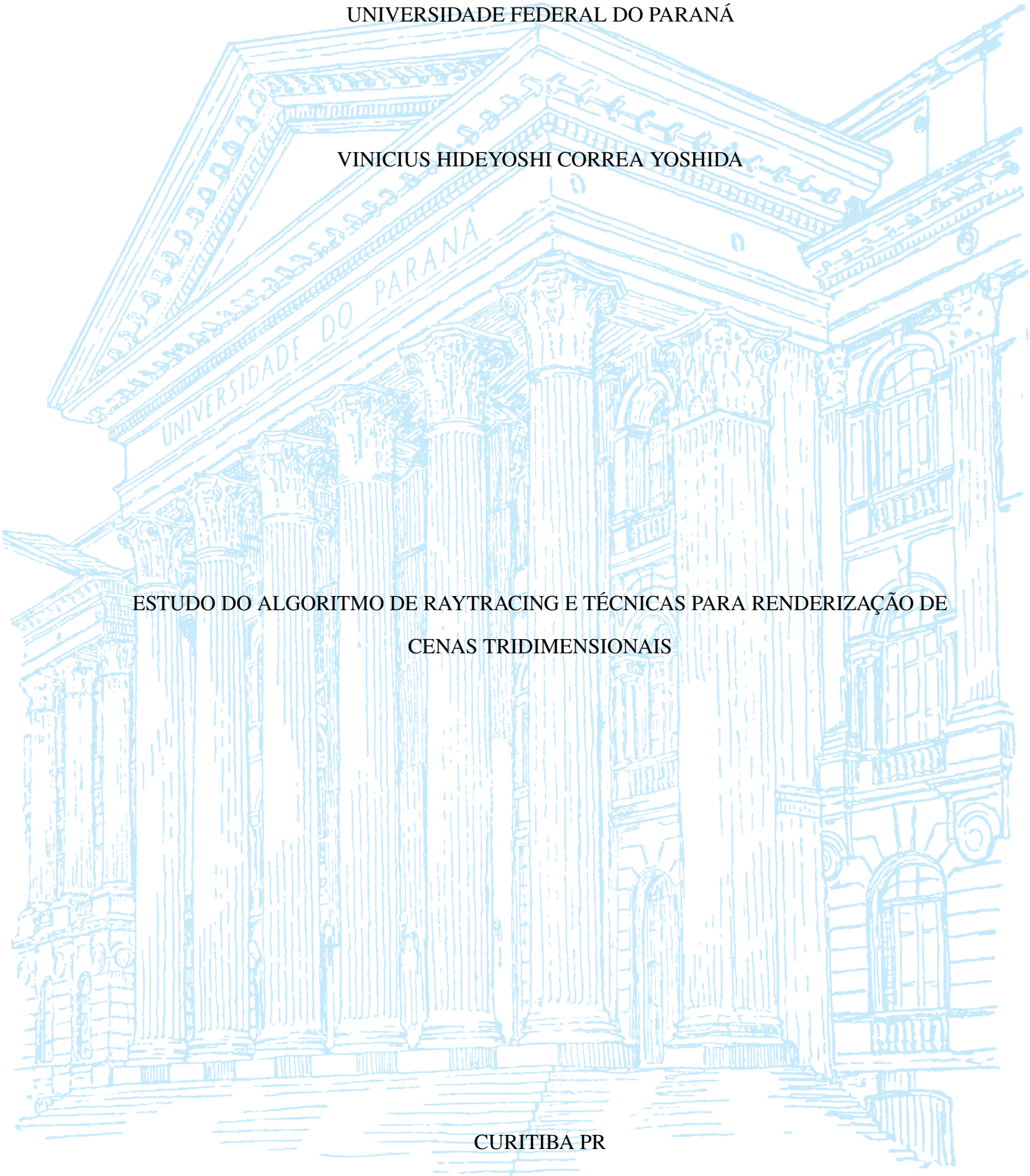
UNIVERSIDADE FEDERAL DO PARANÁ

VINICIUS HIDEYOSHI CORREA YOSHIDA



ESTUDO DO ALGORITMO DE RAYTRACING E TÉCNICAS PARA RENDERIZAÇÃO DE CENAS TRIDIMENSIONAIS

CURITIBA PR

2024

VINICIUS HIDEYOSHI CORREA YOSHIDA

ESTUDO DO ALGORITMO DE RAYTRACING E TÉCNICAS PARA RENDERIZAÇÃO DE CENAS TRIDIMENSIONAIS

Trabalho apresentado como requisito parcial à conclusão do Curso de Bacharelado em Ciência da Computação, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: André Luiz Pires Guedes.

CURITIBA PR

2024

AGRADECIMENTOS

Aos meus amigos que me acompanharam e ajudaram durante essa etapa.
Aos meus pais pelos esforços e dedicação, me permitindo melhores condições de estudo.
Aos meus avós por todo apoio, desde sempre.
Aos professores e colegas de curso.

RESUMO

O *Ray Tracing* é um algoritmo que possui história na área da computação gráfica, sendo utilizado a bastante tempo na indústria cinematográfica (Roth, 1982), porém apenas recentemente, com a evolução de *hardware*, começou a ser utilizado em aplicações de tempo real como simulações e jogos. O objetivo desse trabalho é entender melhor o funcionamento dessa técnica e os desafios relacionados à sua utilização no processador, compreendendo a teoria e aplicando ela num experimento prático.

Palavras-chave: Ray Tracing. Computação gráfica. Renderização.

ABSTRACT

The *Ray Tracing* is an algorithm with history in the computer graphics area, having been used for a long time in the film industry (Roth, 1982), but only recently, with hardware evolution, started to be used in real-time applications such as simulations and games. The objective of this work is to better understand how this technique works and the challenges related to its use in the processor, understanding the theory and applying it in a practical experiment.

