Renderização em Tempo Real utilizando Ray Marching e sua Aplicação em Jogos Digitais

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

Departamento de Ciência da Computação, Instituto de Matemática e Estatística, Universidade de São Paulo

Motivação

A **rasterização** é a técnica predominante na renderização de jogos 3D devido à sua eficiência e suporte por hardware gráfico dedicado [2]. No entanto, seu uso impõe limitações no processo criativo devido a particularidades do funcionamento dessa técnica [1]. Como exemplos, podemos citar:

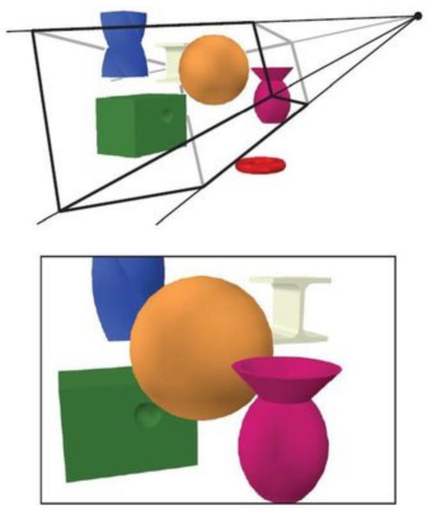
- **Efeitos complexos de iluminação** (reflexão, refração e Lei de Beer). Como o algoritmo de rasterização não simula o comportamento físico da luz no mundo real, reproduzir estes efeitos se torna computacionalmente custoso e pouco intuitivo;
- **Geometria procedural**. Como a grande maioria dos renderizadores trabalha com malhas de triângulos, qualquer sistema de geração procedural precisa criar essas malhas, dificultando, ou, até, inviabilizando, em alguns casos, sua execução em tempo real.

O ***Ray Tracing*** pode ser utilizado junto com a rasterização para simular efeitos de iluminação, mas é uma técnica computacionalmente cara [1], se tornando inviável para renderização completa de cenas complexas. Nesse contexto, o ***Ray Marching*** surge como uma alternativa promissora, oferecendo maior flexibilidade criativa [5]. Por simular o comportamento da luz de forma semelhante ao *Ray Tracing*, é possível criar os mesmos efeitos de iluminação com *Ray Marching*. Adicionalmente, por operar sobre superfícies implicitamente definidas por funções de distância, a geração de conteúdo procedural também pode ser implementada diretamente [5].

Assim, esse trabalho busca avaliar a viabilidade do uso de ***Ray Marching***, utilizando o algoritmo de ***Sphere Tracing***, para a renderização de cenas tridimensionais em aplicações interativas de tempo real. Além disso, será desenvolvido um renderizador utilizando a técnica de renderização supracitada como uma demonstração prática da viabilidade do algoritmo.

Estratégias de Renderização

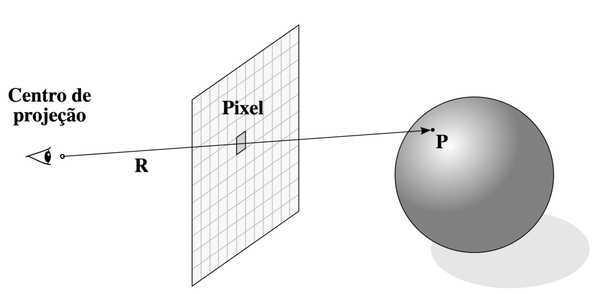
A função principal de qualquer **renderizador** é resolver o problema da **visibilidade de objetos** [1]. Esse problema consiste na determinação de quais regiões de uma superfície estão visíveis para um determinado observador. Um algoritmo de renderização pode ser entendido como uma caixa preta que, 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. A Figura ao lado mostra um exemplo de problema de visibilidade em uma cena contendo vários objetos.



Fonte: Allan Ryan, 2018.

Ray Tracing

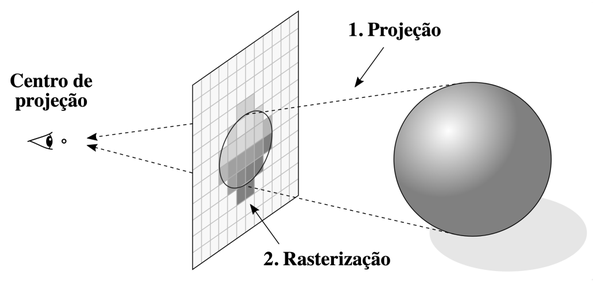
Ray Tracing faz parte do grupo de técnicas baseadas na simulação do caminho dos raios de luz [4]. Estas técnicas, de forma geral, procuram resolver o problema da visibilidade simulando como a luz reflete nas superfícies do mundo real. *Ray Tracing*, especificamente, utiliza fórmulas de intersecção e propriedades fotogramétricas dos objetos para calcular reflexões dos raios de luz na cena, até que eles cheguem à câmera. A Figura abaixo mostra a renderização de uma esfera utilizando *Ray Tracing*. Para cada pixel da imagem final, é emitido um ou mais raios da perspectiva do observador ao pixel para calcular sua cor.



Fonte: Harlen Batagelo, Bruno Marques, 2021.

Rasterização

Para renderizar uma cena tridimensional, um renderizador por **rasterização** passa por três etapas [2]. Primeiro, é feita a projeção de todas as primitivas que compõem a cena sobre a imagem final. Embora seja possível calcular a projeção de diversas formas geométricas sobre um plano, o triângulo é a forma primitiva mais usualmente utilizada pelos renderizadores. 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 Figura abaixo mostra o processo de rasterização de uma esfera.

