Universidade Federal da Paraíba
Centro de Informática

Introdução à Computação Gráfica
Atividade Prática 5 - *Ray Tracing*

Francisco Siqueira Carneiro da Cunha Neto[1]
20190029015
[1]francisconeto@eng.ci.ufpb.br

26 de novembro de 2021

1 Atividade Desenvolvida

Nesta atividade foram realizadas extensões ao *ray tracer* disponível em https://codepen.io/ICG-UFPB/pen/BaRqvpR. Nominalmente, foram implementados: a possibilidade de renderizar de mais de um objeto na cena, brilho especular nos objetos de acordo com o modelo de iluminação de Phong [1] e uma primitiva adicional (triângulos). Além disso, uma cena foi gerada utilizando as duas primitivas disponíveis e aplicando materiais diferentes a elas.

O código fonte também pode ser encontrado em seus repositórios no GitHub e CodePen.

2 Renderizando Múltiplas Primitivas

O código disponibilizado declarava uma variável com uma esfera, e checava intersecções com essa única primitiva. A primeira mudança necessária foi substituir essa esfera por uma lista de primitivas, e inserir uma esfera idêntica nessa lista. Para cada raio, a lista é percorrida e são buscadas intersecções com cada primitiva nela.

Também foi necessário mover as variáveis relacionadas a coeficientes de iluminação da esfera para dentro da classe *Esfera*, bem como remover a parte do código que colore com preto os pixels cujos raios não possuem intersecção, já que isso sobrescreve pixels que representam objetos, renderizando apenas o último objeto da lista.

Com essas mudanças, foi possível renderizar múltiplos objetos em uma cena, como demonstrado nas Figuras 5 e 7, por exemplo.

3 Brilho Especular

Os termos ambiente e difuso já se encontravam implementados no código original, de forma que só foi necessário adicionar o cálculo do termo especular.

Para tal, foram adicionadas duas propriedades à classe *Esfera*, uma que guarda o coeficiente especular do material e outra o parâmetro n relacionado ao tamanho do brilho.

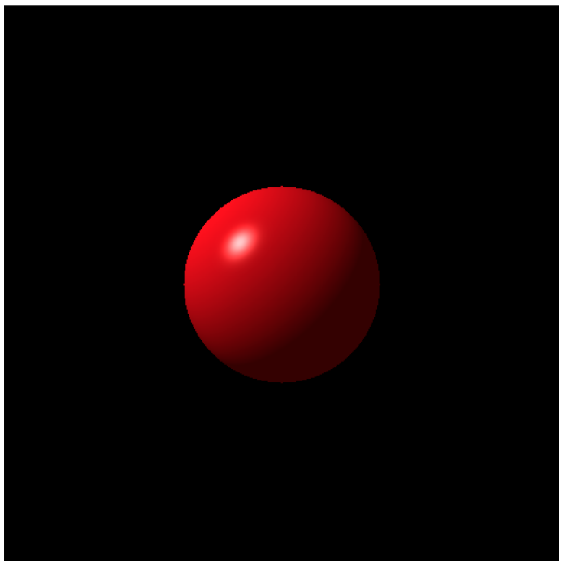

Figura 1: *Esfera com brilho especular.*

Os vetores V e R, usados no cálculo do brilho, foram derivados da seguinte forma: V é a substração de $(0, 0, 0)$ (posição da câmera) e a posição da intersecção, normalizado; R é a reflexão de L sobre a normal da intersecção, calculada usando a função `reflect` da biblioteca ThreeJS [2] e multiplicada por -1 devido a uma particularidade dessa função.

O termo especular é então definido por

$$S = I_p * k_s * max(0, R \cdot V)^n$$

Esse vetor é somado aos termos difuso e ambiente ao se escolher a cor do pixel, resultando na renderização exibida na Fig. 1.

4 O Triângulo

Para renderizar essa nova primitiva, foi criada uma nova classe *Triangle*, com as propriedades a, b e c, que representam seus vértices, assim como propriedades relacionadas ao material/iluminação, além do método que calcula uma possível intersecção com um raio.