Keywords: Ray Tracing. Computer graphics. Rendering.

LISTA DE FIGURAS

LISTA DE ACRÔNIMOS

RT	Ray Tracing
CGI	Computer-Generated Imagery
CPU	Central Processing Unit
GPU	Graphics Processing Unit
AABB	Axis Aligned Box Bounding
PPM	Portable Pixmap
RGB	Red Green Blue

SUMÁRIO

1 INTRODUÇÃO

Atualmente é muito comum ver algum jogo, filme ou animação com imagens geradas artificialmente por meio de CGI[1], porém, tendo uma alta qualidade parecendo até mesmo fotos reais. Existem estudos desde a década de 70 buscando por maneiras de representar no computador informações de um plano tridimensional, sendo (Goldstein e Nagel, 1971) um dos primeiros modelos de renderização utilizando a ideia conhecida como *Ray Casting*. Além dessa técnica, o contexto da computação gráfica evoluiu com a passagem do tempo e o avanço do *hardware*, permitindo novas técnicas de guardar informações em estruturas de dados e realizar uma renderização. A seguir será comentado algumas técnicas de renderização bastante conhecidas:

- Rasterização: processo pelo qual um primitivo é convertido em uma imagem bidimensional, formada pixel a pixel, com cada um deles representando uma cor.
- Imagem Vetorial: Imagem vetorial é aquela gerada a partir de linhas e pontos definidos com cálculos matemáticos. Calculando-se de novo sempre que ampliar ou diminuir tais imagens.
- Ray Marching: utiliza um traçado de raio, aplicando vários volumes sobre o raio iterativamente (é comum usar esferas), assim dividindo ele em diversos segmentos através da ideia de função de distância sinalizada.
- Path Tracing: método de renderização usado para definir traçados que deixam a iluminação da imagem mais próxima da realidade. Diferente do *Ray Tracing* e do *Ray Casting*, pois projeta os raios a partir de fontes de luz distribuídas na cena, e não da posição de observação (processo semelhante à luz real).

Para esse trabalho irá ser utilizado as técnicas de *Ray Casting* e *Ray Tracing*, em ambas são utilizados traçados de raios, mas no RT existe uma simulação do caminho inverso que um raio de luz faria, por isso os efeitos de luz criados são mais realistas pois, deixa iluminações, sombras e reflexos com aparência quase indistinguível da realidade. Entretanto, é fato que essa técnica não é barata, sendo necessário muito tempo de processamento, com até mesmo uso de *hardware* dedicado. Assim, buscando entender o funcionamento do *Ray Tracing* e o que ele faz para gerar imagens melhores, esse trabalho visa estudar o algoritmo de RT e as técnicas envolvidas nele, aplicando-se esses conhecimentos num experimento prático de criar um renderizador RT, mas com o processamento realizado somente sobre a CPU.

Esse trabalho se divide da seguinte forma: no Capítulo 2 será apresentado a parte teórica de como renderizar uma cena com o algoritmo de *Ray Tracing*, mostrando algoritmos de detecção de interseção do raio com alguma superfície, tipos de *shading*, tipos de material e características físicas no processo; no Capítulo 3 é mostrado em pseudocódigo uma maneira de implementar um programa renderizador RT, e os resultados obtidos; e o Capítulo 4 serve para considerações finais sobre o experimento realizado.

[1] Computer-generated imagery (CGI), ou Imagens geradas por computador, é a aplicação do campo da computação gráfica para efeitos especiais em arte, filmes, programas de televisão, comerciais e simuladores.

2 RENDERIZAÇÃO COM RAYTRACING

Como visto no capítulo anterior, existe uma busca por maneiras de como representar figuras e formas geométricas no computador, e posteriormente aplicar diferentes técnicas que resultam na renderização de cenas em forma de imagens ou aplicações em tempo real. Portanto, nesse capítulo será explicado o algoritmo por trás da técnica de *Ray Tracing*, utilizando como referência os livros (Shirley et al., 2009) e os materiais de apoio (Prunier, 2009), (Peter Shirley, 2024a) e (Peter Shirley, 2024b). De uma maneira simplificada, o processo consiste em disparar raios na direção de pixels posicionados em uma tela na frente de uma câmera. Esse raio então percorre pela cena tridimensional registrando as informações dos objetos interseccionados, e com isso utiliza-se uma técnica de *shading* ou sombreamento para determinar a cor do pixel (aquele na tela por onde o raio passou).

A teoria envolvida no processo de renderizar cenas com *Ray Tracing* envolve alguns conhecimentos prévios:

- O que é uma câmera numa cena tridimensional qualquer.
- Como utilizar a técnica de *Ray Casting* para renderizar superfícies geométricas
- Como descobrir quando ocorrem intersecções dos raios com as superfícies.
- Como aplicar técnicas de sombreamento sobre as superfície.

2.1 CÂMERA

A câmera representa a abstração de um observador no cenário, consiste em um ponto qualquer no espaço com uma direção sendo observada. Esse ponto representa sua posição de origem O e a direção observada é um vetor do ponto O para alguma posição do espaço tridimensional. Para realizar uma renderização, será considerado a existência de uma tela posicionada na frente da câmera, possuindo pixels que são pontos com determinadas coordenadas em duas dimensões. A Figura 2.1 representa essa ideia com um ponto de origem e e a direção sendo o centro de algum pixel da tela,

2.2 RAY CASTING

Para realizar a renderização de cenários de um espaço tridimensional, será utilizado a técnica conhecida como *Ray Casting* que consiste em, a partir de um ponto de origem, disparar raios em uma determinada direção. Primeiramente, um raio é um componente da computação gráfica que possui duas informações:

- Um ponto tridimensional o representando sua origem.
- Um vetor tridimensional $\vec{b}$ com sua direção alvo.

