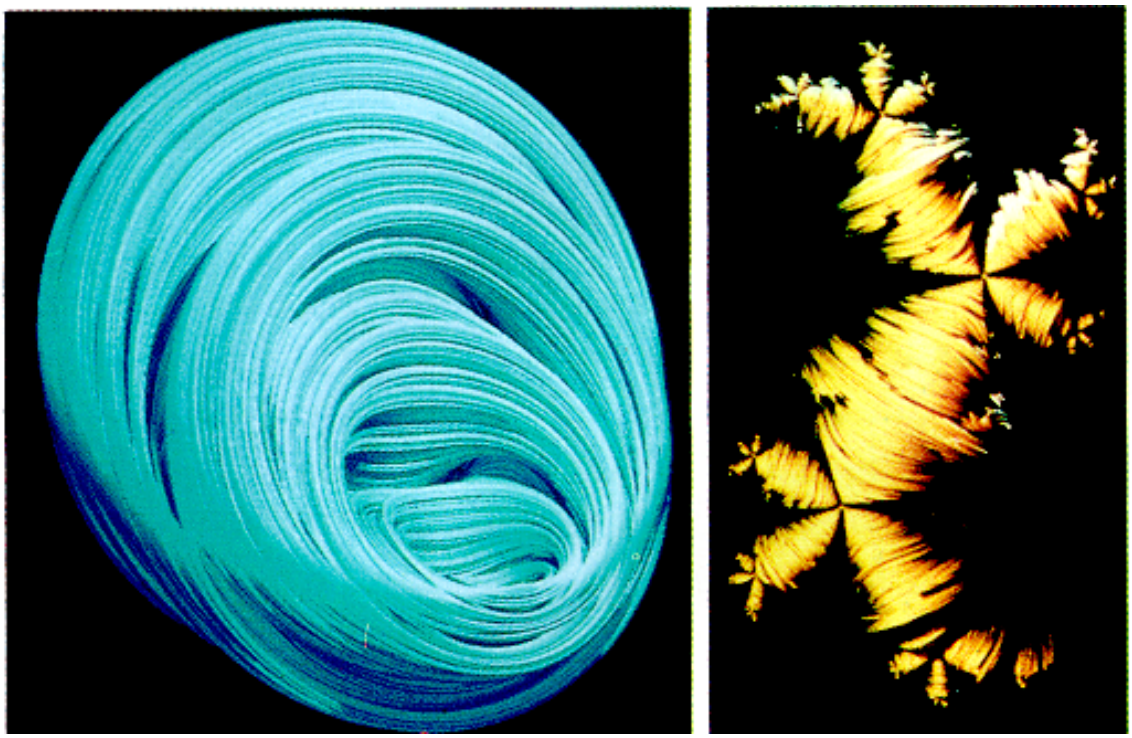

Figura 1.2: Exemplo de fractal 3D renderizado por John C. Heart; et al, utilizando a técnica de *Ray Marching*.
Fonte: John C. Heart; et al. 1989. Disponível:
https://www.evl.uic.edu/hypercomplex/html/book/rtqjs.pdf

Atualmente, o *Ray Marching* vem ganhando mais presença no mercado em grande parte pelo trabalho de Inigo Quilez, ex-diretor técnico de renderização na *Pixar*, que cria conteúdo gratuito em seu *website* [20] sobre renderização, com um foco particular em *Ray Marching* e *Sphere Tracing*. Além disso, há um projeto desenvolvido por Florian Hoenig e Andrea Interguglielmi chamado *Unbound*[3]. Trata-se de uma *Game Engine* baseada em *Ray Marching*, *Sphere Tracing* e funções de distância com sinal (que serão discutidas em detalhes na Seção 2.2). Infelizmente, até a data da escrita desta monografia, o projeto está disponível apenas para um grupo seleto de pessoas no formato *closed beta*, e, portanto, não foi possível utiliza-lo como comparação na pesquisa deste projeto. No entanto, seu desenvolvimento e sua popularidade[4] mostram que há interesse geral nos benefícios que o uso *Ray Marching* proporcionam.

1.2 Objetivos

Este trabalho possui o objetivo principal de avaliar a viabilidade de utilizar a técnica de *Ray Marching* para a renderização de aplicações em tempo real, com foco em jogos digitais. Para tal, será desenvolvido um renderizador como prova de conceito, utilizando como base a técnica de *Ray Marching*. Além disso, como o projeto é focado na aplicação da técnica

[3] Mais informações em: https://www.unbound.io/
[4] Quase quatro mil seguidores em seu *Twitter* e mais de três mil membros em seu canal no *Discord*

em jogos digitais, serão implementadas algumas funcionalidades que seriam esperadas para o desenvolvimento desse tipo de software, como:

- Estrutura modular que permita a integração do renderizador em uma *Game Engine*;
- Programar o comportamento dos diferentes objetos que compõem a cena;
- Permitir a interação do usuário com a aplicação por meio do teclado.

1.3 Estrutura desta monografia

No Capítulo 2 serão explorados os conceitos importantes para o entendimento do trabalho. Serão explicadas em detalhe as diferentes técnicas de renderização citadas até agora e será explicado todo o conhecimento prévio necessário para discutir o método de *Ray Marching*.

No Capítulo 3 será explicado como foi desenvolvido o renderizador baseado em *Ray Marching*, entrando em detalhes das diferentes partes do *software* e como elas interagem entre si.

No Capítulo 4 os resultados serão apresentados e analisados, e, no Capítulo 5, será encerrado o trabalho.

Capítulo 2

Fundamentação teórica

Neste capítulo serão discutidos conceitos teóricos importantes para o entendimento adequado deste trabalho. Primeiro, serão definidas em detalhes as estratégias de renderização citadas no capítulo anterior. Isso será importante para entender as diferenças entre elas, e, mais importante, os benefícios que o uso de *Ray Marching* pode trazer. Em seguida, serão discutidos conceitos específicos da técnica de *Ray Marching*. Finalmente, será apresentada uma visão geral do algoritmo de *Ray Marching* por meio de uma implementação em pseudo código, e explicações detalhadas das etapas do processo.

2.1 Estratégias de renderização

Nesta seção serão explicadas em mais detalhes as diferentes estratégias de renderização apresentadas no capítulo anterior. Como este trabalho foca na aplicação do *Ray Marching* em jogos digitais 3D, este será o foco do capítulo. No entanto, a maioria dos conceitos apresentados podem ser generalizados para a renderização em 2D. Tanto é que muitas *Game Engines* modernas, como *Unity 3D* e *Unreal Engine*, utilizam um único renderizador 3D tanto para cenas tridimensionais quanto para cenas bidimensionais.

A função principal de qualquer renderizador é resolver o problema da visibilidade de objetos [23]. Esse problema consiste na determinação de quais regiões de uma superfície estão visíveis para um determinado observador, como mostrado na Figura 2.1. Um algoritmo de renderização pode ser entendido como uma caixa preta que, para cada *pixel* da tela, dada uma determinada cena 3D, calcula a cor que deve ser exibida naquele *pixel* baseado nos objetos que são visíveis de um determinado referencial. Os algoritmos de renderização diferem, principalmente, na forma como resolvem o problema da visibilidade [22]. Os tipos de renderizadores mais utilizados atualmente serão apresentados em detalhes a seguir.

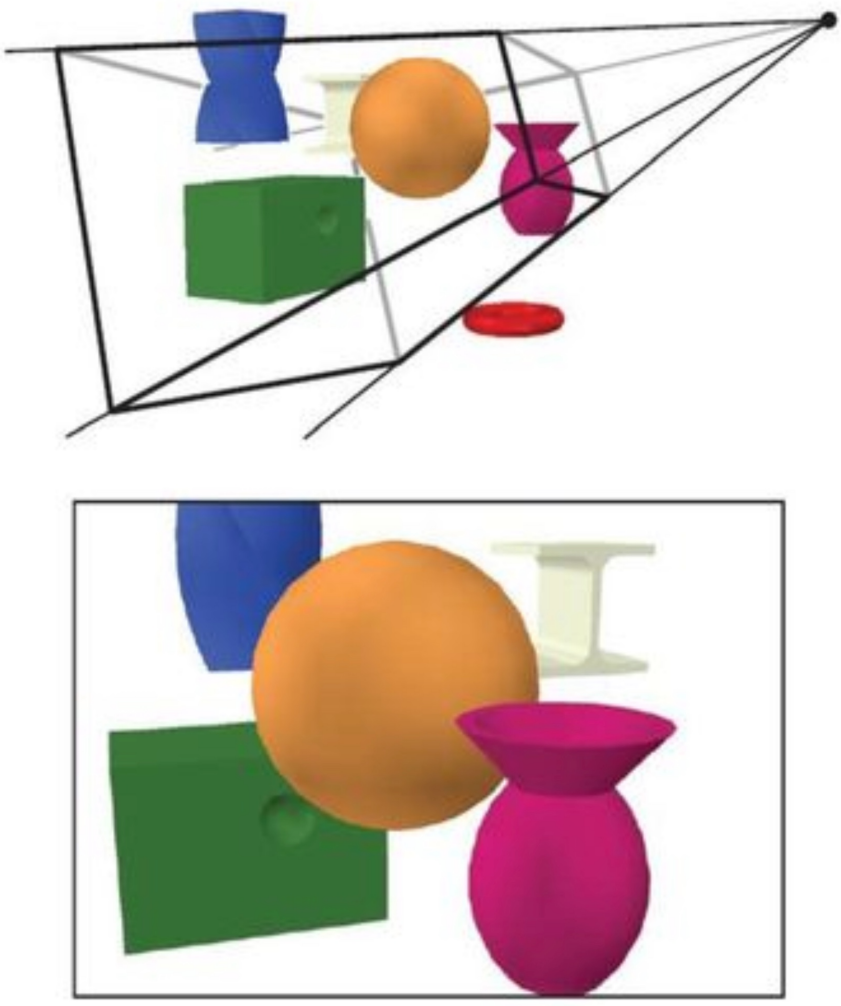

Figura 2.1: Visualização do problema da visibilidade de objetos.
Na figura de baixo, projeção da cena do ponto de vista do observador representado. Na figura de cima, visão alternativa da cena mostrando o campo de visão do observador. Fonte: Allan Ryan, 2018. Disponível: https://slideplayer.com/slide/12875358/

2.1.1 *Ray Tracing*

Ray Tracing faz parte do grupo de técnicas baseadas na simulação do caminho dos raios de luz. Estas técnicas, de forma geral, procuram resolver o problema da visibilidade simulando como a luz reflete nas superfícies do mundo real, e como os olhos humanos percebem e processam esses raios de luz que chegam até eles [25]. *Ray Tracing*, especificamente, utiliza fórmulas de intersecção e propriedades fotogramétricas dos objetos para calcular reflexões dos raios de luz na cena, até que eles cheguem à câmera [29].

No mundo real, os raios de luz saem das fontes de luz, refletem nas superfícies do ambiente, e chegam aos nossos olhos para formar as imagens. No entanto, esse processo é computacionalmente ineficiente, visto que a maioria dos raios de luz emitidos pela fonte

de luz não chegariam até a câmera da cena, e, portanto, não contribuiriam para a imagem final [25]. Felizmente, a propriedade de reversibilidade da propagação da luz permite que os raios de luz sejam emitidos diretamente da câmera, e que seu caminho seja traçado de forma reversa até as fontes de luz [25]. Os raios emitidos pela câmera que atingem uma fonte de luz estarão iluminados na imagem final. Os raios que não atingirem nenhuma fonte de luz não estarão iluminados na imagem final.

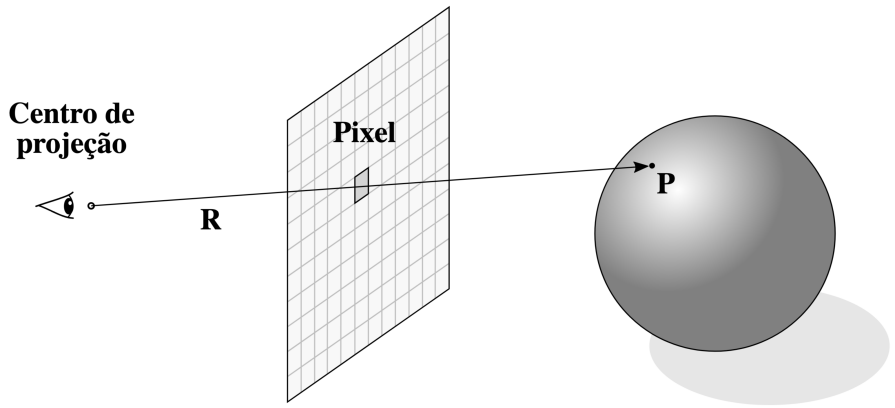

Figura 2.2: Ilustração da renderização de uma esfera por *Ray Tracing*. Para cada pixel, é emitido um raio da perspectiva do observador ao pixel para calcular a cor do pixel.
Adaptado de: Harlen Batagelo, Bruno Marques, 2021. Disponível:
https://www.brunodorta.com.br/cg/ray-casting-x-rasteriza%C3%A7%C3%A3o.html

Uma cena tridimensional pode, portanto, ser composta por diversas formas primitivas, cuja variedade está limitada pela possibilidade de calcular sua intersecção com um raio. A Figura 2.2 mostra a renderização de uma esfera, mas muitas outras formas primitivas podem ser implementadas, sendo o triângulo a mais utilizada em jogos digitais [1]. Usualmente, a geometria 3D utilizada em jogos digitais é composta por um conjunto de triângulos que formam o modelo final.

Para renderizar a cena por *Ray Tracing* o renderizador passa por alguns passos [7]: primeiro, são emitidos um ou mais raios da câmera passando por cada *pixel* da imagem final. A trajetória dos raios pode, então, ser calculada utilizando as funções de intersecção e refletindo o raio diversas vezes na cena até que ele atinja uma fonte de luz. A forma como os raios refletem pode ser alterada baseada nas propriedades do material da superfície que foi atingida. Por exemplo, para uma superfície espelhada, todos os raios que atingem aquela superfície devem ser refletidos perfeitamente [25]. Em contrapartida, para uma superfície opaca, os raios que atingem a superfície devem ser refletidos em direções aleatórias.

Como essa técnica simula a forma como raios de luz funcionam no mundo real, muitos efeitos complexos de iluminação podem ser simulados, como refração, difusão e Lei de Beer [28], como pode ser visto na Figura 2.3. No entanto, se trata de um algoritmo computacionalmente caro, pois, para cada iteração do laço, é necessária manutenção da cena inteira na memória [7], além de apresentar padrões imprevisíveis de acesso a memória, o que causa problemas de invalidação de cache [6]. Portanto, *Ray Tracing* é usualmente utilizado apenas para renderizar efeitos mais complexos de iluminação em jogos digitais, sendo o

restante da cena renderizada por rasterização, que é significativamente menos custoso.

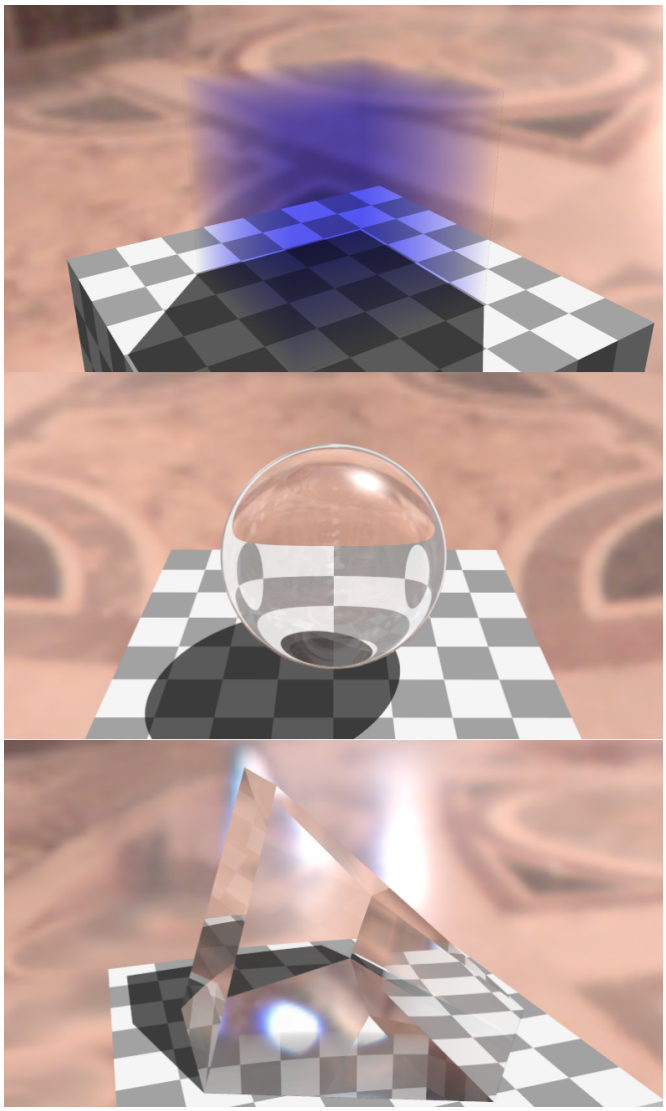

Figura 2.3: Exemplo de efeitos de iluminação possíveis com *Ray Tracing*. Lei de Beer, refração e reflexão total interna, respectivamente.
Fonte: Alan Wolfe, 2019. Disponível: https://blog.demofox.org/2017/01/09/raytracing-reflection-refraction-fresnel-total-internal-reflection-and-beers-law/

2.1.2 Rasterização

Enquanto o *Ray Tracing* opera em um laço que itera por cada *pixel* na imagem, a rasterização foca primariamente nas formas primitivas que compõem a cena [7]. Para renderizar uma cena tridimensional, o renderizador passa por três etapas [7]. Primeiro, é feita a projeção de todas as primitivas que compõem a cena sobre a imagem final. Isso inclui fazer comparações de profundidade entre as primitivas renderizadas para determinar quais estão atrás de outras primitivas. Essa etapa geralmente consiste na ordenação das primitivas da cena baseado em sua profundidade, e a projeção é feita de frente para trás para objetos opacos (para evitar renderizar objetos que estão completamente obstruídos de outros objetos, problema conhecido como *overdraw*), e de trás para frente para objetos translúcidos (pois a cor de um objeto transparente depende dos objetos que estão atrás dele) [2]. Embora seja possível calcular a projeção de diversas formas geométricas sobre um plano, o triângulo é a forma primitiva mais usualmente utilizada pelos renderizadores [1]. A etapa de projeção pode ser visualizada na Figura 2.4. Depois, é realizada a etapa de rasterização, que dá o nome para a técnica. Nesta etapa, é realizada a discretização da projeção das primitivas nos *pixels* da imagem; ou seja: o objetivo desta etapa é determinar quais *pixels* da imagem final fazem parte de cada objeto que foi projetado na etapa passada. Por fim, há a etapa de colorização. A etapa de rasterização pode ser visualizada na Figura 2.5. Sabendo quais *pixels* da imagem final pertencem a cada objeto diferente, é possível colorir os *pixels* baseado nas propriedades dos materiais de cada superfície e gerar a imagem final.