O algoritmo utilizado nesse cálculo foi o algoritmo de Möller-Trumbore, como descrito no site *Scratchapixel* [3]. Primeiro foram declarados três vetores que formam a base do "espaço do triângulo", $T, E1$ e $E2$, sendo T a subtração da origem do raio por a do triângulo; $E1$ a subtração de b por a; $E2$ a subtração de c por a. A normal do triângulo foi calculada como $E2 \times E1$, e guardada em uma variável.

O valor do produto interno entre a direção do raio e a normal do triângulo foi guardado na variável $DdotN$, que será usada na derivação das coordenadas t, u, v do "espaço do triângulo" e para o possível *backface culling*. Este último é realizado imediatamente a seguir (buscando poupar processamento desnecessário), pela condição $DdotN \geq -0.001$ que quando alcançada faz a função retornar falso.

Outras variáveis foram declaradas com valores que são usados repetidamente no calculo de t, u, v: $DxE2$ que guarda o produto vetorial entre a direção do raio e $E2$; $TxE1$ com o produto vetorial entre T e $E1$.

Finalmente, realizou-se a derivação de t, u, v, baseado no exposto em [3]. Primeiro, foi definido $u = \dfrac{DxE2 \cdot T}{DdotN}$. Com u, já é possível checar se o raio não intersecciona o triângulo: no caso de $u \leq 0$ ou $u \geq 1$, a função retorna falso. Similarmente, calcula-se $v = \dfrac{TxE1 \cdot Dir_{raio}}{DdotN}$ e, caso $v \leq 0$, $v \geq 1$ ou $u + v \geq 1$, não há intersecção e a função retorna falso.

A partir desse ponto, sabemos que o raio intersecciona o triângulo e tudo que resta a fazer é definir os dados do objeto *intersecção*. Como u e v representam as coordenadas baricêntricas [4] do triângulo, tendo a como origem, a posição da intersecção pode ser definida por $a + u * E1 + v * E2$. A distância da câmera até a intersecção é a coordenada t, definida por $\dfrac{TxE1 \cdot E2}{DdotN}$. O normal da intersecção é definido como igual ao do triângulo; que deve então ser normalizado, mas inicialmente isso não foi feito, sendo a fonte de um erro encontrado. Finalmente, a função retorna verdadeiro.

Em seguida, um triângulo foi definido com vértices nas coordenadas $(-1.0, -1.0, -3.5), (1.0, 1.0, -3.0)$ e $(0.75, -1, -2.5)$ e um material vermelho com brilho especular branco. Além disso, três esferas cujos centros coincidem com as arestas do triângulo foram adicionadas a lista de objetos, como uma forma de avaliar se o triângulo havia sido renderizado corretamente. A Fig. 2 exibe essa primeira tentativa.

Como pode ser observado, o triângulo ficou "invertido" e completamente branco. Analisando o código, foi percebido que o vetor normal não estava normalizado, o que tende a gerar problemas na iluminação. A primeira tentativa de resolver o problema foi normalizar este vetor assim que ele é declarado, no inicio do código. Essa solução resultou no triângulo exibido na Fig. 3, que aparenta ter um brilho especular correto, mas tem tamanho reduzido e continua invertido.

Após análise mais cuidadosa foi notado que, devido a ordem em que os vértices eram passados no construtor do triângulo, o produto vetorial $E1 \times E2$ gera um vetor normal apontando para o lado contrário ao desejado. Invertendo os membros da operação para $E2 \times E1$ na definição do normal, o triângulo da Fig. 4 foi gerado. Dessa vez ele não parece mais estar invertido, mas continua menor do que esperado.