Um raio pode se estender infinitamente nessa direção alvo e um determinado ponto dele é obtido com algum valor escalar t pela equação 2.1.

$$P(t) = o + \vec{b}t \tag{2.1}$$

Assim, como ocorre na Figura 2.1, a câmera realiza esse procedimento, criando raios do seu ponto de origem e até um determinado pixel P da tela que se estende até o infinito e serve para capturar as informações do cenário, como determinar se houve uma interseção com algum objeto ao longo desse caminho, e eventualmente, a normal e o material desse objeto. Então, essas informações são armazenadas e posteriormente podem ser utilizadas na etapa de sombreamento para determinar a cor do pixel.

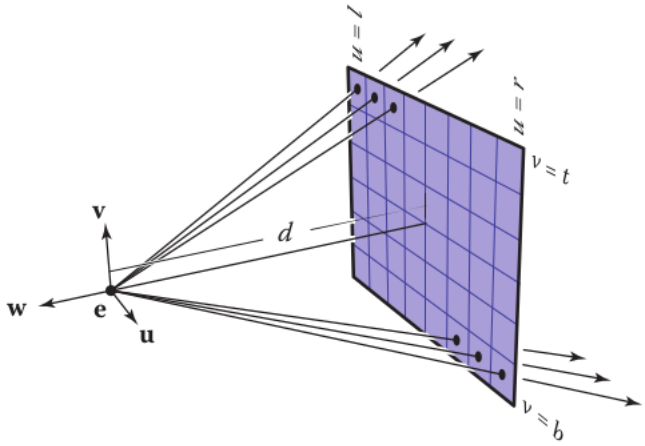


Figura 2.1: Raios traçados a partir de uma câmera, saindo de uma origem e e indo em direção a um pixel de coordenadas u e v. Imagem adaptada do livro (Shirley et al., 2009)

2.3 INTERSEÇÕES

Para descobrir informações dos objetos da cena, verifica-se a existência de uma interseção do raio com alguma superfície por meio de modelos matemáticos. Essas superfícies são formas geométricas e os objetos principais são esferas, triângulos e quadriláteros. Ademais, uma possível otimização para esse cálculo de interseção é envolver o objeto com uma caixa de dimensões proporcionais ao seu tamanho, e assim permitindo verificar se o raio atravessa essa caixa, essa otimização é a técnica *Axis Aligned Box Bounding* e também será explicada a seguir.

Para encontrar em que ponto houve essa interseção do raio com alguma superfície, iguala-se a equação do raio com a equação dessa forma geométrica, resolvendo em função de t no domínio dos reais. A seção 2.3.1 explica esse procedimento quando um raio intersecciona uma esfera; a seção 2.3.2 explica esse procedimento quando um raio intersecciona um triângulo; a seção 2.3.3 explica esse procedimento quando um raio intersecciona um quadrilátero qualquer; e por fim, a seção 2.3.4 explica um processo de otimização nessa avaliação da ocorrência de uma interseção.

2.3.1 Esferas

Para calcular a interseção de um raio com uma esfera, será utilizado a equação 2.2 que corresponde a uma esfera com centro no ponto (C_x, C_y, C_z) e raio r.

$$(x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2 = r^2 \tag{2.2}$$

Como o vetor do centro C dessa esfera até um ponto P qualquer em sua superfície corresponde a $(P - C)$, sendo $C = (C_x, C_y, C_z)$ e $P = (x, y, z)$. Isso implica em:

$$(P - C) \cdot (P - C) = (x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2$$

E portanto, a equação da esfera pode ser reescrita na forma vetorial, ficando igual a

$$(P - C) \cdot (P - C) = r^2 \tag{2.3}$$

Assim, para saber se um raio está colidindo com a esfera, existe um ponto P que satisfaz a equação 2.3, e logo pela equação 2.1 do raio, existe um t que define esse ponto de interseção. Realizando a expansão da equação da esfera em função desse ponto $P(t)$ e isolando em função de t:

$$\begin{aligned}
(P(t) - C) \cdot (P(t) - C) &= r^2 \\
((A + tb) - C) \cdot ((A + tb) - C) &= r^2 \\
(tb + (A - C) \cdot (tb + (A - C)) &= r^2 \\
t^2 b \cdot b + 2tb \cdot (A - C) + (A - C) \cdot (A - C) - r^2 &= 0
\end{aligned} \tag{2.4}$$

Como os vetores, pontos e o r da equação 2.4 são conhecidos, apenas t é desconhecido e para encontrá-lo basta resolver as raízes dessa equação do segundo grau $\alpha t^2 + \beta t + \gamma = 0$. Dessa forma, ao calcular o delta é possível descobrir se houve interseção, quando o delta for positivo significa que o raio atravessa a esfera em dois pontos, quando ele for igual a zero o raio tangencia a esfera em um único ponto, e quando ele for negativo o raio não intersecciona a esfera. Essas situações podem ser observadas na Figura 2.2.