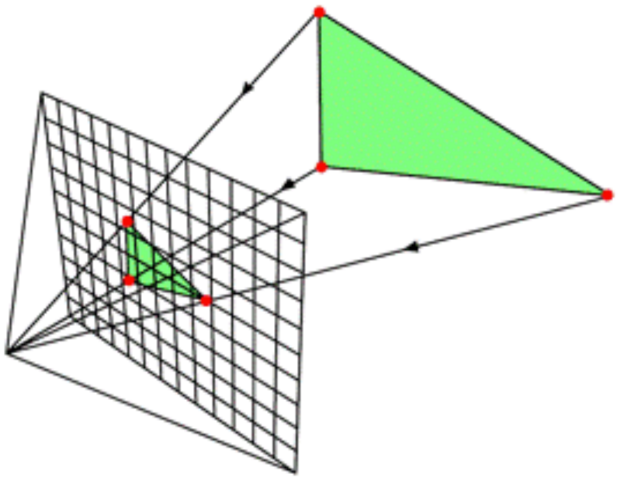

Figura 2.4: Ilustração da etapa de projeção de um triângulo na renderização por rasterização.
Fonte: Alexandre Ziebert, NVIDIA, 2020. Disponível:
https://www.nvidia.com/pt-br/drivers/prbr-05282018/

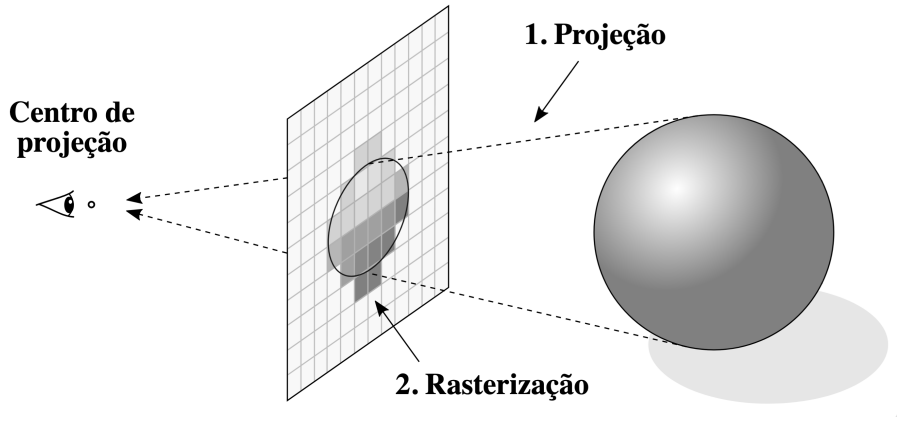

Figura 2.5: Ilustração da renderização de uma esfera por Rasterização.
Adaptado de: Harlen Batagelo, Bruno Marques, 2021. Disponível:
https://www.brunodorta.com.br/cg/ray-casting-x-rasteriza%C3%A7%C3%A3o.html

Como a técnica de rasterização não simula o comportamento da luz como o *Ray Tracing*, a implementação de efeitos de iluminação se torna muito mais complexa. Por exemplo, para renderizar sombras geralmente é utilizada a técnica de *Shadow Mapping* [10]. Essa técnica requer a renderização da sombra novamente pela perspectiva da fonte de luz para determinar quais partes da cena são iluminadas por aquela fonte, e quais estão em sombras. Isso é feito para cada fonte de luz na cena, e os resultados são agregados para renderizar as sombras na imagem principal. Os demais efeitos de iluminação demonstrados na Figura 2.3 são computacionalmente custosos e, muitas vezes, inviáveis para aplicações de tempo real [2].

Como mencionado anteriormente, a técnica de rasterização é computacionalmente menos custosa do que técnicas baseadas em raios, como *Ray Tracing* [6]. Diversos fatores influenciam essa diferença de complexidade. Um deles é o fato de que, na rasterização, cada iteração do laço requer manter apenas um objeto da cena na memória por vez [7]. Além disso, por ser uma técnica muito utilizada em diversas áreas da computação gráfica (não apenas em jogos digitais) [22], grande parte do *hardware* gráfico moderno conta com estruturas especializadas em acelerar aspectos específicos do processo de rasterização [6].

2.1.3 *Ray Marching*

A técnica de *Ray Marching* parte do mesmo princípio do *Ray Tracing*, visto que também utiliza raios propagados pela cena para resolver o problema da visibilidade [23]. A diferença entre os dois está na forma como os raios são propagados. Enquanto no *Ray Tracing* a colisão dos raios com a geometria da tela é diretamente calculada por equações de intersecção, *Ray Marching* utiliza um processo iterativo para propagar o raio [24] [4]. Isso significa que, para cada raio que sai da câmera, é iniciado um laço, e, para cada iteração desse laço, o raio avança uma certa distância em sua direção. Dessa forma, a técnica de *Ray Marching* engloba diversas técnicas diferentes que utilizam o mesmo princípio de propagação iterativa, mas diferem na foma como decidem a distância que o raio deve avançar em um determinado momento. Alguns exemplos de técnicas de *Ray Marching* são: marcha uniforme, *Sphere Tracing* e *Cube-assisted*. As duas primeiras serão exploradas em detalhes a seguir.

2.1.3.1 Marcha Uniforme

Esta variante de *Ray Marching* consiste em avançar cada raio uma distân- cia fixa a cada iteração do laço. Essa técnica é muito utilizada para renderização volumétrica, como nuvens e fumaça [13], e para visualização de dados médicos [27] de ressonância magnética, por exemplo.

Para renderizar volumes utilizando essa técnica, o renderizador passa por quatro passos [27]. Primeiro, assim como em *Ray Tracing*, um ou mais raios são emitidos saindo da câmera e passando por cada *pixel* da imagem final. Nessa etapa pode ser utilizado *Ray Tracing* para determinar o ponto de entrada e de saída do raio com a área do volume que se deseja renderizar (geralmente delimitado por uma forma geométrica simples como um cuboide ou um elipsoide). Isso é feito para que o raio não precise marchar fora do espaço delimitado pelo volume, permitindo utilizar passos menores em um tempo razoável. Ao determinar que o raio está dentro do volume, se inicia a etapa de *Ray Marching*, em que o raio avança uma distância constante a cada iteração. Toda vez que avançar, é feita uma leitura do valor do volume naquela posição. Esse passo é repetido até que o raio saia do volume. Depois, é feita a colorização de cada uma das leituras baseada no material desejado para o volume. Por fim, é feita a composição de todas as leituras feitas pelo raio para determinar a cor do *pixel* na imagem final. Os passos descritos podem ser observados na Figura 2.6

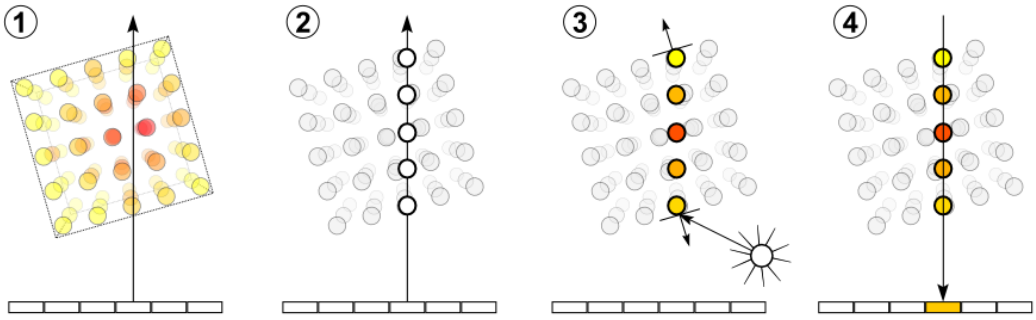

Figura 2.6: Diagrama dos passos para renderizar um volume utilizando *Ray Marching*. Na primeira imagem, um raio é emitido e sua colisão é calculada com o volume cúbico. Na segunda imagem, ocorre a marcha uniforme dentro do volume. Na terceira imagem, a cor de cada ponto do volume é calculada baseada nas propriedades do volume e da iluminação. Na quarta imagem, os dados de todos os passos de um mesmo raio são agregados para calcular a cor final de um pixel.
Fonte: Florian Hofmann, 2011. Disponível:
https://commons.wikimedia.org/wiki/File:Volume_ray_casting.png

No entanto, essa técnica é pouco eficiente para renderização de cenas completas de geometria sólida [4]. Caso seja escolhida uma constante muito pequena para o passo dos raios, cada raio demoraria muito para atravessar a cena inteira. Em contrapartida, se for escolhido um passo muito grande, o raio pode pular completamente pedaços mais finos da geometria. Para que esse tipo de renderização seja viável utilizando *Ray Marching* é necessária uma técnica mais sofisticada para escolher a distância que cada raio vai avançar em uma determinada iteração. Idealmente, seria utilizada uma técnica que escolha passos grandes quando o raio estiver em uma parte vazia da cena, e passos menores quando ele

estiver próximo de alguma geometria para otimizar o tempo de renderização e a qualidade de imagem final. Para isso, é possível utilizar a técnica de *Sphere Tracing* com estimadores de distância.

2.1.3.2 *Sphere Tracing*

Como mencionado anteriormente, *Sphere Tracing* se trata de um método específico de *Ray Marching* que utiliza um algoritmo mais sofisticado para decidir quanto um raio deve avançar em uma determinada iteração. De acordo com John C. Hart:

> "*Sphere Tracing* utiliza funções que retornam a distância a uma superfície implícita para definir uma sequência de pontos que converge linearmente para a primeira intersecção entre o raio e um superfície." - John C. Hart (1996) [15] (tradução própria) [1]

Em outras palavras, para decidir o quanto um raio deve avançar em uma determinada iteração, *Ray Marching* utiliza funções de distância (que serão discutidas em mais detalhes na seção 2.2) para descobrir o quão distante a posição atual do raio está da geometria da cena. Essa medida de distância não se importa com a direção do raio, apenas com a posição atual dele. Essa distância pode ser visualizada ao pensar em uma esfera com centro na posição atual do raio. O diâmetro da esfera expande continuamente até que ela encoste em alguma superfície da geometria da cena. Não se sabe onde na cena essa colisão ocorreu. O raio pode, então, avançar a medida do raio da esfera sem colidir com nenhum objeto da cena. Esse processo é iterativamente repetido para cada raio, e, quando a distância retornada pela função de distância for zero, ou muito próxima de zero, sabemos que o raio colidiu com algum objeto da cena. Dessa forma, quando o raio está muito distante de qualquer geometria da cena ele poderá avançar mais, e, quando estiver perto de algum objeto, irá avançar mais devagar para garantir que não atravessará nenhuma parte da geometria.

[1] Texto origianl: "Sphere tracing capitalizes on functions that return the distance to their implicit surfaces to define a sequence of points that converges linearly to the first ray-surface intersection."

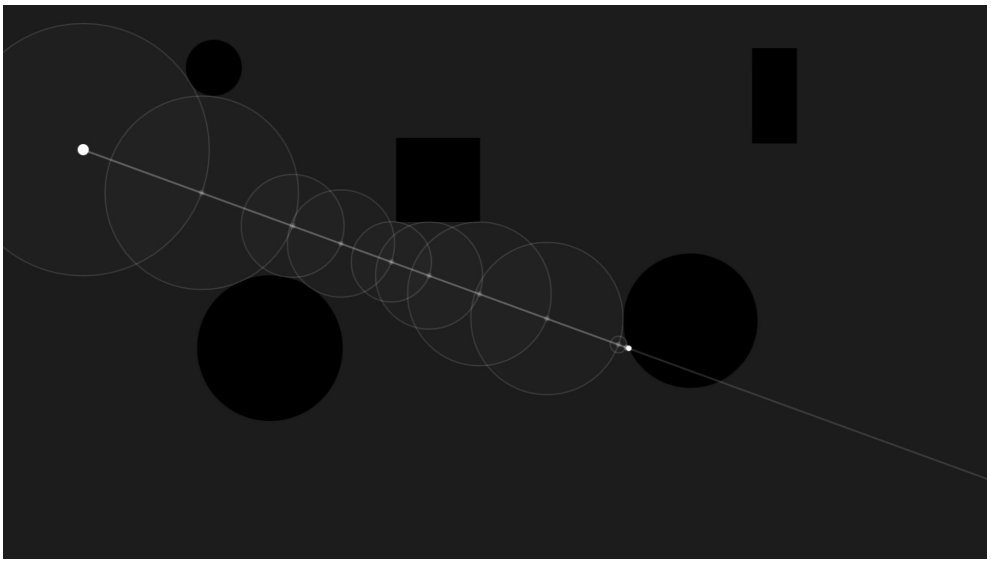

Figura 2.7: Visualização em 2D do algoritmo de *Sphere Tracing* para um único raio.
Fonte: Sebastian Lague, 2019. Disponível: https://www.youtube.com/watch?v=Cp5WWtMoeKg

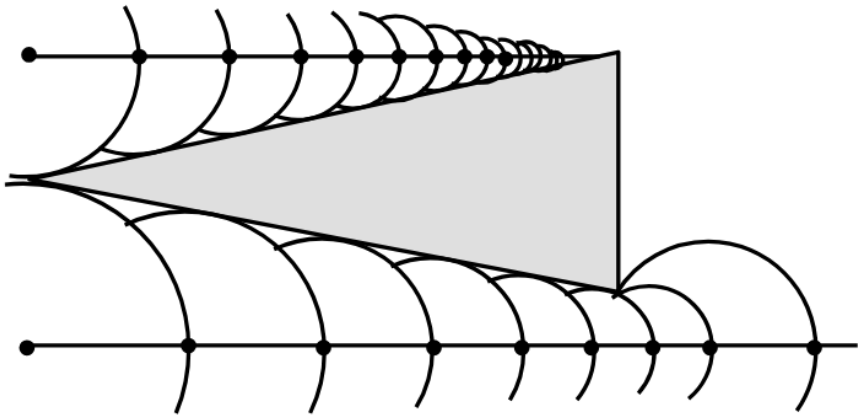

Figura 2.8: Exemplos de raios que passam muito próximos da geometria da cena.
Fonte: John C. Hart, 1996 [15]

Na Figura 2.7 está representado o caminho de um único raio. Cada circunferência representa um passo da iteração do *Sphere Tracing* e o raio de cada circunferência é igual à distância retornada pela função de distância. Há algumas situações em que um raio é obrigado a fazer passos extremamente pequenos, mesmo que não vá colidir com nenhuma geometria. Isso ocorre, em geral, quando um raio passa muito perto de alguma superfície, como está representado na Figura 2.8. Podemos visualizar o número de iterações que cada raio precisou fazer para colidir com a cena na imagem final. Na Figura 2.9 está ilustrada uma comparação entre uma imagem renderizada utilizando *Sphere Tracing* e a visualização do número de iterações que cada *pixel* levou para ser renderizado. Quanto mais branco o *pixel*, mais iterações foram necessárias. É possível ver como os raios que mais demoram a convergir são aqueles que passam perto da geometria mas não colidem nela. Existem algumas

estratégias para mitigar esses casos, mas sua explicação e implementação estão fora do escopo deste trabalho.