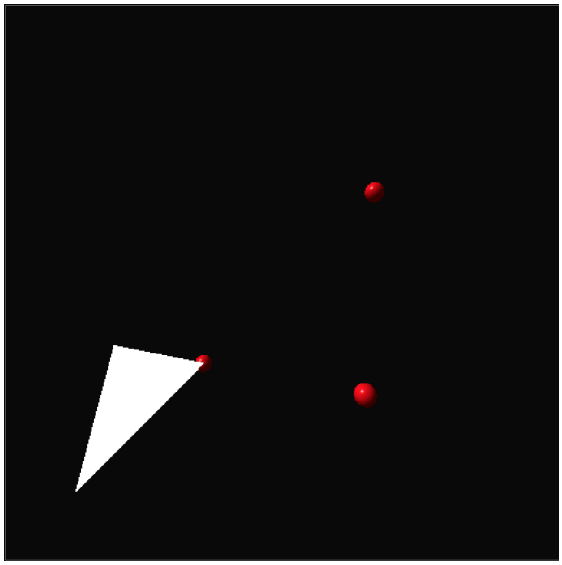

Figura 2: *Primeira tentativa de renderização de triângulos.*

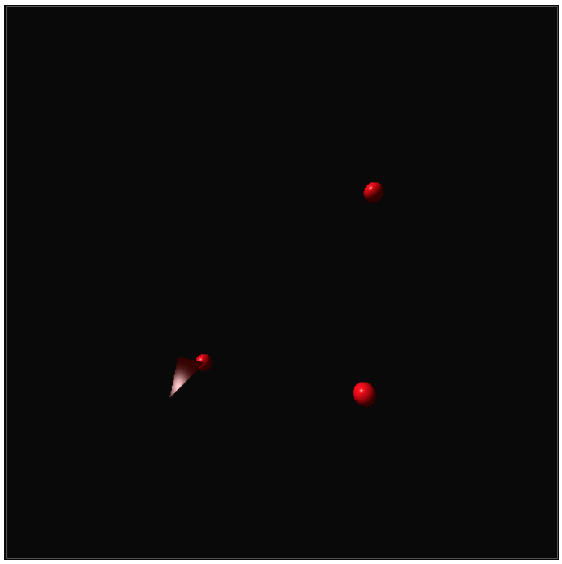

Figura 3: *Triângulo com brilho especular renderizado corretamente.*

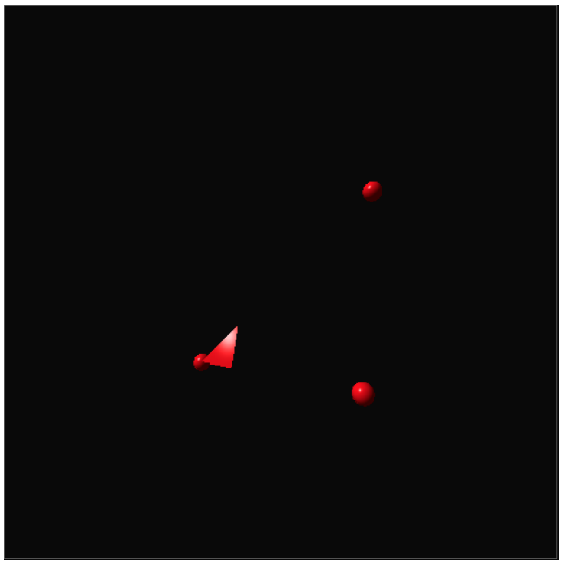

Figura 4: *Triângulo renderizado em tamanho reduzido.*

O problema estava em quando o vetor normal estava sendo normalizado. Ao usa-lo normalizado no cálculo da intersecção, o triângulo tinha o tamanho de suas arestas erroneamente encurtadas, matematicamente falando. A solução foi normalizar esse vetor apenas ao defini-lo no objeto *intersecção*. O resultado é o demonstrado na Fig. 5. A Fig. 6 mostra a primitiva renderizada exatamente como nas especificações da atividade.