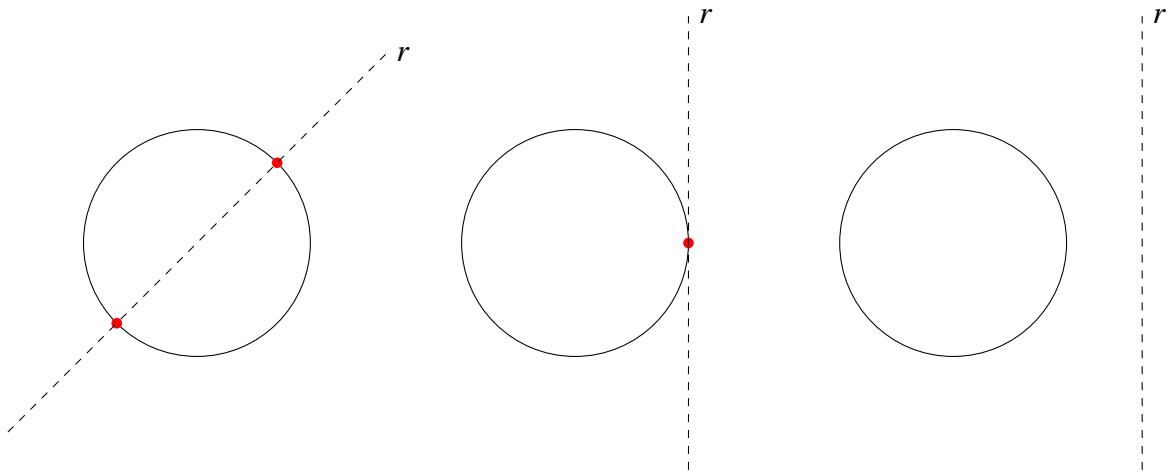


Figura 2.2: Raio cruzando esfera em três situações possíveis. Na esquerda, delta positivo e duas raízes. No meio, delta igual a zero e uma mesma raiz. Na direita, delta negativo e nenhuma raiz

O Algoritmo 1 apresenta o pseudocódigo utilizado para encontrar se existe uma interseção, guardando essa informação de t na estrutura *HitRecord*, assim como esse ponto no raio, a normal e o material da esfera (Linhas 16 a 19).

Algoritmo 1 Interseção de Raio com Esfera

Require: *ray*, *sphere*, *interval*
Ensure: *HitRecord*
1: $\overrightarrow{OC} \leftarrow ray.origin - sphere.center$ // vetor da origem do raio até o centro da esfera
2: $a \leftarrow$ **length_squared**$(ray.direction)$
3: $b_{half} \leftarrow \overrightarrow{OC} \cdot ray.direction$
4: $c \leftarrow$ **length_squared**$(\overrightarrow{OC}) - (sphere.radius)^2$
5: $discriminant \leftarrow {b_{half}}^2 - a * c$
6: **if** $discriminant < 0$ **then**
7: $\quad HitRecord \leftarrow null$
8: $\quad$ **return false**
9: **end if**
10: $sqrtd \leftarrow$ **sqrt**$(discriminant)$
11: $root \leftarrow \min\left(\frac{-b_{half} \pm sqrtd}{a}\right)$
12: **if** *root* fora de interval **then**
13: $\quad HitRecord \leftarrow null$
14: $\quad$ **return false**
15: **end if**
16: $HitRecord.t \leftarrow root$
17: $HitRecord.p \leftarrow ray$ no ponto de $HitRecord.t$
18: $HitRecord.normal \leftarrow$ **unit_vector**$\left(\frac{HitRecord.p - sphere.center}{sphere.radius}\right)$
19: $HitRecord.material \leftarrow sphere.material$
20: **return true**

2.3.2 Triângulos

Para calcular a interseção de um raio com um triângulo, foi utilizado a forma que usa coordenadas baricêntricas para um plano paramétrico que contem esse triângulo (Snyder e Barr, 1987). Seja a, b e c os vértices de um triângulo qualquer, a interseção ocorre quando um raio r está no plano do triângulo e existe um ponto $P(t)$ do raio que satisfaz a equação 2.5 abaixo para algum t, β e γ.

$$e + td = a + \beta(b - a) + \gamma(c - a) \tag{2.5}$$

Esse ponto P definido por $e + td$ estará dentro do triângulo apenas se $\beta > 0, \gamma > 0$ e $\beta + \gamma < 1$, pois eles representam coordenadas baricêntricas. Mais precisamente, existe α, β e γ e todos eles devem ser maiores que 0 e menores que 1 para indicar que o ponto P está dentro do triângulo, caso um deles seja zero o ponto se encontra numa aresta, e caso dois deles sejam zero o ponto se encontra num vértice.

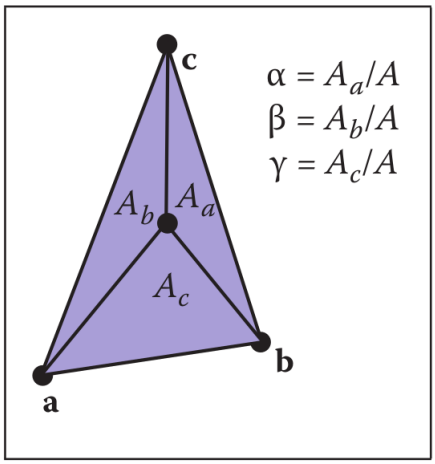


Figura 2.3: Coordenadas baricêntricas para um triângulo de vertices a, b e c. Imagem adaptada do livro (Shirley et al., 2009)

Para resolver essa equação de interseção para valores t, β e γ, expande-se ela da forma vetorial para três equações relativas a cada uma das coordenadas:

$$x_e + tx_d = x_a + \beta(x_b - x_a) + \gamma(x_c - x_a)$$

$$y_e + ty_d = y_a + \beta(y_b - y_a) + \gamma(y_c - y_a)$$

$$z_e + tz_d = z_a + \beta(z_b - z_a) + \gamma(z_c - z_a)$$

Podendo ser reescrito sem alterar a igualdade como:

$$\beta(x_a - x_b) + \gamma(x_a - x_c) + tx_d = x_a - x_e$$

$$\beta(y_a - y_b) + \gamma(y_a - y_c) + ty_d = y_a - y_e$$

$$\beta(z_a - z_b) + \gamma(z_a - z_c) + tz_d = z_a - z_e$$

E sobrescrevendo como um sistema linear:

$$A \cdot \vec{x} = \vec{b}$$

$$\begin{bmatrix} x_a - x_b & x_a - x_c & x_d \\ y_a - y_b & y_a - y_c & y_d \\ z_a - z_b & z_a - z_c & z_d \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} x_a - x_e \\ y_a - y_e \\ z_a - z_e \end{bmatrix}$$