Figura 2.9: Visualização do número de passos necessários para cada *pixel* ser renderizado. Adaptado de: Inigo Quilez, 2015. Disponível: https://www.shadertoy.com/view/ld3Gz2

Por seguir o mesmo princípio básico da técnica de *Ray Tracing* os mesmos efeitos complexos de iluminação podem ser simulados utilizando *Ray Marching*, como os mostrados na Figura 2.3. Além disso, embora *Sphere Tracing* possa ser utilizado para renderizar modelos 3D convencionais construídos a partir de diversos polígonos (como os comumente utilizados em rasterizadores), sua principal aplicação é na renderização de superfícies implícitas [15], incluindo formas geométricas complexas como fractais [14][4]. Essas superfícies são descritas pelas funções de distância citadas anteriormente, e são as formas primitivas do *Sphere Tracing*. Cenas mais complexas, como a mostrada na Figura 2.9, podem ser modeladas utilizando combinações das formas primitivas, por meio de operações unárias e operações binárias, que serão discutidas em mais detalhes na Seção 2.4. Por fim, o resultado das funções de distância podem ser ainda mais modificadas utilizando funções de manipulação de domínio, que serão discutidas em mais detalhes na Seção 2.3.

2.2 Funções de distância

Formalmente, John C. Hart [15] define funções de distância da seguinte forma:

Seja uma função f um mapeamento contínuo de $f : \mathbb{R}^n \to \mathbb{R}$ que descreve o conjunto $A \subset \mathbb{R}^n$ como o conjunto de pontos $x \in \mathbb{R}^n$ tal que

$$A = \{x : f(x) \leq 0\}$$

Por continuidade, f é igual a zero na fronteira δA que representa a superfície implícita de f. Ademais, espera-se que f seja estritamente negativa no interior $\overset{\circ}{A}$, que permite que a função multivalorada f^{-1} represente, no ponto $f^{-1}(0)$, a superfície implícita de f.

Definição 1 A distância[2] entre um ponto $x \in \mathbb{R}^3$ e um conjunto $B \subset \mathbb{R}^3$ é definida como a distância entre x e o ponto mais próximo de B:

$$d(x, B) = \min_{y \in B} \parallel x - y \parallel$$

Dado um conjunto B, a distância ponto-conjunto $d(x, B)$ define implicitamente B [16], visto que $B = \{x : d(x, B) = 0\}$. Aqui, nós estamos interessados no inverso: dado uma função implícita, qual a distância ponto-conjunto para a sua superfície?

Definição 2 Uma função $f : \mathbb{R}^3 \to \mathbb{R}$ é um limite de distância[3] com sinal da sua superfície implícita $f^{-1}(0)$ se e somente se

$$\mid f(x) \mid \leq d(x, f^{-1}(0))$$

No caso específico em que $\mid f(x) \mid = d(x, f^{-1}(0))$ chamamos f de função de distância com sinal (FDS).

Assim, os pontos em que $f(x) < 0$ serão definidos como o interior da superfície e os pontos em que $f(x) > 0$ serão definidos como exterior da superfície.

Alguns exemplos de superfícies tridimensionais junto com as FDSs que as definem implicitamente são [18]:

1. Esfera de raio $r \in \mathbb{R}$:
$$f(x) = \parallel x \parallel - r$$
2. Plano de normal $n \in \mathbb{R}^3$ e altura $h \in \mathbb{R}$:
$$f(x) = x \cdot n + h$$

[2] Aqui utilizamos distância Euclidiana, mas outros tipos de cálculo de distância poderiam ser utilizados no lugar, como a distância de Manhattan.

[3] Limites de distância possuem aplicações importantes em *Sphere Tracing* pois, mesmo que não seja possível calcular uma função de distância com sinal exata, com um limite de distância ainda é possível garantir que o renderizador não vai marchar mais do que deveria, e será possível renderizar a superfície implícita, ainda que tomando mais passos do que seriam necessários.

Outros exemplos de FDSs de primitivas mais complexas são melhor visualizados na forma de código em GLSL (linguagem de *shaders* que será utilizada para exemplificar FDSs neste capítulo) no Algoritmo 1. Também podemos visualizar as funções de distância de algumas formas primitivas em 2D na Figura 2.10. Nela, áreas verdes representam pontos onde $f(x) > 0$, áreas azuis representam pontos onde $f(x) < 0$ e contornos brancos representam pontos onde $f(x) = 0$.

Algoritmo 1 Implementação em GLSL das FDSs de um cuboide e de um cilindro [18].

```
1  // ‘b‘ representa o tamanho do cuboide em cada uma das 3 dimensões.
2  float cuboide(vec3 x, vec3 b) {
3    vec3 q = abs(x) - b;
4    return length(max(q,0.0)) + min(max(q.x,max(q.y,q.z)),0.0);
5  }
6
7  // ‘h‘ e ‘r‘ representam a altura e raio do cilindro, respectivamente.
8  float cilindro(vec3 x, float h, float r) {
9    vec2 d = abs(vec2(length(x.xz),x.y)) - vec2(r,h);
10   return min(max(d.x,d.y),0.0) + length(max(d,0.0));
11 }
```

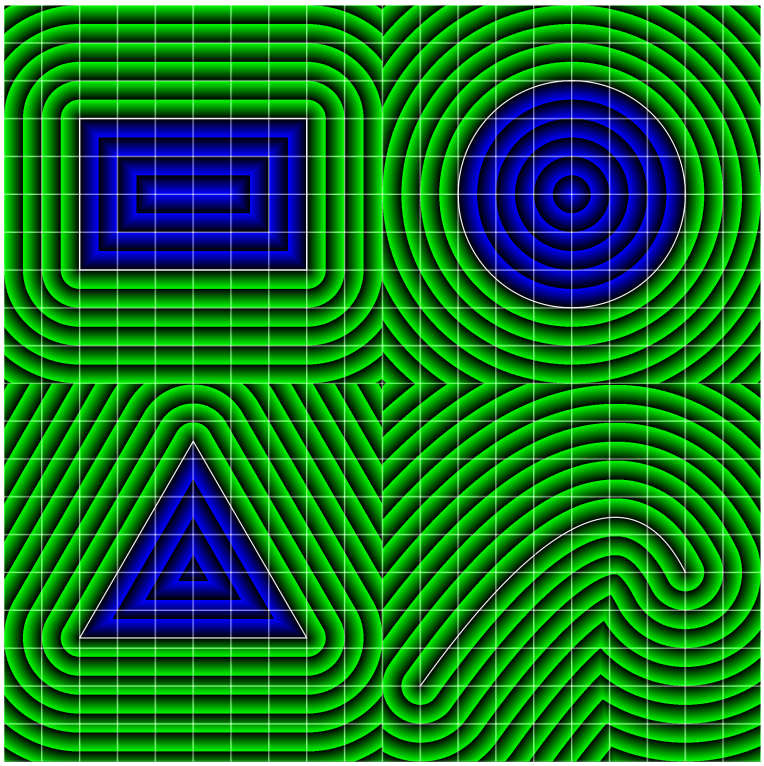

Figura 2.10: Visualização das FDSs em 2D de um círculo, um paralelogramo reto, um triângulo equilátero e uma curva Bézier quadrática. Áreas verdes representam pontos onde $f(x) > 0$, áreas azuis representam pontos onde $f(x) < 0$ e contornos brancos representam pontos onde $f(x) = 0$.

No Algoritmo 1, `b` representa o tamanho do cuboide em cada uma das 3 dimensões,

`h` e `r` representam a altura e raio do cilindro, respectivamente.

2.3 Manipulações de domínio

Manipulações de domínio são operações que podem ser aplicadas no domínio das FDSs para alterar o resultado final [18]. Como são aplicadas sobre o domínio das FDSs, as manipulações de domínio têm sempre como resultado alguma alteração no espaço (neste trabalho, consideraremos o espaço $\mathbb{R}^3$ na maior parte do tempo). Os exemplos mais utilizados dessas operações são repetições e distorções de domínio.

As operações de repetição de domínio tem como objetivo repetir as FDSs ao longo do espaço. Quando aplicadas para renderização de cenas com *Ray Marching* e *Sphere Tracing*, podem ser utilizadas para repetir infinitamente [4] ou de forma limitada algum objeto, sem calcular múltiplas vezes a FDS do mesmo objeto, como pode ser visto nas Figuras 2.11, 2.12 e 2.13.

Algoritmo 2 Implementação em GLSL das operações de repetição infinita e repetição limitada em 2D[18].

```
1 // ‘s‘ representa o número de unidades no espaço que separam cada repetiçã
     o.
2 vec2 repetição(vec2 x, vec2 s) {
3     return uv - s * floor(uv / s + 0.5);
4 }
5
6 // ‘l‘ representa o número de repetições em cada dimensão.
7 vec2 repetição_limitada(vec2 x, float s, vec2 l) {
8     return uv - s * clamp(floor(uv / s + 0.5), -l, l);
9 }
```

No Algoritmo 2, `s` representa o número de unidades no espaço que separam cada repetição e `l` representa o número de repetições em cada dimensão.

[4] Nesse caso, ficando restritas à resolução da imagem renderizada e às configurações do renderizador, como distância máxima e número máximo de passos

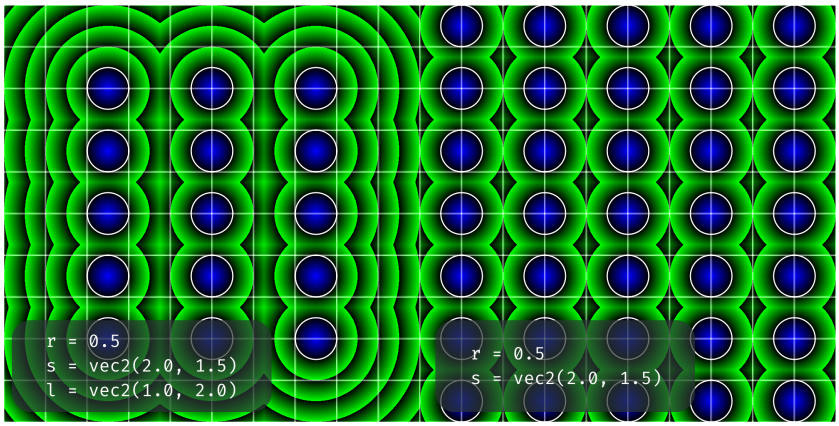

Figura 2.11: Visualização das operações de repetição limitada e repetição infinita, respectivamente, aplicadas sobre a FDS de um único círculo de raio `r`. Áreas verdes representam pontos onde $f(x) > 0$, áreas azuis representam pontos onde $f(x) < 0$ e contornos brancos representam pontos onde $f(x) = 0$.

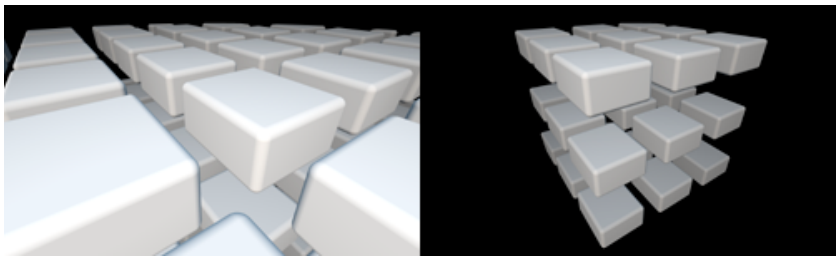

Figura 2.12: Exemplo em 3D do uso de operações de repetição infinita e limitada, respectivamente, na FDS de um único cuboide arredondado.
Fonte: Inigo Quilez, 2015. Disponível: https://iquilezles.org/articles/distfunctions/

Figura 2.13: Exemplo em 3D do uso operações de repetição limitada (5 x 9) para modelar as colunas do templo.
Fonte: Inigo Quilez, 2013. Disponível: https://iquilezles.org/articles/sdfrepetition/

Já as operações de distorção de domínio podem ser utilizadas para modificar a FDS de um objeto para deformá-lo de alguma maneira. Alguns exemplos de operações possíveis

são as de deslocamento, torção e dobra, como pode ser visto nas Figuras 2.15 e 2.14.

Algoritmo 3 Implementação em GLSL das operações de deslocamento, torção e dobra em 3D [18].

```
1  // ‘primitiva‘ representa a FDS de alguma forma primitiva.
2  float deslocamento(sdf3d primitiva, vec3 x) {
3      float d1 = primitiva(x);
4      float d2 = sin(20.0 * x.x) * sin(20.0 * x.y) * sin(20 * x.z);
5      return d1+d2;
6  }
7
8  // ‘primitiva‘ representa a FDS de alguma forma primitiva e ‘k‘ representa
       a intensidade da torção.
9  float torção(sdf3d primitiva, vec3 x, float k) {
10     float c = cos(k * x.y);
11     float s = sin(k * x.y);
12     mat2  m = mat2(c, -s, s, c);
13     vec3  q = vec3(m * x.xz, x.y);
14     return primitiva(q);
15 }
16
17 // ‘primitiva‘ representa a FDS de alguma forma primitiva e ‘k‘ representa
       a intensidade da dobra.
18 float dobra(sdf3d primitiva, vec3 x, float k) {
19     float c = cos(k * x.x);
20     float s = sin(k * x.x);
21     mat2  m = mat2(c, -s, s, c);
22     vec3  q = vec3(m * x.xy, x.z);
23     return primitiva(q);
24 }
```

No Algoritmo 3, `primitiva` representa a FDS de alguma primitiva qualquer e `k` representa a intensidade de cada uma das operações.

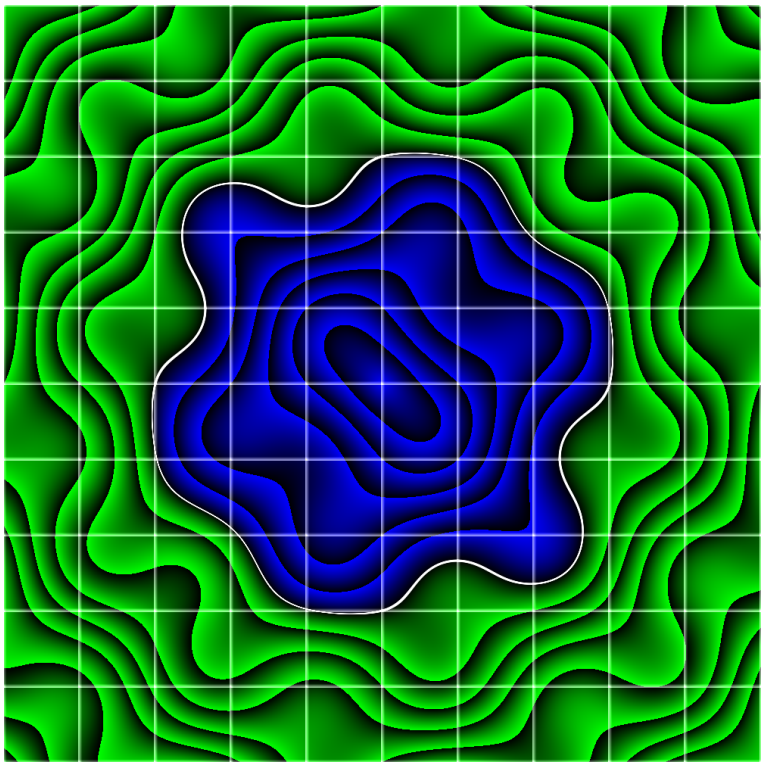

Figura 2.14: Visualização da operação de deslocamento aplicada sobre a FDS de um círculo. Áreas verdes representam pontos onde $f(x) > 0$, áreas azuis representam pontos onde $f(x) < 0$ e contornos brancos representam pontos onde $f(x) = 0$.

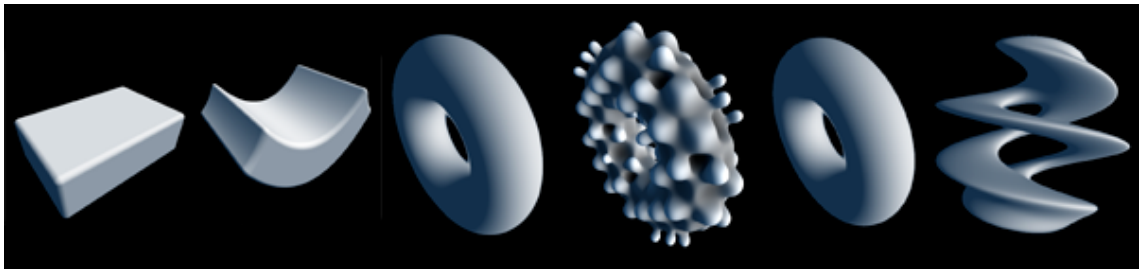

Figura 2.15: Exemplo em 3D do uso de operações de dobra sobre a FDS de um cuboide, deslocamento sobre a FDS de um toróide e torção sobre a FDS de um toróide respectivamente.
Fonte: Inigo Quilez, 2015. Disponível: https://iquilezles.org/articles/distfunctions/

Embora as deformações sejam um artefato muito útil na criação de novas formas, elas podem deformar o espaço de forma a gerar artefatos visuais na renderização. Esse problema ocorre quando as deformações transformam o domínio da FDS em um espaço de distâncias não linear. Isso pode ser observado na Figura 2.14, já que as faixas não estão mais uniformemente distribuídas no espaço (diferente das outras figuras, como a 2.10).