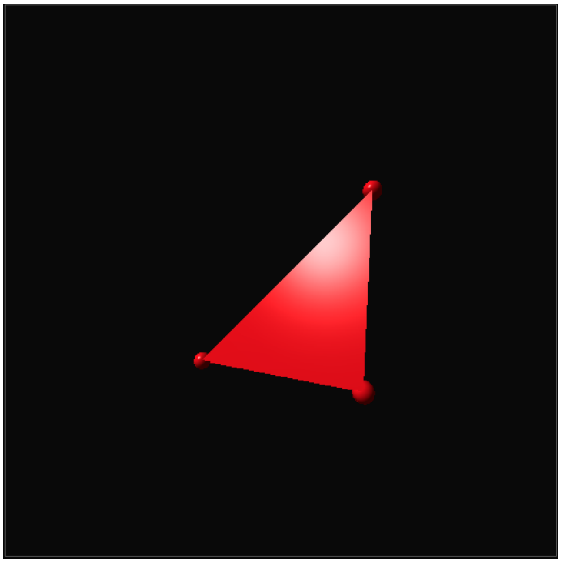

Figura 5: *Triângulo renderizado corretamente.*

Figura 6: *Triângulo renderizado corretamente.*

5 Demonstração

Finalmente, foi gerada uma cena que demonstre as duas primitivas, diferentes materiais que podem ser utilizados nelas, e o brilho especular. A cena, que modela um sorvete de casquinha, é mostrada na Fig. 7.

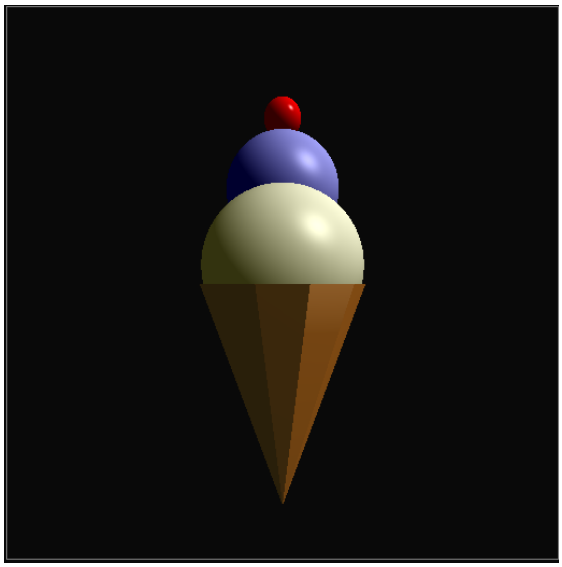

Figura 7: *Cena renderizada usando ray tracing.*

O sorvete foi criado com 6 triângulos, que formam o cone, e 3 esferas, formando o sorvete com uma cereja no topo. O material dos triângulos foi definido como o seguinte: $k_d = (1, 0.5, 0.1)$, $k_a = (0.8, 0.6, 0.2)$, $k_s = (0.4, 0.4, 0.4)$ e $n = 1$. O material das esferas, é, de baixo para cima: $k_d = (1, 0, 0)$, $k_a = (1, 0, 0)$, $k_s = (1, 1, 1)$ e $n = 32$; $k_d = (0.8, 0.8, 1)$, $k_a = (0, 0, 0.9)$, $k_s = (0.4, 0.4, 0.4)$ e $n = 16$; $k_d = (1, 1, 1)$, $k_a = (1, 1, 0.3)$, $k_s = (0.3, 0.3, 0.3)$ e $n = 16$.

Referências

[1] B. T. Phong, "Illumination for computer generated pictures," *Communications of the ACM*, v. 18, n. 6, pp. 311–317, 1975.

[2] "three.js docs - Vector3 reflect." (2021), endereço: https://threejs.org/docs/index.html?q=vec#api/en/math/Vector3.reflect (acesso em 26/11/2021).

[3] "Ray Tracing: Rendering a triangle (Möller-Trumbore algorithm)." (nov. de 2016), endereço: https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/moller-trumbore-ray-triangle-intersection.

[4] "Ray Tracing: Rendering a triangle (Barycentric Coordinates)." (nov. de 2016), endereço: https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/barycentric-coordinates.