Utilizando a regra de Cramer para resolver esse sistema linear 3×3, a solução para cada valor seria:

$$\beta = \frac{\begin{vmatrix} x_a - x_e & x_a - x_c & x_d \\ y_a - y_e & y_a - y_c & y_d \\ z_a - z_e & z_a - z_c & z_d \end{vmatrix}}{|A|}, \gamma = \frac{\begin{vmatrix} x_a - x_b & x_a - x_e & x_d \\ y_a - y_b & y_a - y_e & y_d \\ z_a - z_b & z_a - z_e & z_d \end{vmatrix}}{|A|}, t = \frac{\begin{vmatrix} x_a - x_b & x_a - x_c & x_a - x_e \\ y_a - y_b & y_a - y_c & y_a - y_e \\ z_a - z_b & z_a - z_c & z_a - z_e \end{vmatrix}}{|A|}$$

Utilizando o Algoritmo 2 é possível calcular os determinantes das matrizes e encontrar os valores para β, γ e t. Então, verifica-se se t está em um intervalo $[t_0, t_1]$ possível, caso não esteja, considera-se que o raio não está cruzando o triângulo. Caso contrário, verifica-se se γ está entre 0 e 1, e se β está entre 0 e $1 - \gamma$, caso todos esses limites sejam respeitados uma interseção foi

detectada no ponto $P(t)$. Também no final do algoritmo, essas informações são guardadas numa estrutura *HitRecord*.

Algoritmo 2 Interseção de Raio com Triângulo

Require: *ray*, *triangle*, *interval*
Ensure: *HitRecord*
1: a,b e c são os vértices do triângulo *triangle*
2: $A_{3,3} \leftarrow \begin{bmatrix} x_a - x_b & x_a - x_c & x_d & x_{ray.direction} \\ y_a - y_b & y_a - y_c & y_d & y_{ray.direction} \\ z_a - z_b & z_a - z_c & z_d & z_{ray.direction} \end{bmatrix}$
3: $rvec_{1,3} \leftarrow \begin{bmatrix} x_a - x_{ray.origin} & y_a - y_{ray.origin} & z_a - z_{ray.origin} \end{bmatrix}$
4: $ei\_minus\_hf \leftarrow A[1][1] * A[2][2] - A[1][2] * A[2][1]$
5: $gf\_minus\_di \leftarrow A[0][2] * A[2][1] - A[0][1] * A[2][2]$
6: $dh\_minus\_eg \leftarrow A[0][1] * A[1][2] - A[1][1] * A[0][2]$
7: $ak\_minus\_jb \leftarrow A[0][0] * rvec[1] - rvec[0] * A[1][0]$
8: $jc\_minus\_al \leftarrow rvec[0] * A[2][0] - A[0][0] * rvec[2]$
9: $bl\_minus\_kc \leftarrow A[1][0] * rvec[2] - rvec[1] * A[2][0]$
10: $detA \leftarrow A[0][0] * ei\_minu\_hf + A[1][0] * gf\_minus\_di + A[2][0] * dh\_minus\_eg$
11: **if** $detA = 0$ **then**
12: $\quad HitRecord \leftarrow null$
13: $\quad$ **return false**
14: **end if**
15: $t \leftarrow -\left(\frac{A[2][1] * ak\_minus\_jb + A[1][1] * jc\_minus\_al + A[0][1] * bl\_minus\_kc}{detA}\right)$
16: $\gamma \leftarrow \left(\frac{A[2][2] * ak\_minus\_jb + A[1][2] * jc\_minus\_al + A[0][2] * bl\_minus\_kc}{detA}\right)$
17: $\beta \leftarrow \left(\frac{rvec[0] * ei\_minus\_hf + rvec[1] * gf\_minus\_di + rvec[2] * dh\_minus\_eg}{detA}\right)$
18: **if** t fora de interval **or** $\gamma \notin (0,1)$ **or** $\beta \notin (0, 1-\gamma)$ **then**
19: $\quad HitRecord \leftarrow null$
20: $\quad$ **return false**
21: **end if**
22: $HitRecord.t \leftarrow t$
23: $HitRecord.p \leftarrow ray$ no ponto de $HitRecord.t$
24: $HitRecord.normal \leftarrow$ **unit_vector**$((b-a) \times (c-a))$
25: $HitRecord.material \leftarrow triangle.material$
26: **return true**

2.3.3 Quadriláteros

Pela definição um quadrilátero seria equivalente a um paralelogramo, possui um ponto Q como vértice de origem e dois vetores $\vec{u}$ e $\vec{v}$ partindo dele, e para obter os outros três vértices somam-se esses vetores em relação à origem, $R = Q + \vec{v}$, $S = Q + \vec{u}$ e $T = Q + \vec{u} + \vec{v}$. Para calcular a interseção de um raio com um quadrilátero é necessário:

- Descobrir qual plano contém esse quadrilátero.
- Encontrar se houve interseção do raio com esse plano.
- Determinar se o ponto que interseccionou o plano está dentro do quadrilátero.