2.4 Operações unárias e binárias

As operações unárias e binárias são operações que podem ser aplicadas sobre a imagem das FDSs para modificar seu resultado, ou, ainda, compor uma cena. Elas são comumente utilizadas para criar FDSs de novas superfícies a partir de FDSs de superfícies já conhecidas.

As operações unárias permitem modificar o resultado de uma única FDS para gerar uma nova superfície. As operações mais utilizadas são as de revolução, extrusão, alongamento, arredondamento e casca.

Revolução e extrusão podem ser utilizadas para criar FDSs 3D a partir de FDSs 2D, como pode ser visto na Figura 2.16.

Algoritmo 4 Implementação em GLSL das operações de revolução e extrusão em 3D [18].

```
1  // ‘primitiva‘ representa a FDS de alguma forma primitiva 2D e ‘o‘
       representa o raio da revolução.
2  float revolução(vec3 x, sdf2d primitiva, float o) {
3      vec2 q = vec2(length(x.xz) - o, x.y );
4      return primitiva(q);
5  }
6
7  // ‘primitiva‘ representa a FDS de alguma forma primitiva 2D e ‘h‘
       representa o comprimento da extrusão.
8  float extrusão(vec3 x, sdf2d primitiva, float h) {
9      float d = primitiva(x.xy)
10     vec2 w = vec2(d, abs(x.z) - h);
11     return min(max(w.x, w.y), 0.0) + length(max(w, 0.0));
12 }
```

No Algoritmo 4, `primitiva` representa a FDS de uma primitiva 2D qualquer, `o` representa o raio da revolução e `h` representa o comprimento da extrusão.

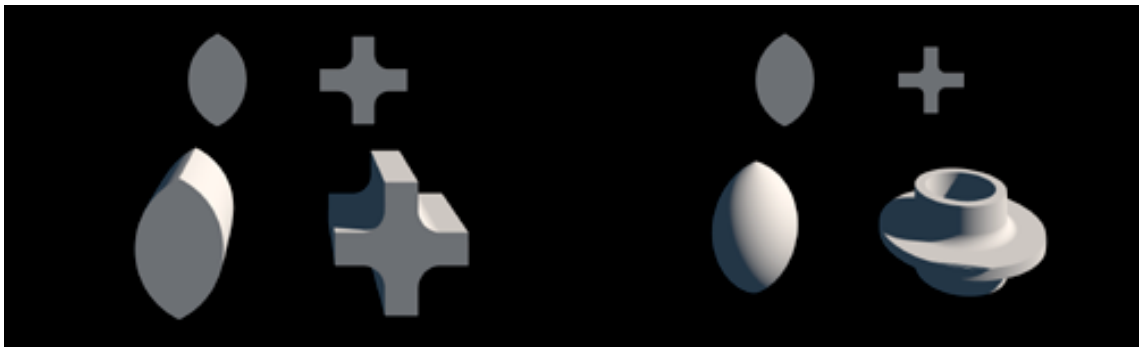

Figura 2.16: Exemplo em 3D do uso de operações unárias de extrusão (à esquerda) e revolução (à direita) sobre a FDS de uma vesica e de uma cruz. As FDSs originais estão atrás do resultado da aplicação das operações.
Fonte: Inigo Quilez, 2015. Disponível: https://iquilezles.org/articles/distfunctions/

Alongamento, arredondamento e casca, por sua vez, podem ser utilizadas para criar novas FDSs 3D a partir de outras FDSs 3D já conhecidas, como pode ser visto na Figura 2.17.

Algoritmo 5 Implementação em GLSL das operações de revolução e extrusão em 3D [18].

```
1  // ‘primitiva‘ representa a FDS de alguma forma primitiva e ‘h‘ representa
       a comprimento do alongamento em cada dimensão.
2  float alongamento(sdf3d primitiva, vec3 x, vec3 h){
3      vec3 q = x - clamp(x, -h, h);
4      return primitiva(q);
5  }
6
7  // ‘primitiva‘ representa a FDS de alguma forma primitiva e ‘r‘ representa
       o raio do arredondamento.
8  float arredondamento(sdf3d primitiva, vec3 x, float r) {
9      return primitiva(x) - rad;
10 }
11
12 // ‘primitiva‘ representa a FDS de alguma forma primitiva e ‘t‘ representa
       a espessura da casca.
13 float casca(sdf3d primitiva, vec3 x, float t) {
14     return abs(primitiva(x)) - t;
15 }
```

No Algoritmo 5, `primitiva` representa a FDS de uma primitiva 3D qualquer, `h` representa o comprimento do alongamento em cada dimensão, `r` representa o raio do arredondamento e `t` representa a grossura da casca.

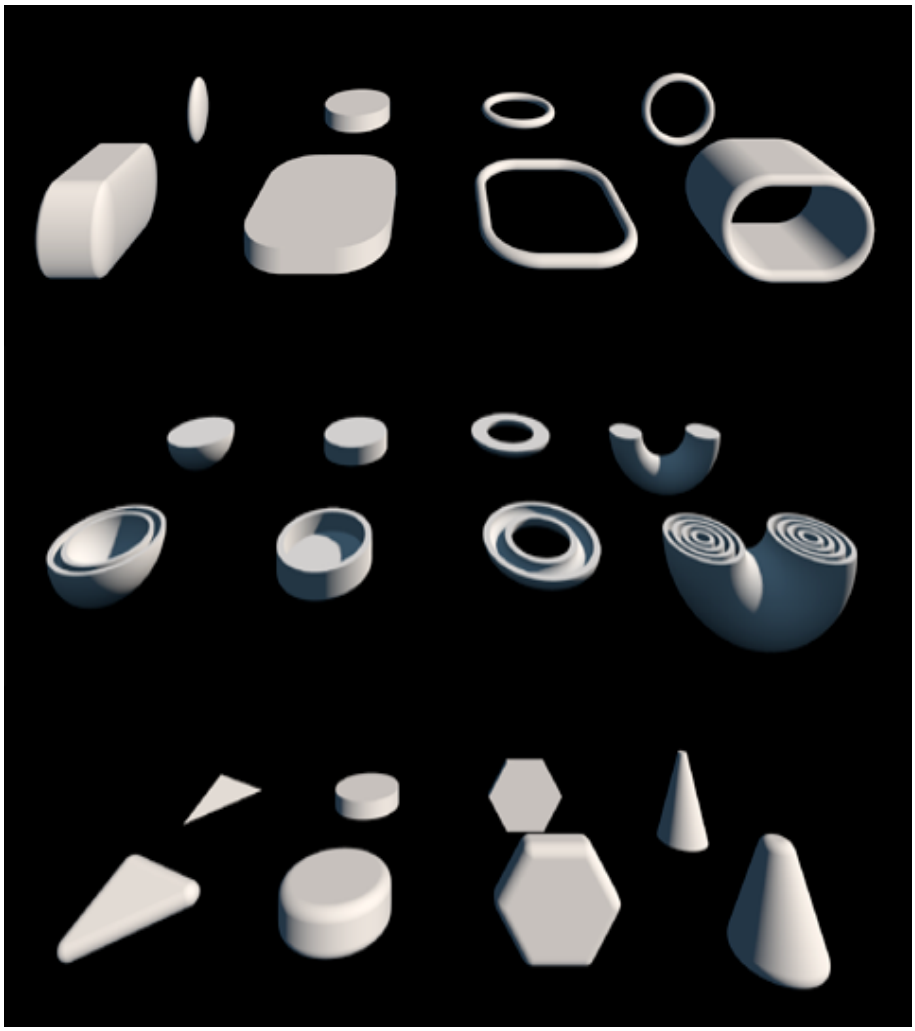

Figura 2.17: Exemplo em 3D do uso de operações unárias de alongamento (sobre as FDSs de um elipsoide, de um cilindro e de uma circunferência), casca (sobre as FDSs de um hemisfério esférico, de um cilindro, de um toróide plano e de um semi toróide) e arredondamento (sobre as FDSs de um triângulo, de um cilindro, de um hexágono e de uma pirâmide de base circular). As FDSs originais estão atrás do resultado da aplicação das operações.
Fonte: Inigo Quilez, 2015. Disponível: https://iquilezles.org/articles/distfunctions/

Já as operações binárias permitem combinar duas FDSs de formas diferentes para criar novas superfícies implícitas. As operações binárias mais comumente utilizadas são união, subtração e intersecção, como pode ser visto nas Figuras 2.18 e 2.19.

Algoritmo 6 Implementação em GLSL das operações binárias de união, subtração e intersecção em 3D [18].

```
1 // ‘d1‘ e ‘d2‘ representam os resultados das FDSs das duas superfícies
    implícitas sobre as quais se deseja aplicar a operação.
2
3 float uniao(float d1, float d2) {
4     return min(d1,d2);
5 }
6
7 float subtracao(float d1, float d2) {
8     return max(-d1,d2);
9 }
10
11 float interseccao(float d1, float d2) {
12     return max(d1,d2);
13 }
```

No Algoritmo 6, `d1` e `d2` representam os resultados das FDSs das duas superfícies implícitas sobre as quais se deseja aplicar a operação.

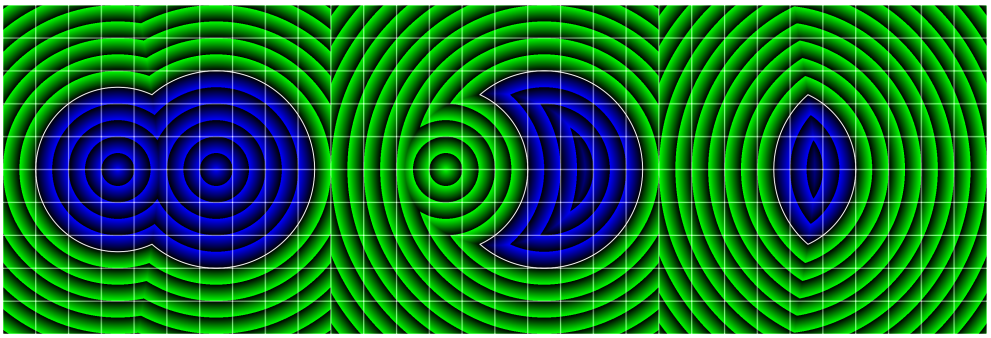

Figura 2.18: Visualização das operação binárias de união, subtração e intersecção, respectivamente, todas sobre a FDSs de dois círculos. Áreas verdes representam pontos onde $f(x) > 0$, áreas azuis representam pontos onde $f(x) < 0$ e contornos brancos representam pontos onde $f(x) = 0$.

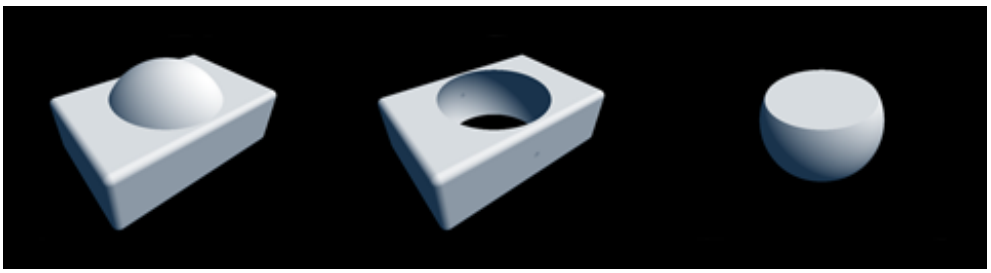

Figura 2.19: Exemplo em 3D do uso de operações binárias de união, subtração e intersecção, respectivamente, todas sobre a FDS de uma esfera e de um cuboide arredondado.
Fonte: Inigo Quilez, 2015. Disponível: https://iquilezles.org/articles/distfunctions/

Por utilizarem as funções de mínimo e máximo, as operações binárias podem ser adaptadas ao modificar o comportamento destas funções [18] [24]. Uma variante das operações

binárias duras (como serão referidas de agora em diante) são as operações binárias suaves, que podem ser vistas nas Figuras 2.21 e 2.20.

Algoritmo 7 Implementação em GLSL das operações binárias suaves de união, subtração e intersecção em 3D [18].

```
// ‘d1‘ e ‘d2‘ representam os resultados das FDSs das duas superfícies
    implícitas sobre as quais se deseja aplicar a operação e ‘k‘ representa
     o fator de suavização da operação.

float uniao_suave(float d1, float d2, float k) {
    float h = clamp(0.5 + 0.5 * (d2 - d1) / k, 0.0, 1.0);
    return mix(d2, d1, h) - k * h * (1.0 - h);
}

float subtracao_suave(float d1, float d2, float k) {
    float h = clamp(0.5 - 0.5 * (d2 + d1) / k, 0.0, 1.0);
    return mix(d2, -d1, h) + k * h * (1.0 - h);
}

float interseccao_suave(float d1, float d2, float k) {
    float h = clamp(0.5 - 0.5 * (d2 - d1) / k, 0.0, 1.0);
    return mix(d2, d1, h) + k * h * (1.0 - h);
}
```

No Algoritmo 7, `d1` e `d2` representam os resultados das FDSs das duas superfícies implícitas sobre as quais se deseja aplicar a operação e `k` representa o fator de suavização da operação.

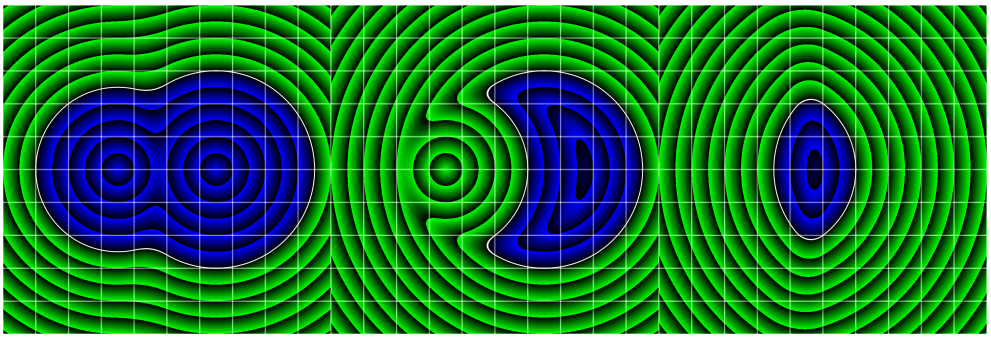

Figura 2.20: Visualização das operação binárias suaves de união, subtração e intersecção, respectivamente, todas sobre a FDS de dois círculos. Áreas verdes representam pontos onde $f(x) > 0$, áreas azuis representam pontos onde $f(x) < 0$ e contornos brancos representam pontos onde $f(x) = 0$.

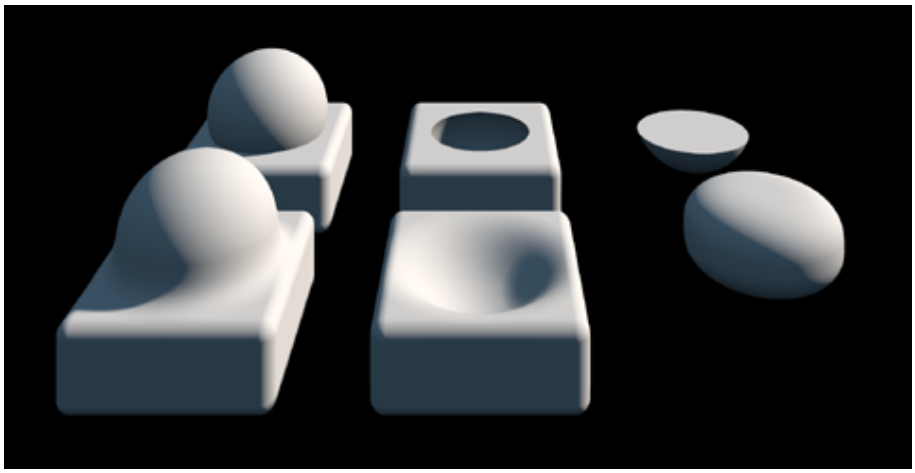

Figura 2.21: Exemplo em 3D do uso de operações binárias suaves de união, subtração e intersecção, respectivamente, todas sobre a FDS de uma esfera e de um cuboide.
Fonte: Inigo Quilez, 2015. Disponível: https://iquilezles.org/articles/distfunctions/

2.5 Visão geral do algoritmo de *Sphere Tracing*

De forma simplificada, o algoritmo de *Ray Marching* utilizando *Sphere Tracing* funciona como descrito no Algoritmo 8.