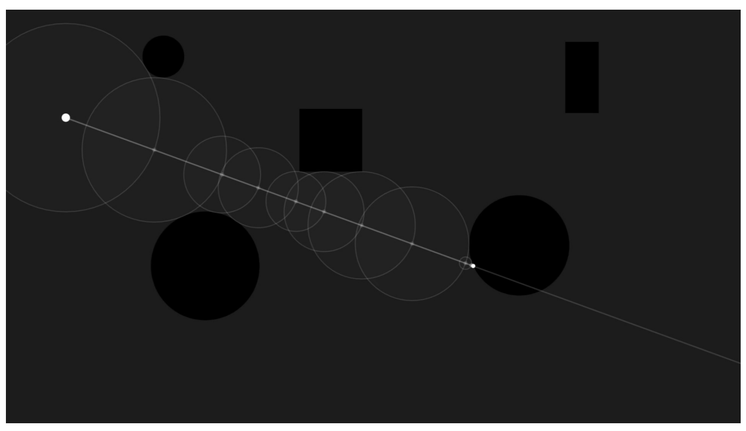


Fonte: Harlen Batagelo, Bruno Marques, 2021.

Ray Marching e *Sphere Tracing*

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 [5]. Enquanto no *Ray Tracing* a colisão dos raios com a geometria da tela é diretamente calculada por equações de intersecção, *Sphere Tracing* utiliza um processo iterativo para propagar o raio. 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.

Sphere Tracing se trata de uma forma específica de *Ray Marching* em que, para decidir o quanto um raio deve avançar em uma determinada iteração, *Ray Marching* utiliza funções de distância para descobrir o quão distante a posição atual do raio está da geometria da cena [3]. A Figura à esquerda mostra a utilização de *Sphere Tracing* para propagar um raio em uma cena. Cada um dos círculos na figura representa uma iteração do algoritmo de *Ray Marching*. A Figura à direita mostra três exemplos de cenas renderizadas utilizando *Ray Marchinig* e *Sphere Traching*.



Fonte: Fonte: Sebastian Lague, 2019.

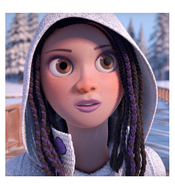
Fonte: Inigo Quilez, 2020.

Tecnologias Utilizadas

O renderizador foi desenvolvido na linguagem **Rust**, utilizando o pacote **WGPU** 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**, e os *shaders* foram compilados para a linguagem binária intermediária **SPIR-V**, que pode ser diretamente utilizada pelo WGPU. Para serialização e desserialização de dados foi utilizado majoritariamente o formato **JSON** e a biblioteca **Serde**.

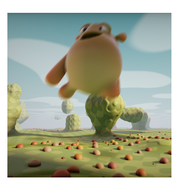
Resultados e Trabalhos Futuros

O renderizador pode ser utilizado por usuários para visualizar cenas com *Ray Marching* nativamente em tempo real. Abaixo estão algumas cenas implementadas no renderizador (todas retiradas do https://www.shadertoy.com/user/iq) e sua performance. Todas elas foram modeladas por Inigo Quilez [5] e adaptadas para serem executadas no renderizador com o objetivo de analisar sua performance na renderização de cenas mais complexas. Todas as cenas foram renderizadas na resolução de 1920x1080 pixels no mesmo computador com as seguintes especificações: CPU Intel(R) Core(TM) i5-13450HX, 24GB de RAM, GPU NVIDIA GeForce RTX 3050 Mobile, 6GB de VRAM. A performance de cada cena foi medida ao longo de 30 segundos de execução em quadros por segunds (QPS).



- Cena: *Selfie Girl*
- Performance: 45 QPS

- Cena: *Snail*
- Performance: 120 QPS



- Cena: *Happy Jumping*
- Performance: 61 QPS

- Cena: *Greek Temple*
- Performance: 97 QPS

Nenhuma dessas cenas foi criada com o objetivo de serem executadas em tempo real, utilizam geometria e efeitos de iluminação complexos, e, mesmo assim, já podem ser executadas em taxas de quadro interativas (nesse caso, consideramos taxas de quadros acima de 30 quadros por segundo como interativas). Há diversas otimizações que podem ser implementadas no renderizador para aumentar ainda mais sua performance, como *Bounding Volume Hierarchies* e a integração de *Ray Tracing* como um acelerador do processo de *Ray Marching*. Mas mesmo usando poucas otimizações relevantes, *Ray Marching* já se prova uma alternativa viável para a renderização de cenas em tempo real.

Informações

Para mais informações, visite a página do projeto e o repositório com o código desenvolvido:

- https://www.linux.ime.usp.br/~ptrschneider/mac0499/
- https://github.com/pedrotrschneider/blob-engine

Escaneie o ***QR Code*** ao lado para ver mais recursos interessantes sobre *Ray Marching*.

Referências

[1] Tomas Akenine-Möller and Eric Haines and Naty Hoffman, "**Real-Time Rendering 3rd Edition**", 2008.

[2] Michael Abrash, "**Michael Abrash's Graphics Programming Black Book**", 1997.

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

[4] Hart, J. C. and Sandin, D. J. and Kauffman, L. H., "**Ray tracing deterministic 3-D fractals**", in ACM SIGGRAPH, 23(3):289–296, 1989.

[5] Inigo Quilez "**Ray Marching Distance Fields**", in https://iquilezles.org/articles/raymarchingdf/, 2008.