O plano que contém o quadrilátero pode ser encontrado utilizando-se a fórmula implícita dos planos, onde A, B, C, D são apenas constantes, e x, y, z são os valores de qualquer ponto

(x, y, z) que esteja no plano. Um plano é, portanto, o conjunto de todos os pontos (x, y, z) que satisfazem a equação 2.6.

$$Ax + By + Cz + D = 0 \tag{2.6}$$

Considerando-se o plano perpendicular ao vetor normal $n = (A, B, C)$, e o vetor da origem para qualquer ponto no plano $v = (x, y, z)$, então podemos usar o produto escalar entre eles para resolver em função de D:

$$Ax + By + Cz = D$$

$$n \cdot v = D$$

Então, o vetor posição v será substituído pela equação 2.1 do raio como $v = P + td$, de maneira que a igualdade pode ser resolvida em função de t.

$$n \cdot (P + td) = D$$

$$n \cdot P + n \cdot td = D$$

$$n \cdot P + t(n \cdot d) = D$$

$$t = \frac{D - n \cdot P}{n \cdot d} \tag{2.7}$$

Portanto, com a equação 2.7 já é possível determinar se o raio está interseccionando o plano, e caso $n \cdot d$ seja 0, ele está paralelo ao plano, e consequentemente também não está interseccionando o quadrilátero.

Sabendo que o raio está interseccionando esse plano, agora é necessário verificar se o ponto P dessa interseção está dentro dos limites do quadrilátero, como observado na Figura 2.4. Assim como no caso da interseção do raio com um triângulo, é possível expressar esse ponto P em função de um sistema de coordenadas seguindo a equação 2.8.

$$P = Q + \alpha(u) + \beta(v) \tag{2.8}$$

Novamente, o ponto P sendo aquele definido pela equação 2.1 do raio, aqui como $P(t) = e + tb$:

$$e + tb = Q + \alpha(u) + \beta(v) \tag{2.9}$$

Os valores da equação 2.9 são tais que:

$$\alpha = w \cdot (p \times v)$$

$$\beta = w \cdot (u \times p)$$

$$p = P - Q$$

$$w = \frac{n}{n \cdot n}$$

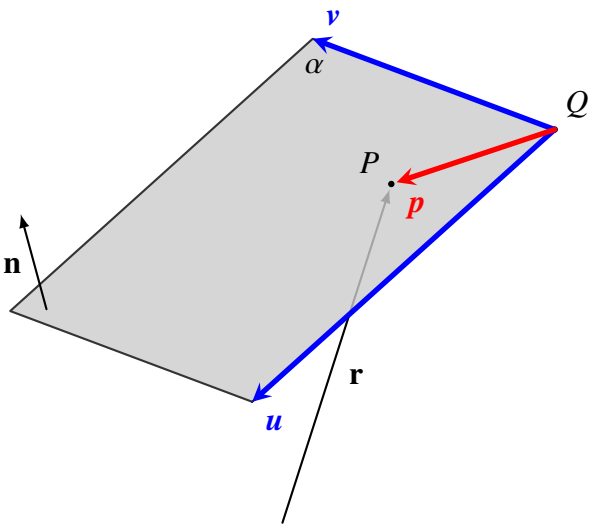


Figura 2.4: Raio cruzando um quadrilátero no ponto P

Com o Algortimo 3 a interseção de um raio com um quadrilátero pode ser encontrada, verifica-se se t está em um intervalo $[t_0, t_1]$ possível, e caso não esteja, considera-se que o raio não está cruzando o plano contendo essa forma geométrica. Porém, com um t válido, encontramos um ponto P dentro do plano que também pertence ao quadrilátero se α e β estiverem dentro do intervalo (0, 1). Com isso, no fim do algoritmo, as informações sobre essa interseção são guardadas dentro da estrutura *HitRecord*.

Algoritmo 3 Interseção de Raio com Quadrilátero

Require: *ray*, *quad*, *interval*
Ensure: *HitRecord*
1: $denom \leftarrow quad.normal \cdot ray.direction$
2: **if** *denom* igual ou aproximadamente 0 **then**
3: $\quad HitRecord \leftarrow null$
4: $\quad$ **return false**
5: **end if**
6: $t \leftarrow \dfrac{(D - quad.normal \cdot ray.origin)}{denom}$
7: **if** t fora de interval **then**
8: $\quad HitRecord \leftarrow null$
9: $\quad$ **return false**
10: **end if**
11: $intersection \leftarrow ray$ no ponto t
12: $p \leftarrow intersection - Q$
13: $\alpha \leftarrow w \cdot (p \times v)$
14: $\beta \leftarrow w \cdot (u \times p)$
15: **if** $\alpha \notin (0, 1)$ **or** $\beta \notin (0, 1)$ **then**
16: $\quad HitRecord \leftarrow null$
17: $\quad$ **return false**
18: **end if**
19: $HitRecord.t \leftarrow t$
20: $HitRecord.p \leftarrow intersection$
21: $HitRecord.normal \leftarrow quad.normal$
22: $HitRecord.material \leftarrow quad.material$
23: **return true**

2.3.4 AABB

O AABB abreviação de *Axis Aligned Box Bounding* é uma forma de acelerar o processo de descobrir interseções entre um raio com algum dos objetos geométricos acima, funciona dividindo o espaço tridimensional em *Bounding Boxes* e caso o raio passe por dentro desse volume, definido pelo intervalo dos três eixos, considera-se que houve uma interseção com o objeto nele contido. Dessa forma, só verifica se tem interseção com um objeto quando sua caixa em volta tiver sendo atravessada pelo raio.