Algoritmo 8 Visão geral do algoritmo de renderização utilizando *Ray Marching* com *Sphere Tracing*.

```
1:  DISTANCIA_MAX ← Distância máxima que um raio pode viajar
2:  PASSOS_MAX ← Número máximo de passos que um raio pode dar
3:  DIST_SUPERF ← Distância mínima para considerar que o raio colidiu com uma superfície
4:
5:  ro ← Origem do raio                                   ▷ Posição da câmera
6:  while Pixels para serem renderizados do               ▷ Renderiza todos os pixels da tela
7:      rd ← Direção do raio                              ▷ Direção calculada com base no pixel atual
8:      dist ← 0                                          ▷ Distância que o raio viajou
9:      passos ← 0                                        ▷ Contador para o número de passos do raio
10:     while passos < PASSOS_MAX  do                     ▷ Enquanto o raio ainda pode dar passos
11:         pos ← ro + rd * d                             ▷ Posição atual do raio
12:         fds ← calcula_fds(pos)                        ▷ FDS da cena na posição atual do raio
13:         dist ← dist + fds                             ▷ Raio viaja a distância dada pela função
14:
15:         if fds < DIST_SUPERF then                     ▷ Se o raio atingiu alguma superfície
16:             calcular_iluminacao()
17:             colorir_pixel()
18:         else if dist > DISTANCIA_MAX then             ▷ Se o raio viajou a distância máxima
19:             colorir_fundo()
20:         end if
21:     end while
22: end while
```

2.6 *Shaders*

Shaders são um tipo específico de programa de computador. Enquanto a grande maioria dos programas utilizados diariamente são executados na CPU, *shaders* são programas escritos para serem executados pela GPU do computador. GPU é a sigla em inglês para *Graphics Processing Unit*, ou Unidade de Processamento Gráfico. O termo foi primeiro usado pela empresa NVIDIA, pioneira no mercado de GPUs, no ano de 1999 para se referir ao seu produto GeForce 256 [2].

CPUs e GPUs diferem na quantidade e magnitude das operações que podem ser realizadas por elas. A CPU é arquitetada para lidar com uma grande variedade de operações complexas, podendo incluir ramificações, estruturas condicionais e laços longos, mas executadas em menor quantidade. Em contrapartida, a GPU é arquitetada para lidar com uma grande variedade de operações mais simples, não sendo capaz de lidar tão bem com operações condicionais e laços de execução (no geral, GPUs não lidam muito bem com programas ramificados [27]), mas sendo capaz de executá-las em maior quantidade simultaneamente. Essa diferença pode ser vista nas especificações de CPUs e GPUs modernas no ano de 2024. Enquanto uma CPU de topo de linha como o AMD Ryzen 7 7950X3D possui 16 núcleos de processamento[5], uma GPU de topo de linha como a NVIDIA RTX 4090 possui 16384 núcleos de processamento[6]. Naturalmente, cada núcleo de processamento individual da GPU é muito menos capaz que um núcleo da CPU. Assim, é possível perceber que a GPU possui uma capacidade muito maior de processar grandes quantidades de dados em paralelo, propriedade importante para colorir os *pixels* da tela, por exemplo.

Dessa forma, enquanto podemos utilizar linguagens como C e Rust para escrever *scripts* para programar a CPU, podemos utilizar linguagens como GLSL, HLSL e Slang para escrever *shaders* e programar a GPU. Além disso, os *shaders* sempre são executados sobre um conjunto de elementos de entrada (vértices de um modelo 3D ou pixels da tela, por exemplo), executando, em paralelo, o mesmo procedimento sobre todos os elementos. Os tipos distintos de *shaders* diferem no domínio sobre o qual são executados. Alguns dos tipos mais utilizados na indústria são:

- *Geometry Shaders*: são executados para cada primitiva que se deseja renderizar, podendo excluir ou criar novas primitivas programaticamente;
- *Vertex Shaders*: são executados para cada vértice de uma malha 3D ou cena e são capazes apenas de modificar as propriedades dos vértices existentes, sendo incapazes de excluir ou criar novos vértices;
- *Fragment Shaders*: são executados para cada pixel que se deseja colorir e apenas retornam como resultado a cor final de cada pixel;
- *Compute Shaders*: podem ser executados sobre uma entrada arbitrária definida pelo usuário (uma lista de números quaisquer, por exemplo). Sua saída também pode ser arbitrariamente definida pelo usuário (desde uma lista de números, até um conjunto de triângulos que forma uma malha gerada proceduralmente).

[5] Mais informações em: https://www.amd.com/pt/products/processors/desktops/ryzen/7000-series/amd-ryzen-9-7950x3d.html

[6] Mais informações em: https://www.nvidia.com/pt-br/geforce/graphics-cards/40-series/rtx-4090/

Destes, os tipos de *shaders* que serão mais discutidos neste trabalho são os *vertex shaders* e os *fragment shaders*. Um exemplo de uso de *vertex shaders* é na modificação procedural de malhas 3D já existentes sem a necessidade de criar uma nova malha. A Figura 2.22 mostra a deformação de uma malha esférica por um *vertex shader* usando uma onda senoidal.

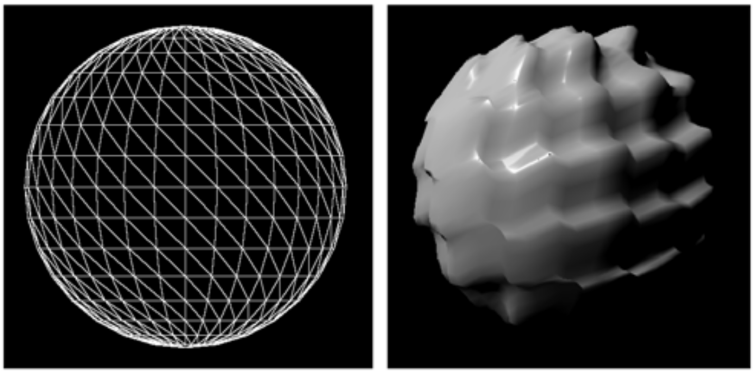

Figura 2.22: Exemplo de uso de um *vertex shaders* para deformar os vértices da malha esférica baseado em uma onda senoidal. À esquerda está a malha original e à esquerda está a malha (renderizada e sombreada) modificada pelo *vertex shader*.
Fonte: Harlem Batagelo e Shin-Ting Wu, 2008. Disponível: https://www.researchgate.net/figure/Top-row-Sphere-deformed-by-sine-functions-evaluated-on-a-vertex-shader-Bottom-row-Quad_fig1_220068295

Já os *fragment shaders* são comumente utilizados após o passo de rasterização para colorir cada pixel que faz parte de um triângulo. Triângulos diferentes podem ter *fragment shaders* diferentes associados a eles. A Figura 2.23 mostra um *fragment shader* sendo utilizado para colorir um triângulo com um gradiente de cores.

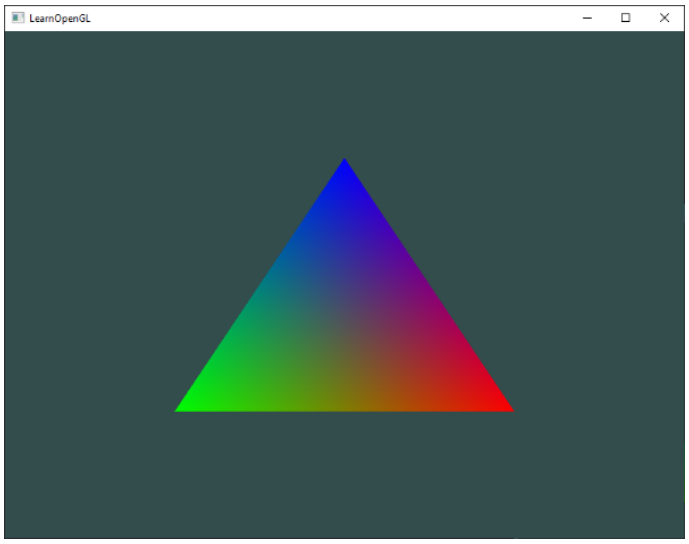

Figura 2.23: Exemplo de uso de um *fragment shader* para colorir um triângulo com um gradiente de cores.
Fonte: Joey de Vries. Disponível: https://learnopengl.com/img/getting-started/shaders3.png

Capítulo 3

Metodologia

Neste capítulo serão apresentados em detalhes os sistemas mais importantes desenvolvidos para o renderizador: a *pipeline* gráfica, o modelo de dados, a descrição das primitivas, a descrição das transformações, a descrição dos materiais, a descrição dos modelos 3D e a descrição das cenas.

O renderizador foi desenvolvido na linguagem Rust[1], utilizando o pacote WGPU[2] para lidar com o *backend* gráfico da aplicação (ou seja, a interface com a GPU do computador) e com o gerenciamento de janelas (para que seja possível visualizar o conteúdo renderizado em uma janela do sistema operacional). Para a escrita dos shaders do renderizador foi utilizada a linguagem Slang[3], e os *shaders* foram compilados para a linguagem binária intermediária SPIR-V[4], que pode ser diretamente utilizada pelo WGPU. Para serialização e desserialização de dados foi utilizado majoritariamente o formato JSON e a biblioteca Serde[5].

A linguagem Rust foi escolhida para o projeto devido ao seu bom repositório de pacotes, ao robusto sistema de tipos e pela experiência prévia do autor. O pacote WGPU foi utilizado para minimizar o tempo gasto com desenvolvimento do *backend* gráfico da aplicação e com o gerenciamento de janelas do sistema operacional. A linguagem de shader Slang foi escolhida pelas boas ferramentas de desenvolvimento feitas para ela (compilador, servidor de linguagem, entre outras) e por sua versatilidade, podendo ser transpilada para várias outras linguagens de shader, como GLSL e HLSL, ou, ainda, compilada para representações binárias como SPIR-V, utilizado nesse projeto. Finalmente, a linguagem binária intermediária SPIR-V foi escolhida por sua alta compatibilidade com outros projetos já existentes, como a *game engine* Godot, que também utiliza SPIR-V como linguagem intermediária para seus shaders[6]. O formato JSON foi escolhido por sua alta legibilidade por humanos (como não há uma forma gráfica de editar os objetos serializados ainda, é um grande benefício para poder editá-los diretamente). A biblioteca Serde foi escolhida por sua facilidade de uso e grande interoperabilidade com outras bibliotecas de Rust.

[1] Mais informações em: https://www.rust-lang.org/
[2] Mais informações em: https://wgpu.rs/
[3] Mais informações em: https://shader-slang.com/
[4] Mais informações em: https://www.khronos.org/spir/
[5] Mais informações em: https://serde.rs/
[6] Mais informações em: https://docs.godotengine.org/en/stable/classes/class_rdshaderspirv.html

3.1 A *Pipeline* Gráfica

A primeira etapa no desenvolvimento da aplicação foi o desenvolvimento da *pipeline* gráfica do renderizador. Uma *pipeline* gráfica consiste na sequência de passos que um renderizador toma para gerar uma imagem [5]. Como mencionado no Capítulo 2, renderizadores que utilizam rasterização costumam utilizar modelos 3D compostos de triângulos, que são, por sua vez, descritos por 3 vértices cada. Portanto, a *pipeline* gráfica nesses renderizadores opera sobre os vértices e os triângulos. Tomas Akenine-Möller, Eric Haines e Naty Hoffman [2] dividem a *pipeline* gráfica tradicional em três etapas (cada uma podendo ser subdividida em mais etapas): a etapa da aplicação, a etapa da geometria e a etapa do rasterizador.

Na etapa da aplicação, o desenvolvedor tem total controle dos procedimentos, já que ela ocorre na CPU. Caso esteja sendo desenvolvida uma simulação física, por exemplo, nessa etapa ocorreria o passo da simulação, em que os objetos seriam modificados pelo solucionador do motor de física utilizado. Já a etapa da geometria é responsável por modificar, por meio de transformações, os vértices dos modelos 3D que devem ser renderizados. Essa etapa pode ser modificada pelo desenvolvedor por meio de *vertex shaders*, que podem ser programados para interagir com os vértices dos triângulos que compõem a cena. Por fim, a etapa do rasterizador passa por cada um dos triângulos da cena, utilizando seus vértices (anteriormente transformados) para determinar quais pixels da tela serão utilizados para renderizar cada triângulo. Os pixels de um triângulo são chamados de seus fragmentos. Ainda na etapa de rasterização, cada fragmento de cada triângulo passa por um *fragment shader*, que também pode ser programado pelo desenvolvedor para alterar a cor final de cada pixel baseado, por exemplo, nas propriedades fotogramétricas de cada material. Finalmente, os resultados do *fragment shader* de cada triângulo são agregados para formar a imagem final. A Figura 3.1 mostra a *pipeline* gráfica de um rasterizador.

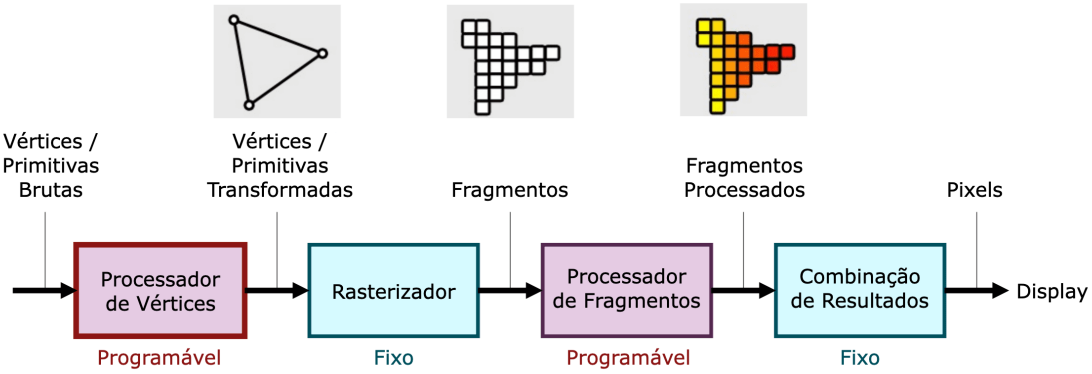

Figura 3.1: Visualização de uma pipeline gráfica de um renderizador por rasterização. Em roxo estão representadas as etapas programáveis, que podem ser modificadas pelo desenvolvedor por meio de *vertex shaders* e *fragment shaders*, por exemplo. Em azul estão representadas as etapas fixas, que não podem ser modificadas pelo desenvolvedor.
Adaptado de: Youngdo Lee, 2024. Disponível:
https://leeyngdo.github.io/blog/computer-graphics/2024-02-29-graphics-pipeline/

Em contrapartida, o renderizador desenvolvido neste trabalho, por utilizar *Ray Mar-*

ching, possui uma *pipeline* gráfica diferente. A etapa da aplicação continua a mesma, mas, como o renderizador foi feito para aceitar diversas primitivas diferentes, não apenas triângulos, a etapa de processamento de vértices não existe. Além disso, a etapa do rasterizador é, naturalmente, substituída por uma etapa de *Ray Marching*. Nessa etapa é executado o algoritmo descrito na Seção 2.5. Essa etapa foi implementada no renderizador na forma de um shader em Slang que é executado uma vez para cada pixel na tela. Portanto, o laço da linha 6 do Algoritmo 8 é feito implicitamente, já que o shader é executado em paralelo para cada pixel. O resultado final do *Ray Marcher* é o mesmo que o do rasterizador: um conjunto de fragmentos que pertencem a cada primitiva da cena. As etapas subsequentes da *pipeline* gráfica são as mesmas. Dessa forma, a diferença para a *pipeline* na Figura 3.1 é que a etapa de processamento de vértices não existe, e a etapa de rasterização é substituída por uma etapa de *Ray Marching*. As diferenças mencionadas estão representadas na Figura 3.2.

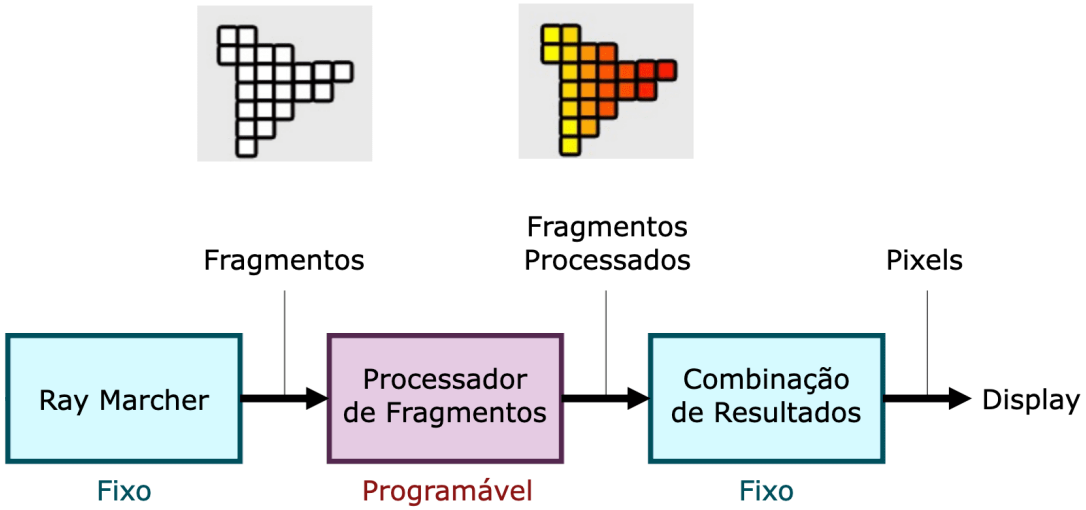

Figura 3.2: Visualização da pipeline gráfica do renderizador de *Ray Marching* desenvolvido. Em roxo estão representadas as etapas programáveis, que podem ser modificadas pelo desenvolvedor por meio de *fragment shaders*, por exemplo. Em azul estão representadas as etapas fixas, que não podem ser modificadas pelo desenvolvedor.
Adaptado de: Youngdo Lee, 2024. Disponível:
https://leeyngdo.github.io/blog/computer-graphics/2024-02-29-graphics-pipeline/

Utilizar uma *pipeline* gráfica parecida com as comumente utilizadas na indústria foi uma decisão deliberada para minimizar o custo de desenvolvimento. Ao utilizar um processo próximo ao padrão da indústria, é possível encontrar muitos materiais de estudo e ferramentas de desenvolvimento que, ainda que não se apliquem diretamente a um renderizador por *Ray Marching*, podem vir a ser úteis dada a similaridade entre as *pipelines* gráficas.

3.2 Modelo de dados

O modelo de dados define os dados que devem ser transferidos da CPU para a GPU na passagem da etapa de aplicação para a etapa de geometria. Esses dados transferidos

devem, idealmente, ser reduzidos o máximo possível pois a transferência de dados entre a CPU e a GPU pode representar um grande gargalo de performance em aplicações gráficas [13]. Portanto, a maior parte dos dados relevantes para o *Ray Marching* e o *Sphere Tracing* são mantidos permanentemente na GPU. O maior exemplo disso são as FDSs das primitivas, que foram implementadas na linguagem de shader Slang, e são mantidas na GPU durante todo ciclo de vida da aplicação. Dessa forma, os únicos dados que precisam ser transferidos da CPU para a GPU são os dados dos objetos que compõem a cena. Esses objetos foram divididos em dois grupos: os objetos estáticos e os objetos dinâmicos.

Os objetos estáticos são as partes da cena que não são modificados pela simulação. Eles irão variar de aplicação para aplicação, mas, utilizando o exemplo de jogos, podemos citar árvores e paredes que, ainda que possam bloquear o movimento do jogador, não se movem. Em outras palavras, objetos estáticos não são modificados durante a simulação na etapa de aplicação. Já os objetos dinâmicos são aqueles que podem ser modificados durante a simulação. A maneira como isso ocorre pode variar, desde uma simulação de física complexa, até um simples sistema de interação com o usuário. Mas, se qualquer propriedade do objeto for modificada ao longo da simulação, esse objeto deve ser dinâmico.

Essa distinção é importante pois, dessa forma, é possível otimizar a comunicação entre a CPU e a GPU. Os dados referentes aos objetos dinâmicos precisam ser enviados para a GPU a cada novo quadro que será renderizado. Caso isso não ocorra, a simulação da aplicação na CPU e a renderização dos gráficos na GPU ficará dessincronizada. No entanto, o mesmo não acontece com os objetos estáticos. Como estes nunca são modificados durante a simulação, seus dados podem ser enviados para a GPU apenas uma vez no começo da execução da aplicação. Dessa forma, é possível reduzir o fluxo de dados entre a CPU e a GPU a cada quadro.

Independente de se tratar de um objeto estático ou dinâmico, os dados que a GPU precisa são os mesmos: o modelo 3D do objeto, as transformação do objeto e o material do objeto.

3.2.1 Descrição de primitivas

Como explicado anteriormente, todas as formas primitivas suportadas pelo renderizador foram estaticamente implementadas na linguagem de *shader* Slang, e são compiladas junto com o restante dos *shaders* utilizados pela aplicação. Dessa forma, os únicos dados que precisam ser transferidos da CPU para a GPU são dados sobre as propriedades da primitiva. Em primeiro lugar, a GPU precisa saber qual primitiva ela deve desenhar. Para isso, um dos dados passado é um identificador que consiste em um único número inteiro de 32 bits que corresponde a uma primitiva específica. Além desse identificador, cada primitiva possui propriedades únicas. Por exemplo, para renderizar uma esfera, é necessário saber seu raio. Já um cuboide necessita do tamanho de cada um de seus lados. Como formas primitivas diferentes possuem um número variado de propriedades, todas as primitivas transferem oito valores em ponto flutuante de 32 bits para a GPU, e a GPU utiliza esses dados de acordo com o tipo de primitiva que está sendo renderizada. Caso seja uma esfera, o primeiro valor será interpretado como o raio, e os outros sete serão descartados. No caso de um cuboide, os três primeiros valores serão interpretados como as dimensões nos eixos X, Y e Z, respectivamente,

e os demais valores serão descartados. O Algoritmo 9 mostra a implementação em Rust do tipo de dados utilizado para descrever uma primitiva em 3D na CPU. Já o Algoritmo 10 mostra a implementação da mesma estrutura de dados na GPU utilizando a linguagem de *shader* Slang.

Algoritmo 9 Implementação em Rust da estrutura de dados utilizada para formas primitivas em 3D na CPU. O tipo `Primitive3D` é implementado em Rust como um enumerador que carrega os dados das estruturas de cada uma das formas primitivas. O Algoritmo mostra somente a implementação das formas primitivas de esfera e cuboide.

```
pub struct Primitive3DSphere {
    radius: f32
}

pub struct Primitive3DCuboid {
    width: f32,
    height: f32,
    depth: f32
}

pub enum Primitive3D {
    Sphere(Primitive3DSphere),
    Cuboid(Primitive3DCuboid)
}
```

Algoritmo 10 Implementação em Slang da estrutura de dados utilizada para formas primitivas em 3D na GPU.

```
struct Primitive3D {
    int id;
    float[8] data;
}
```

Além das primitivas previamente implementadas pelo renderizador, o usuário também pode definir formas primitivas customizadas. Essa especificação é feita por meio de arquivos `JSON` e precisam ser compilados junto com o renderizador. O arquivo criado pelo usuário deve conter a escolha de um identificador para a nova forma primitiva, e a implementação em Slang da FDS da primitiva. Além disso, um nome para a primitiva pode ser opcionalmente definido. No Algoritmo 11 foi definida a forma primitiva de um prisma hexagonal com o `id` de 33. O valor de 33 foi escolhido pois os primeiros 32 valores de `id` são reservados para primitivas implementadas diretamente pelo renderizador.

Algoritmo 11 Exemplo de `JSON` que pode ser utilizado para definir uma nova primitiva no renderizador. Este arquivo definiria uma primitiva de um prisma hexagonal [18] com um `id` de 33.

```
{
"name": "HexPrism",
"id": 33,
"code":
    "float sdHexPrism(vec3 p, Primitive3D primitive) {
        float2 h = float2(primitive.data[0], primitive.data[1]);

        const vec3 k = vec3(-0.8660254, 0.5, 0.57735);
        p = abs(p);
        p.xy -= 2.0*min(dot(k.xy, p.xy), 0.0)*k.xy;
        vec2 d = vec2(
            length(p.xy - vec2(clamp(p.x,-k.z*h.x,k.z*h.x), h.x))*sign(p.y
   -h.x), p.z-h.y);
        return min(max(d.x,d.y),0.0) + length(max(d,0.0));
    }"
}
```

A implementação em Rust das primitivas, no Algoritmo 9, também precisou ser aprimorada para ser compatível com a descrição de primitivas customizadas. O Algoritmo 12 mostra a nova implementação para ser compatível com primitivas customizadas.

Algoritmo 12 Extensão da implementação em Rust mostrada no Algoritmo 9 para ser compatível com primitivas customizadas.

```
pub struct Primitive3DSphere {
    radius: f32
}

pub struct Primitive3DCuboid {
    width: f32,
    height: f32,
    depth: f32
}

pub struct Primitive3DCustom {
    id: i32,
    data: [f32; 8]
}

pub enum Primitive3D {
    Sphere(Primitive3DSphere),
    Cuboid(Primitive3DCuboid),
    Custom(Primitive3DCustom)
}
```

3.2.2 Descrição de transformações

Para o renderizador, as transformações são o conjunto de translações, rotações e mudanças de escala aplicadas sobre um determinado objeto. Seu modelo de dados consiste no uso de três conjuntos de três propriedades, uma para cada eixo de cada transformação. O Algoritmo 13 mostra a implementação em Rust do tipo de dados utilizado para descrever uma transformação em 3D na CPU. Já o Algoritmo 14 mostra a implementação da mesma estrutura de dados na GPU utilizando a linguagem de *shader* Slang.

Algoritmo 13 Implementação em Rust da estrutura de dados utilizada para transformações em 3D na CPU.

```
pub struct Float3 {
    x: f32,
    y: f32,
    z: f32,
}

pub struct Transform3D {
    translation: Float3,
    rotation: Float3,
    scale: Float3
}
```

Algoritmo 14 Implementação em Slang da estrutura de dados utilizada para transformações em 3D na GPU.

```
struct Transform3D {
    float3 translation;
    float3 rotation;
    float3 scale;
}
```

Essas transformações são utilizadas pelo renderizador antes de calcular a FDS de uma forma para modificar o resultado final. Elas são aplicadas sobre a imagem da FDS por meio de um conjunto de multiplicações de matrizes 4x4. Como a multiplicação de matrizes não é comutativa, a ordem de multiplicação escolhida foi a seguinte: primeiro é aplicada a transformação de escala, depois a de rotação e, por fim, a de translação.

Ademais, para que seja possível multiplicar um valor $x \in \mathbb{R}^3$ por uma matriz $A \in \mathbb{R}^{4\times4}$ é necessário que x seja transformado em um valor no $\mathbb{R}^4$ adicionando um número 1 ao final dele.

3.2.2.1 A matriz de translação

As matrizes de translação possuem a seguinte forma geral:

$$T = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Onde t_x, t_y e t_z correspondem ao valor da translação da transformação nos eixos X, Y e Z, respectivamente. Um exemplo de translação sobre um cuboide pode ser visto na Figura 3.3, onde, em vermelho está o eixo X, em verde o eixo Y, em azul o eixo Z e em amarelo o plano XY.

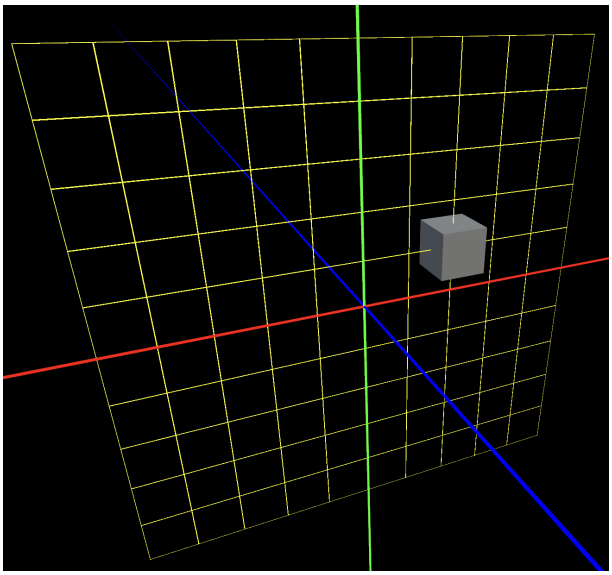

Figura 3.3: Visualização de um cuboide com translação de $(2, 1, 0)$. Em vermelho está o eixo X, em verde o eixo Y, em azul o eixo Z e em amarelo o plano XY.

3.2.2.2 A matriz de escala

As matrizes de escala possuem a seguinte forma geral:

$$S = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Onde s_x, s_y e s_z correspondem ao valor da escala nos eixos X, Y e Z, respectivamente. Um exemplo de escala sobre um cuboide pode ser visto na Figura 3.4, onde, em vermelho está o eixo X, em verde o eixo Y, em azul o eixo Z, em amarelo o plano XY e em magenta o plano XZ.

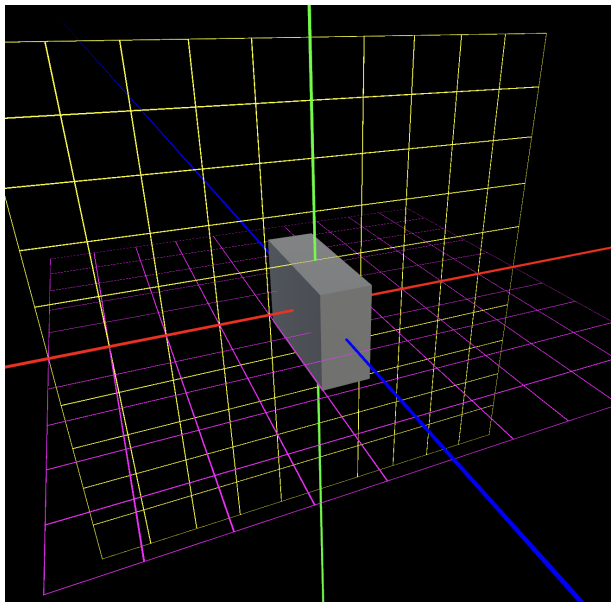

Figura 3.4: Visualização de um cuboide com escala de $(1, 2, 4)$. Em vermelho está o eixo X, em verde o eixo Y, em azul o eixo Z, em amarelo o plano XY e em magenta o plano XZ.

3.2.2.3 A matriz de rotação

As matrizes de rotação são diferentes a depender do eixo em que se deseja rotacionar. Elas possuem a seguinte forma geral:

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(r_x) & \sin(r_x) & 0 \\ 0 & -\sin(r_x) & \cos(r_x) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y = \begin{bmatrix} \cos(r_y) & 0 & -\sin(r_y) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(r_y) & 0 & \cos(r_y) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z = \begin{bmatrix} \cos(r_z) & -\sin(r_z) & 0 & 0 \\ \sin(r_z) & \cos(r_z) & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Onde R_x, R_y e R_z são as matrizes de rotação nos eixos X, Y e Z, respectivamente, e r_x, r_y e r_z correspondem ao valor da rotação nos eixos X, Y e Z, respectivamente. Ademais, como cada objeto possui três matrizes de rotação, a ordem de multiplicação dessas matrizes também precisa ser definida para gerar a matriz global de rotação do objeto. A ordem utilizada no renderizador foi a seguinte:

$$R = R_z * R_y * R_x$$

Um exemplo de rotação sobre um cuboide pode ser visto na Figura 3.5, onde, em vermelho está o eixo X, em verde o eixo Y, em azul o eixo Z, em amarelo o plano XY, em magenta o plano XZ e em ciano o plano YZ.

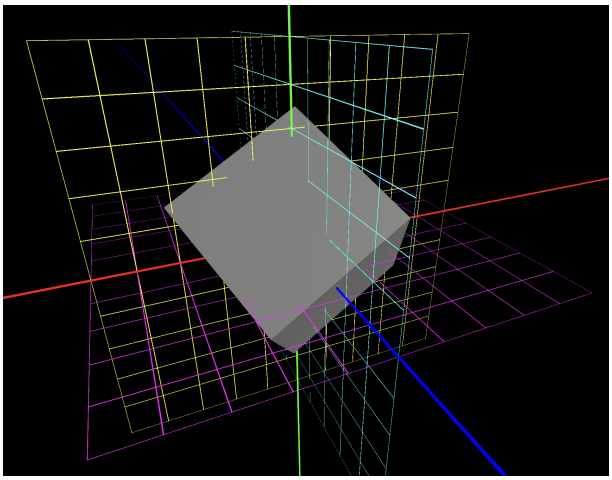

Figura 3.5: Visualização de um cuboide com rotação de $(\frac{\pi}{4}, \frac{\pi}{6}, \frac{\pi}{4})$. Em vermelho está o eixo X, em verde o eixo Y, em azul o eixo Z, em amarelo o plano XY, em magenta o plano XZ e em ciano o plano YZ.

3.2.3 Descrição de materiais

Os materiais representam as propriedades fotogramétricas dos objetos na cena e ditam como os raios de luz irão interagir com as superfícies, determinando, assim, a cor final dos objetos. Cada material existe como uma instância de um *fragment shader* que, por sua vez, é apenas uma função cujo resultado final deve ser a cor do objeto. Os parâmetros dessa função podem, no entanto, variar. Essa variação é dada por diferentes instâncias de um mesmo *shader*. Essas diferentes instâncias são chamadas de materiais. Assim, pode-se entender o *fragment shader* como a definição de uma classe, e o material como uma instância dessa classe.

Tanto o *fragment shader* quanto o material são definidos em arquivos `JSON`. O *fragment shader* possui duas partes. A primeira é a definição das propriedades. Esses são os valores que serão alterados a cada instância do *shader* (ou seja, em cada material). A segunda parte é a definição do código. O código de um *fragment shader* precisa definir uma função `shade` que possui como parâmetro um único objeto do tipo `FragmentProperties`, e retorna a cor do objeto para um dado pixel. Além disso, devem estar definidos no escopo global variáveis para cada um dos parâmetros que foram definidos na seção anterior.