2.4 SHADING

Ao descobrir o ponto em que o raio interseccionou algum objeto, sabe-se que existe uma superfície visível interagindo com o ambiente e, portanto, é possível avaliar qual valor o pixel deverá assumir, o método que computa essa avaliação é chamado de *shading model* ou *modelo de sombreamento* em tradução literal. Existem vários modelos de *shading*, mas em geral todos têm o objetivo de capturar o processo de reflexão da luz, sua interação com a superfície de determinado objeto, e quanto de luz é refletido para uma câmera observadora. Em modelos simples de *shading*, as principais componentes para a reflexão da luz são a direção que o ponto de iluminação se encontra, a direção da câmera e a normal da superfície analisada. A seguir, será explicado alguns exemplos de *shading*, a seção 2.4.1 descreve uma forma de sombreamento utilizando a normal; a seção 2.4.2 uma forma que usa somente o material da superfície; a seção 2.4.3 apresenta o sombreamento proposto por Lambert; a seção 2.4.4 apresenta o sombreamento proposto por Blinn e Phong; e por fim a seção 2.4.5 incrementa a existência de uma iluminação ambiente ao sombreamento de Blinn-Phong.

2.4.1 Normal Shading

Uma maneira direta de colorir o pixel é utilizando o vetor unitário $\vec{n}$, a normal da superfície onde houve a interseção, atribuindo uma cor para cada eixo dele, o valor de x representa o vermelho; o valor de y o verde; e o valor de z o azul. Um possível resultado pode ser observado na Figura 2.5.

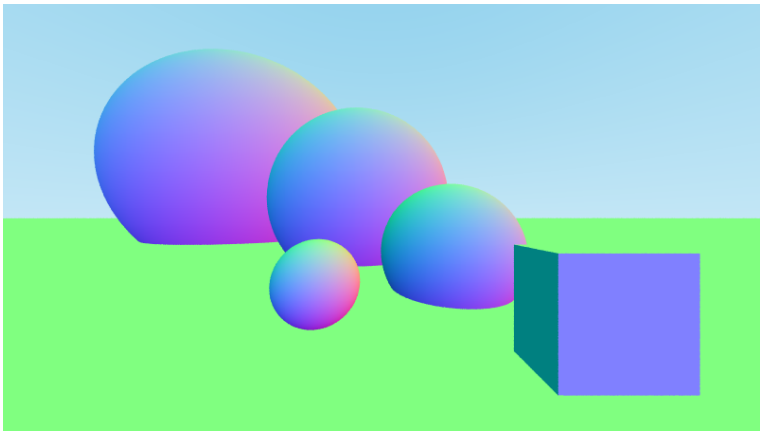


Figura 2.5: Esferas e cubo em um plano avaliado pelo modelo Normal Shading

2.4.2 Solid Color Shading

Outra maneira direta de colorir o pixel é utilizando somente o material atribuído ao objeto interseccionado pelo raio, sem levar em conta normal, iluminação e posição em relação a câmera. Um possível resultado pode ser observado na Figura 2.6.

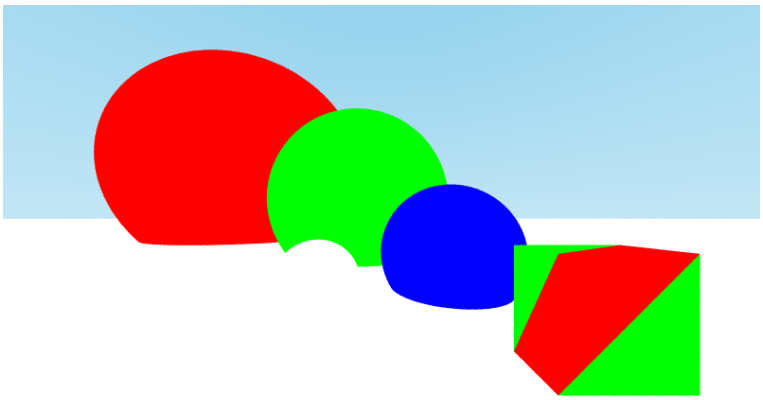


Figura 2.6: Esferas e cubo em um plano avaliado pelo modelo Solid Color Shading

2.4.3 Lambertian Shading

Esse modelo foi baseado nas observações de Johann Heinrich Lambert, descrevendo como a energia da iluminação de uma fonte luminosa é consequência do ângulo dessa superfície em relação à luz. Uma superfície qualquer que esteja exatamente embaixo da luz recebe iluminação máxima, se sua face estiver voltada para longe da direção da luz ela não recebe iluminação. Caso a superfície esteja entre esses extremos, a iluminação é proporcional ao cosseno do ângulo θ entre a normal n da superfície e a direção da luz l, como observado na Figura 2.7. A cor do pixel nesse modelo pode ser encontrado através da equação 2.10.

$$L = k_d I \max(0, n \cdot l) \tag{2.10}$$

A cor resultante do pixel será L, k_d é o coeficiente de difusão (ou cor da superfície desse objeto), I é a intensidade da fonte de luz, e o $\cos\theta$ pode ser escrito como $n \cdot l$, pois ambos são vetores unitários. A figura 2.7 apresenta a geometria por trás desse modelo de *shading*, sendo l um vetor unitário com origem no ponto de interseção indo em direção à luz e n a normal da superfície. Um possível resultado pode ser observado na Figura 2.8.