O Algoritmo 15 mostra a implementação do tipo `FragmentProperties` na linguagem de *shader* Slang. As propriedades presentes na estrutura de dados `FragmentProperties` são:

- `world_position`: posição no espaço global do pixel que está sendo renderizado agora.

Corresponde ao ponto em que ocorreu a colisão entre o raio emitido pelo pixel atual e a geometria da cena;

- `object_position`: posição do pixel atual no espaço do objeto. Corresponde à diferença entre `world_position` e a translação do objeto;
- `world_normal`: vetor normal à superfície no ponto de colisão do raio emitido pelo pixel atual;
- `ray_direction`: direção do raio emitido pelo pixel atual;
- `ray_distance`: distância viajada pelo raio emitido pelo pixel atual até colidir com a geometria da cena.

Algoritmo 15 Implementação do tipo de dados `FragmentProperties`, que contém dados que um *fragment shader* pode utilizar para colorir um objeto.

```
1 struct FragmentProperties {
2     float3 world_position;
3     float3 object_position;
4     float3 world_normal;
5
6     float3 ray_direction;
7     float  ray_distance;
8 }
```

No Algoritmo 16 foi definido um *fragment shader* com nome de *Blend Shader*. Sua funcionalidade é de colorir todos os pixels de um objeto utilizando uma interpolação linear entre duas cores (`_Color1` e `_Color2`) baseada em um fator de interpolação (`_Blend`). Esses três valores foram definidos como propriedades e serão diferentes para cada material desse *shader*, como mostra o Algoritmo 17. As propriedades podem possuir anotações próximas de seus nomes. As anotações de tipo (`[float3]` e `[float]`) são necessárias. Existem, também, anotações opcionais, como o `[Range(0, 1)]`, que garante que o valor da propriedade `_Blend` sempre será um escalar entre zero e um.

Algoritmo 16 Exemplo de `JSON` que pode ser utilizado para definir um novo *fragment shader* no renderizador. Esse *shader* possui duas cores e um escalar como propriedades e define uma função que retorna a interpolação linear entre as duas cores baseado no valor do escalar.

```
{
"name": "Blend Shader",
"properties": [
    "[float3] _Color1",
    "[float3] _Color2",
    "[float] [Range(0, 1)] _Blend"
],
"code": "
    float3 _Color1;
    float3 _Color2;
    float  _Blend;

    float3 shade(FragmentProperties properties) {
        return lerp(_Color1, _Color2, _Blend);
    }"
}
```

No Algoritmo 17 foi definido um material para o *fragment shader* definido no Algoritmo 16. Esse material define os valores das propriedades que serão utilizadas pelo *shader* quando o objeto for renderizado. A Figura 3.6 mostra o material definido no Algoritmo 17 aplicado a uma esfera.

Algoritmo 17 Exemplo de `JSON` que pode ser utilizado para definir um novo material no renderizador, baseado no *fragment shader* definido no Algoritmo 16.

```
{
"base_shader": "Blend Shader",
"properties": [
    { "name": "_Color1", "type": "float3", "value": "(0.5,0.1,0.5)" },
    { "name": "_Color2", "type": "float3", "value": "(0.2,0.7,0.6)" },
    { "name": "_Blend",  "type": "float",  "value": "0.61803398874" }
]
}
```

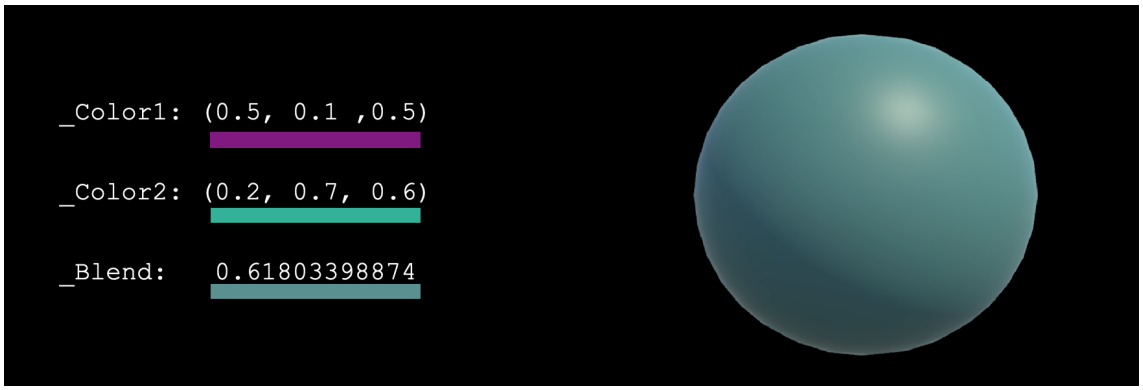

Figura 3.6: Material definido no Algoritmo 17 sobre uma esfera.

A descrição do material em formato `JSON` é feita apenas para serialização. Esse material é lido e colocado em uma estrutura de dados na inicialização do renderizador. O Algoritmo 18 mostra a implementação dos tipos `Material` e `MaterialProperty` em Rust. O tipo `MaterialProperty` é um enumerador com cada tipo possível de propriedade que um material pode ter, e carrega o valor da propriedade. Além de um membro do tipo `MaterialProperty`, o tipo `Material` também possui uma `String base_shader` que é o nome do *fragment shader* que é base daquele material.

Algoritmo 18 Implementação em Rust da estrutura de dados utilizada para materiais na CPU. O tipo `MaterialProperty` é um enumerador com cada tipo possível de propriedade que um material pode ter, e carrega o valor da propriedade. Além de um membro do tipo `MaterialProperty`, o tipo `Material` também possui uma `String base_shader` que é o nome do *fragment shader* que é base daquele material.

```
pub enum MaterialProperty {
    Int(i32),
    Int2((i32, i32)),
    Int3((i32, i32, i32)),
    Float(f32),
    Float2((f32, f32)),
    Float3((f32, f32, f32))
}

pub struct Material {
    base_shader: String,
    properties: HashMap<String, MaterialProperty>
}
```

3.2.4 Descrição de modelos 3D

Um modelo 3D é um objeto que pode ser composto por uma ou mais primitivas, utilizando uma série de operações unárias ou binárias. Assim como primitivas customizadas e *fragment shaders*, modelos 3D são definidos em arquivos `JSON`. O Algoritmo 19 define um modelo 3D composto de uma única forma primitiva de esfera de raio 0.5 na posição $(0.3, 0.2, 0.0)$, com escala $(1.0, 2.0, 4.0)$ e sem rotação. O modelo utiliza um único material, que está asso-

ciado à esfera, identificado pelo nome do arquivo de descrição do material `blend_material`. Por fim, a primitiva não possui nenhum filho na hierarquia (essa propriedade será melhor explicada mais adiante). A Figura 3.7 mostra uma visualização do modelo 3D definido no Algoritmo 19.

Algoritmo 19 Exemplo de `JSON` que pode ser utilizado para definir um novo modelo 3D no renderizador. Esse modelo consiste em uma única forma primitiva de esfera de raio 0.5 na posição $(0.3, 0.2, 0.0)$, com escala $(1.0, 2.0, 4.0)$ e sem rotação. O modelo utiliza um único material, que está associado à esfera, identificado pelo nome do arquivo de descrição do material `blend_material`. Por fim, a primitiva não possui nenhum filho (essa propriedade será melhor explicada mais adiante).

```
{
    "name": "Esfera",
    "materials": [
        "blend_material"
    ],
    "model": {
        "type": "Primitives/Sphere",
        "data": [0.5],
        "transform": {
            "position": { "x": 0.3, "y": 0.2, "z": 0.0 },
            "rotation": { "x": 0.0, "y": 0.0, "z": 0.0 },
            "scale":    { "x": 1.0, "y": 2.0, "z": 4.0 }
        },
        "material_id": 0,
        "children": []
    }
}
```

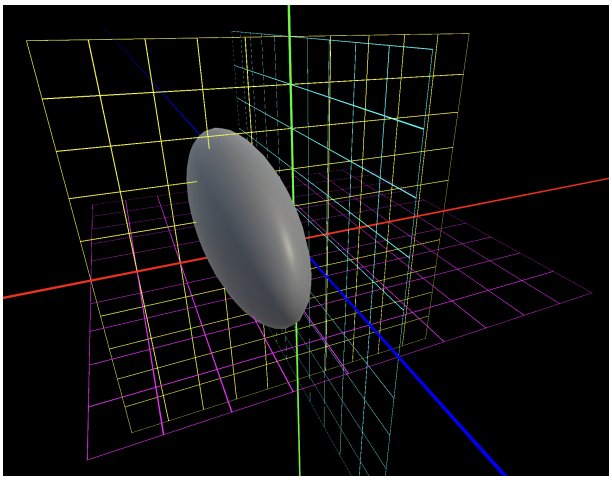

Figura 3.7: Visualização do modelo 3D definido no Algoritmo 19. Em vermelho está o eixo X, em verde o eixo Y, em azul o eixo Z, em amarelo o plano XY, em magenta o plano XZ e em ciano o plano YZ.

Uma forma útil de organizar uma estrutura de cena é utilizando uma hierarquia em formato de árvore. Para tal, objetos na cena podem definir os chamados filhos (no campo `children`). As transformações de um objeto filho sempre são relativas às transformações do

objeto pai. Quando um objeto do tipo `Operation` possui filhos, a operação será aplicada a todos os filhos. Caso uma forma primitiva tenha filhos definidos, a operação padrão de união será aplicada.

Algoritmo 20 Exemplo de `JSON` que pode ser utilizado para definir um novo modelo 3D no renderizador. Esse modelo consiste em três primitivas de esfera de raio 0.5 nas posições $(0.3, 0.2, 0.0)$, $(0.1, 0.4, 0.0)$ e $(0.3, 0.1, 0.0)$, respectivamente. Todas essas esferas são filhas de um único objeto do tipo `Operations/Union`, que irá aplicar a operação de união em todos os seus filhos. O modelo utiliza dois materiais, identificados pelo nome do arquivo de descrição do material `blend_material1` e `blend_material2`.

```
{
    "name": "TresEsferas",
    "materials": [
        "blend_material1",
        "blend_material2"
    ],
    "model": {
        "type": "Operations/Union",
        "material_id": 0,
        "children": [
                {"type": "Primitives/Sphere",
                 "data": [0.5],
                 "transform": {
                     "position": { "x": 0.3, "y": 0.2, "z": 0.0 }
                 }},
                {"type": "Primitives/Sphere",
                 "data": [0.5],
                 "material_id": 1,
                 "transform": {
                     "position": { "x": 0.1, "y": 0.4, "z": 0.0 }
                 }},
                {"type": "Primitives/Sphere",
                 "data": [0.5],
                 "transform": {
                     "position": { "x": 0.3, "y": 0.1, "z": 0.0 }
                 }}
        ]
    }
}
```

O Algoritmo 20 define um modelo 3D composto por três primitivas de esfera de raio 0.5 nas posições $(0.3, 0.2, 0.0)$, $(0.1, 0.4, 0.0)$ e $(0.3, 0.1, 0.0)$, respectivamente. Todas essas esferas são filhas de um único objeto do tipo `Operations/Union`, que irá aplicar a operação de união em todos os seus filhos. O modelo utiliza dois materiais, identificados pelo nome do arquivo de descrição do material. Assume-se que os campos omitidos no arquivos `JSON` (como `transform` e `children`) não são modificados. Quando um objeto filho não define o campo `material_id`, assume-se que ele herdou o material do pai na hierarquia. A Figura 3.8 mostra uma visualização do modelo 3D definido no Algoritmo 20. Em vermelho está o eixo X, em verde o eixo Y, em azul o eixo Z, em amarelo o plano XY, em magenta o plano XZ e em ciano o plano YZ.

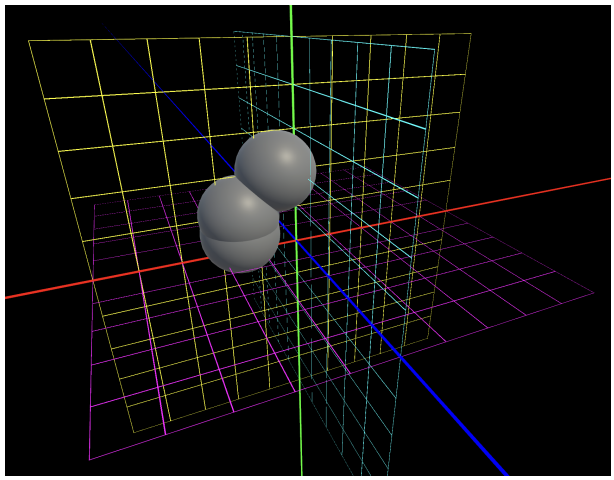

Figura 3.8: Visualização do modelo 3D definido no Algoritmo 20. Em vermelho está o eixo X, em verde o eixo Y, em azul o eixo Z, em amarelo o plano XY, em magenta o plano XZ e em ciano o plano YZ.

Algoritmo 21 Implementação em Rust da estrutura de dados utilizada para modelos 3D na CPU. O tipo `Operation3D` é um enumerador utilizado para identificar uma operação. O tipo `Object3D` é utilizado para representar qualquer objeto 3D que compõe um modelo 3D, seja ele uma primitiva, uma operação ou um modelo 3D completo. O tipo `Ojbect3DType` é utilizado para identificar os tipos diferentes de `Object3D`, e guardar os dados sobre cada um deles (caso o tipo `Model`, ele guarda apenas o nome do arquivo do modelo 3D). O tipo `Model3D` guarda a estrutura de árvore do modelo 3D, como a definida no Algoritmo 20.

```
pub enum Operation3D {
    Union,
    Intersection,
    Subtraction
}

pub enum Object3DType {
    Primitive(Primitive3D),
    Operation(Operation3D),
    Model(String)
}

pub struct Object3D {
    object_type: Object3DType,
    transform: Transform3D,
    material_id: i32,
    children: Vec<Object3D>
}

pub struct Model3D {
    name: String,
    materials: Vec<Material>,
    model: Object3D
}
```

Assim como os materiais, a descrição dos modelos 3D no formato `JSON` é utilizada

apenas para serialização. Quando o renderizador é iniciado, os arquivos `JSON` são interpretados e mantidos em memória na CPU. Para isso, o Algoritmo 21 mostra a implementação das estruturas de dados utilizadas para manter os modelos 3D na CPU. Não existe uma implementação de modelos 3D para a GPU pois cada primitiva que compõe um modelo 3D é individualmente transferida da CPU para a GPU na forma de uma `Primitive3D`, como mostrado no Algoritmo 10.

3.2.5 Descrição de cenas

Cenas, no contexto do renderizador, nada mais são do que conjuntos de modelos 3D dispostos de uma maneira específica. Portanto, a descrição da cena é muito parecida com a descrição dos modelos 3D. O Algoritmo 22 define uma nova cena composta pela união dos modelos 3D definidos no Algoritmo 19 (identificado pelo tipo `Models/Esfera`) e no Algoritmo 20 (identificado pelo tipo `Models/TresEsferas`). O segundo modelo possui uma translação de $(-0.4, 0.0, 0.0)$. A Figura 3.9 mostra uma visualização da cena definida no Algoritmo 22.

Algoritmo 22 Exemplo de `JSON` que pode ser utilizado para definir uma nova cena no renderizador. A cena é composta pela união dos modelos 3D definidos no Algoritmo 19 (identificado pelo tipo `Models/Esfera`) e no Algoritmo 20 (identificado pelo tipo `Models/TresEsferas`). O segundo modelo possui uma translação de $(-0.4, 0.0, 0.0)$.

```
{
    "name": "Cena1",
    "scene": [
       {"type": "Operations/Union",
        "children": [
           {"model": "Models/TresEsferas"},
           {"model": "Models/Esfera"},
            "transform": {
                "position": { "x": -0.8, "y": 0.0, "z": 0.0 }
            }}
        ]}
    ]
}
```

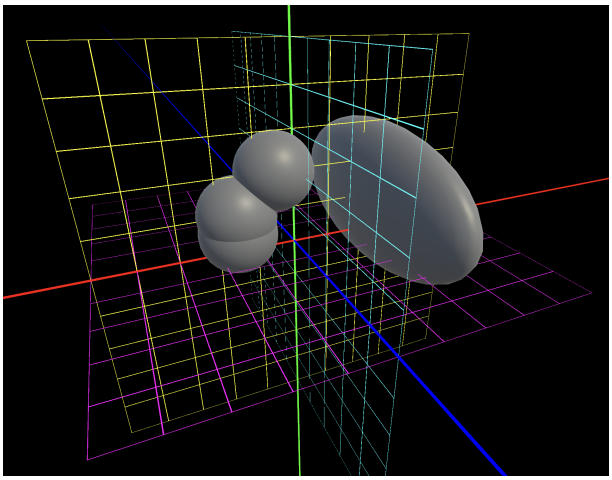

Figura 3.9: Visualização da cena definida no Algoritmo 22. Em vermelho está o eixo X, em verde o eixo Y, em azul o eixo Z, em amarelo o plano XY, em magenta o plano XZ e em ciano o plano YZ.

Assim como materiais e modelos 3D, as cenas utilizam o formato `JSON` apenas para serialização, e precisam ser lidas e mantidas em memória na CPU no início da execução. O Algoritmo 23 mostra a implementação da estrutura de dados utilizada para armazenar cenas na CPU. Assim como modelos 3D, não há uma representação de cenas na GPU, pois todos os elementos são enviados diretamente para a GPU na forma de primitivas.

Algoritmo 23 Implementação em Rust da estrutura de dados utilizada para cenas na CPU.

```
1 pub struct Scene3D {
2     name: String,
3     scene: Vec<Object3D>
4 }
```

Capítulo 4

Resultados

O renderizador pode ser utilizado por usuários para renderizar cenas utilizando *Ray Marching* em tempo real. Como o propósito principal do trabalho era avaliar a viabilidade desta técnica de renderização, o renderizador foi desenvolvido apenas como uma prova de conceito, não sendo recomendado para usuários iniciantes na área. Melhorias de usabilidade e novas funcionalidades estão planejadas para o futuro, e serão discutidas em mais detalhes na Seção 5.2. Mesmo não tendo uma usabilidade amigável, o renderizador cumpriu sua proposta de demonstrar a viabilidade da renderização de cenas tridimensionais em tempo real com *Ray Marching*. Para tal, foram escolhidas algumas cenas do artista Inigo Quilez [20] de complexidade variada para serem adaptadas ao renderizador desenvolvido. A performance de cada uma dessas cenas foi, então, medida em quadros por segundo (QPS) ao longo de 30 segundos de execução com resolução de 1920x1080. Essa medida permite determinar se um renderizador foi capaz de atingir taxas de quadros consideradas interativas. Para ser considerada interativa, uma aplicação deve ter uma taxa de QPS alta o suficiente para que um usuário consiga identificar movimentos e reagir apropriadamente.

Tomas Akenine-Möller, Eric Haines e Naty Hoffman [2] consideram que, a partir de 6 QPS, já é possível ter uma sensação de interatividade - ainda que limitada. A partir de 15 QPS, um usuário já é capaz de focar em ação e reação. A partir de 72 QPS, diferenças na taxa de atualização se tornam praticamente indetectáveis. No entanto, o aspecto visual não é o único fator relevante para uma aplicação. Outro fator relevante é a latência da interação com o usuário. A uma taxa de 15 QPS, embora o usuário seja capaz de identificar movimentos e reagir apropriadamente, ele só verá o resultado de sua interação exibido no próximo quadro, o que pode levar até 66 milissegundos. Embora taxas superiores a 72 QPS não ofereçam melhorias visuais significativas, quanto maior a taxa de atualização, menor será a latência que o usuário sentirá entre sua ação (como clicar em um botão no mouse) e a visualização do resultado dessa ação na tela. Para este trabalho, 60 QPS será considerada a taxa desejada para as cenas.

Todos os testes de performance foram feitos em um notebook com as seguintes especificações: CPU Intel(R) Core(TM) i5-13450HX, 24GB de RAM, GPU NVIDIA GeForce RTX 3050 Mobile, 6GB de VRAM. No ano de 2024, essa é uma máquina com performance mediana para os jogos do mercado: é possível jogar quase todos os jogos disponíveis, mas,

muitas vezes, será necessário comprometer a qualidade gráfica para atingir uma performance aceitável. Os resultados dos testes de performance podem ser vistos a seguir:

- A Figura 4.1 mostra a cena *Selfie Girl*, que atingiu a performance de 45 QPS. Embora não tenha atingido os 60 QPS desejados, essa é a cena mais complexa de todas as quatro cenas renderizadas e, ainda assim, consegue atingir uma taxa de quadros apropriada para ser considerada interativa;
- A Figura 4.2 mostra a cena *Snail*, que atingiu a performance de 120 QPS, bem acima da taxa desejada. Essa é, no entanto, a cena mais simples e bem otimizada dentre as quatro renderizadas;
- A Figura 4.3 mostra a cena *Happy Jumping*, que atingiu a performance de 61 QPS, exatamente a taxa desejada;
- Por fim, a Figura 4.4 mostra a cena *Greek Temple*, que atingiu a performance de 97 QPS, bem acima da taxa desejada.

Figura 4.1: Cena *Selfie Girl* renderizada utilizando o renderizador desenvolvido.
Cena original: Inigo Quilez, 2020. Disponível em: https://www.shadertoy.com/view/WsSBzh.

Figura 4.2: Cena *Snail* renderizada utilizando o renderizador desenvolvido.
Cena original: Inigo Quilez, 2015. Disponível em: https://www.shadertoy.com/view/WsSBzh.

Figura 4.3: Cena *Happy Jumping* renderizada utilizando o renderizador desenvolvido.
Cena original: Inigo Quilez, 2019. Disponível em: https://www.shadertoy.com/view/3lsSzf.

Figura 4.4: Cena *Greek Temple* renderizada utilizando o renderizador desenvolvido.
Cena original: Inigo Quilez, 2017. Disponível em: https://www.shadertoy.com/view/ldScDh.

Capítulo 5

Conclusão

5.1 Considerações finais

Este trabalho teve como objetivo avaliar a viabilidade da utilização do algoritmo de *Ray Marching* com *Sphere Tracing* na renderização de jogos digitais. Para demonstrar essa viabilidade, foi desenvolvida uma prova de conceito de um renderizador que utiliza essa técnica. O programa permite que usuários renderizem cenas tridimensionais utilizando *Ray Marching*, mas não possui uma interface de usabilidade amigável para usuários, visto que foi escrito apenas para efeitos de demonstração.

Os resultados dos testes de performance apresentados no Capítulo 4 superaram as expectativas. Todas as cenas que haviam sido utilizadas nos testes de performance foram originalmente desenvolvidas por Inigo Quilez no site *Shader Toy*[1]. Por se tratar de um ambiente em WebGL sendo executado no navegador, a performance das cenas não era nem próxima de ser suficiente. A maioria das cenas (com exceção da cena *Snail*) não conseguia chegar nem a uma taxa de 30 QPS sendo renderizada na resolução de 1280x720 (menos da metade da resolução utilizada nos testes de performance). Era esperado que, ao adaptar as cenas para um renderizador que seja executado nativamente, haveria um ganho de performance, mas não esperava que seria tão significativo. A cena *Selfie Girl*, por exemplo, sendo executada no *Shader Toy* com resolução de 1920x1080 possui performance em torno de 10 QPS, enquanto que, ao ser adaptada para o renderizador desenvolvido, sua performance aumentou quase cinco vezes, atingindo 45 QPS. A Tabela 5.1 mostra a comparação de performance entre as cenas sendo executadas no ambiente *WebGL* no *Shader Toy* e no renderizador desenvolvido. Como é possível ver pelos dados, todos os exemplos tiveram um ganho de performance entre quatro e cinco vezes quando comparados à performance no *Shader Toy*.

[1]Mais informações em: https://www.shadertoy.com/

Tabela 5.1: Tabela comparativa entre os resultados de performance no *Shader Toy* e no renderizador desenvolvido.

Cena	Performance no Shader Toy	Performance no renderizador	Ganho de performance
Selfie Girl	10 QPS	45 QPS	4.5x
Snail	27 QPS	120 QPS	4.44x
Happy Jumping	13 QPS	61 QPS	4.69x
Greek Temple	18 QPS	97 QPS	5.38x

Além disso, nenhuma das cenas utilizadas foi desenvolvida tendo em mente sua aplicação em jogos digitais ou aplicações interativas. Assim como malhas triangulares muito densas não são apropriadas para uso em renderizadores de tempo real por rasterização, as cenas construídas com superfícies implícitas também podem ser otimizadas para renderizadores por *Ray Marching* - com o custo de alguma perda de detalhes visuais. Isso significa que há um grande potencial de otimização dessas cenas para seu uso em tempo real. No entanto, mesmo sem esse trabalho feito, todas elas apresentaram taxas de quadros que podem ser consideradas interativas.

Um grande problema na análise dos resultados deste trabalho é que não há nenhuma forma muito simples de comparar a performance do renderizador desenvolvido com renderizadores por rasterização ou *Ray Tracing* disponíveis no mercado. Caso o renderizador desenvolvido fosse utilizado para renderizar uma malha de triângulos tradicional que um renderizador por rasterização utilizaria, a performance seria muito baixa. Isso ocorreria pois *Ray Marching* não foi criado para ser utilizado com esse tipo de modelo 3D. De forma semelhante, um renderizador por rasterização ou *Ray Tracing* também não conseguiria renderizar um modelo 3D feito para um renderizador por *Ray Marching*. São técnicas muito diferentes e difíceis de comparar.

No entanto é importante ressaltar que este trabalho não apresenta o *Ray Marching* como uma forma de substituir a rasterização ou o *Ray Tracing*. A proposta principal do trabalho era mostrar que *Ray Marching* é uma técnica viável para renderização de cenas 3D em tempo real, e este objetivo foi atingido com sucesso. Há efeitos que *Ray Marching* é capaz de produzir que seriam muito difíceis ou até impossíveis de reproduzir com rasterização ou *Ray Tracing*, e, nesses casos, *Ray Marching* pode ser utilizado pelos desenvolvedores mesmo quando execução em tempo real é uma necessidade.

Por fim, vale citar que este trabalho configurou parte importante do portfólio do autor para atingir a contratação em um emprego na área de computação gráfica e desenvolvimento de jogos voltados para aprendizado em uma das maiores empresas do mundo na área de tecnologia, a Amazon Web Services.

5.2 Trabalhos futuros

O autor pretende continuar trabalhando no renderizador e, futuramente, desenvolver uma *Game Engine* completa utilizando ele. Para isso, além de diversas melhorias no renderizador, muitos outros componentes precisarão ser desenvolvidos.

No que diz respeito a melhorias no renderizador, a maioria dos trabalhos que se deseja realizar no futuro próximo são melhorias de performance utilizando estruturas de aceleração. Mais especificamente, deseja-se implementar estruturas de *Bounding Volume Hierarchies* [3], juntamente com uma etapa de *Ray Tracing* antes de iniciar o *Ray Marching* para acelerar o processo. Funcionará de forma semelhante à renderização volumétrica discutida na Seção 2.1.3.1, mas, ao invés de utilizar Marcha Uniforme dentro do volume, seria utilizado *Sphere Tracing*. Isso permitiria essencialmente pular grande parte do laço de *Ray Marching*, já que o *Ray Tracing* consegue calcular intersecções de forma muito eficiente entre raios e primitivas simples como cubos e esferas. Outras formas de otimização também estão sendo estudadas e serão implementadas.

No que diz respeito a novos componentes a serem desenvolvido para tornar o projeto uma *Game Engine* completa, a primeira coisa a ser desenvolvida será uma biblioteca para lidar com operações comuns em desenvolvimento de jogos, como operações de álgebra linear. Além disso, será desenvolvido um *Entity Component System* para programar o comportamento dos objetos em um alto nível, semelhante ao que a *Bevy Engine* utiliza [26]. Também será desenvolvida uma linguagem de programação para ser usada na definição do comportamento dos objetos. O diferencial dessa linguagem será o fato de que ela será interpretada, mas, ela também poderá ser transpilada diretamente para Rust, e compilada junto com a aplicação final. Dessa forma, durante o desenvolvimento será possível usufruir do maior benefício de linguagens interpretadas: seu baixo tempo de iteração. Depois, quando o programa estiver finalizado, ele poderá ser transpilado para Rust e compilado junto com o resto da aplicação para ter a melhor performance possível de um código nativo. Por fim, será desenvolvida uma interface gráfica para a *Game Engine* com o intuito de tornar seu uso mais amigável para novos usuários. No entanto, também será possível utilizá-la apenas por código para os usuários que desejarem.

Todos esses componentes serão combinados em um produto final que estará disponível de forma livre e aberta a todos, permanentemente, podendo ser utilizado para qualquer finalidade desejada.

Capítulo 6

Bibliografia

[1] Michael Abrash. *Michael Abrash's Graphics Programming Black Book*. Coriolis, 1997.

[2] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[3] Jacco Bikker. How to build a bvh – part 1: Basics. Disponível em https://jacco.ompf2.com/2022/04/13/how-to-build-a-bvh-part-1-basics/. Acesso em 09/12/2024.

[4] CodeParade. How to make 3d fractals. Disponível em https://www.youtube.com/watch?v=svLzmFuSBhk. Acesso em 27/10/2024.

[5] OpenGL Wiki contributors. Rendering pipeline overview. Disponível em https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview. Acesso em 10/11/2024.

[6] Tomáš Davidovič, Thomas Engelhardt, Iliyan Georgiev, Philipp Slusallek, and Carsten Dachsbacher. 3d rasterization: a bridge between rasterization and ray casting. In *Proceedings of Graphics Interface 2012*, GI '12, page 201–208, CAN, 2012. Canadian Information Processing Society.

[7] Harlen Batagelo e Bruno Marques. Mcta008-17 computação gráfica. Disponível em https://www.brunodorta.com.br/cg/. Acesso em 18/04/2024.

[8] Peter Shirley e Trevor David Black e Steve Hollasch. Ray tracing in one weekend. Disponível em: https://raytracing.github.io/books/RayTracingInOneWeekend.html, April 2024. Acesso em 18/04/2024.

[9] Steven White; et al. Rasterization rules (direct3d 9). Disponível em: https://learn.microsoft.com/en-us/windows/win32/direct3d9/rasterization-rules, Abril 2021. Acesso em 18/04/2024.

[10] Randima Fernando. *GPU gems 1 GPU gems:Programming Techniques, tips, and tricks for real-time graphics*, volume 1. Addison-Wesley, 1 edition, 2004.

[11] Shader Fun. Signed distance fields part 1: Unsigned distance fields. Disponível em: https://shaderfun.com/2018/03/23/signed-distance-fields-part-1-unsigned-distance-fields/, Março 2018. Acesso em 18/04/2024.

[12] Rainer Goebel. Spatial transformation matrices. Disponível em https://www.brainvoyager.com/bv/doc/UsersGuide/CoordsAndTransforms/SpatialTransformationMatrices.html. Acesso em 16/11/2024.

[13] Garret Gunnel. I tried recreating counter strike 2's smoke grenades. Disponível em https://www.youtube.com/watch?v=ryB8hT5TMSg. Acesso em 27/10/2024.

[14] J. C. Hart, D. J. Sandin, and L. H. Kauffman. Ray tracing deterministic 3-d fractals. *ACM SIGGRAPH Computer Graphics*, 23(3):289–296, July 1989.

[15] John C. Hart. Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12(10):527–545, December 1996.

[16] Irving Kaplansky. *Set theory and metric spaces*. Boston, MA u.a, 1975.

[17] Inigo Quilez. 2d distance functions. Disponível em https://iquilezles.org/articles/distfunctions2d/. Acesso em 30/10/2024.

[18] Inigo Quilez. Distance functions. Disponível em https://iquilezles.org/articles/distfunctions/. Acesso em 30/10/2024.

[19] Inigo Quilez. Domain repetition. Disponível em https://iquilezles.org/articles/sdfrepetition/. Acesso em 30/10/2024.

[20] Inigo Quilez. Ray marching distance fields. Disponível em https://iquilezles.org/articles/raymarchingdf/. Acesso em 01/12/2024.

[21] Inigo Quilez. Sdf bounding volumes. Disponível em https://iquilezles.org/articles/sdfbounding/. Acesso em 24/10/2024.

[22] Scratchapixel. An overview of the rasterization algorithm. Disponível em: https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation/overview-rasterization-algorithm.html. Acesso em 26/10/2024.

[23] Scratchapixel. The visibility problem. Disponível em: https://www.scratchapixel.com/lessons/3d-basic-rendering/rendering-3d-scene-overview/visibility-problem.html. Acesso em 26/10/2024.

[24] SimonDev. Ray marching, and making 3d worlds with math. Disponível em https://www.youtube.com/watch?v=BNZtUB7yhX4. Acesso em 30/10/2024.

[25] Walt Disney Animation Studios. Disney's practical guide to path tracing. Disponível em https://www.youtube.com/watch?v=frLwRLS_ZR0. Acesso em 26/10/2024.

[26] Carter Anderson *et al.* Bevy quick start - entity component system. Disponível em https://bevyengine.org/learn/quick-start/getting-started/ecs/. Acesso em 09/12/2024.

[27] Daniel Weiskopf. *GPU-based interactive visualization techniques*. Springer, 2007.

[28] Alan Wolfe. Raytracing reflection, refraction, fresnel, total internal reflection, and beer's law. Disponível em https://wp.me/p8L9R6-24f. Acesso em 24/10/2024.

[29] Alexandre Ziebert. Precisamos falar sobre ray tracing. Disponível em https://www.nvidia.com/pt-br/drivers/prbr-05282018/. Acesso em 18/04/2024.