2.4.4 Blinn-Phong Shading

No modelo de Lambert da seção 2.4.3 a cor da superfície é independente do observador e, portanto, não é influenciada pela posição da câmera, mas geralmente uma superfície, na realidade, possui um certo ponto de destaque quando exposta a uma fonte luminosa, esse grau de brilho é chamado de *specular reflection*. Sem essa característica especular na superfície ela fica com uma aparência fosca, como é possível observar no Lambertian *shading*, porém isso se resolve adicionando o *specular reflection* nele, assim o modelo de *shading* passa a ter a componente difusa e a nova componente especular. Assim, esse modelo proposto por Bui Tuong Phong (Phong, 1975) e posteriormente atualizada por Jim Blinn (Blinn e Newell, 1976) utiliza a ideia da reflexão ser mais brilhante quando o observador e a luz estão simetricamente

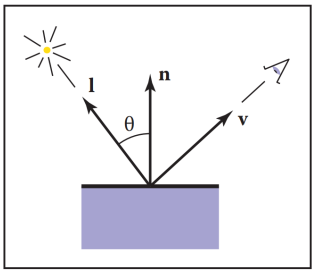


Figura 2.7: Geometria do Lambertian Shading. Imagem adaptada do livro (Shirley et al., 2009)

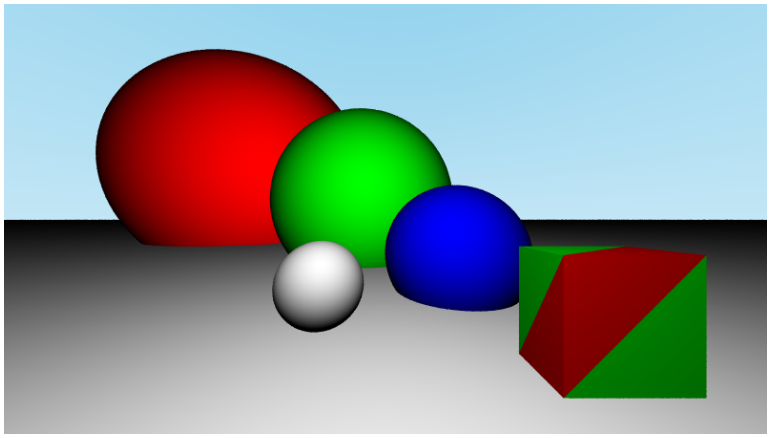


Figura 2.8: Esferas e cubo em um plano avaliado pelo modelo Lambertian Shading

posicionados sobre a normal da superfície (Figura 2.9), que é quando iria ocorrer uma reflexão de espelho. Pela mesma razão, a reflexão gradativamente diminui à medida que esses vetores se distanciam, afastando-se da configuração espelho.

Para determinar o quanto se está próximo dessa configuração espelho, compara-se o vetor bissetriz h entre l e v, caso esteja próximo da normal n a componente especular é brilhante, e caso esteja afastada a componente será proporcionalmente mais ofuscada. A equação 2.11 serve para encontrar a cor do pixel nesse modelo.

$$h = \frac{v + l}{\|v + l\|},$$
$$L = k_d I \max(0, n \cdot l) + k_s I \max(0, n \cdot h)^p \tag{2.11}$$

Onde h é a bissetriz entre l e v, k_s o coeficiente especular da superfície, o produto escalar $n \cdot h$ é o $\cos \alpha$, e p a expoente de Phong que controla o brilho aparente da superfície. Um possível resultado do *shading* de Blinn-Phong pode ser observado na figura 2.10.

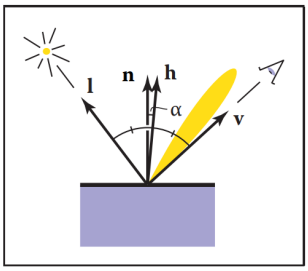


Figura 2.9: Geometria do Blinn-Phong Shading. Imagem adaptada do livro (Shirley et al., 2009)

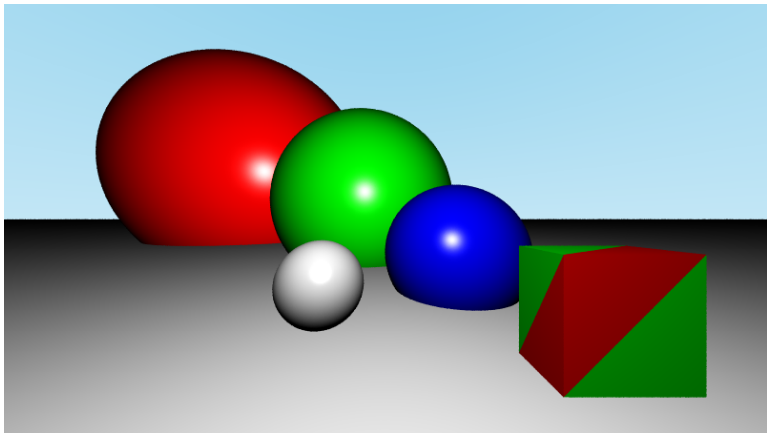


Figura 2.10: Esferas e cubo em um plano avaliado pelo modelo Blinn-Phong Shading

2.4.5 Ambient Shading

Superfícies que não recebem nenhuma iluminação são geralmente renderizadas como totalmente pretas (Shirley et al., 2009), uma maneira de evitar essa situação é adicionar outra

componente à equação 2.11, que adiciona diretamente ao pixel a cor do objeto interseccionado, sem levar em conta a geometria de sua superfície. Esse modelo é chamado de *Ambient Shading* e é calculado através da equação 2.12.

$$L = k_a I_a + k_d I \max(0, n \cdot l) + k_s I \max(0, n \cdot h)^p \tag{2.12}$$

Sendo k_a o coeficiente ambiente da superfície, ou "cor ambiente", e I_a é a intensidade dessa luz ambiente. Dessa forma, buscando replicar o que ocorre na realidade, o comportamento de superfícies não iluminadas serem clareadas pela luz indireta refletida de superfícies iluminadas está adicionado. Um possível resultado pode ser observado na figura 2.11.

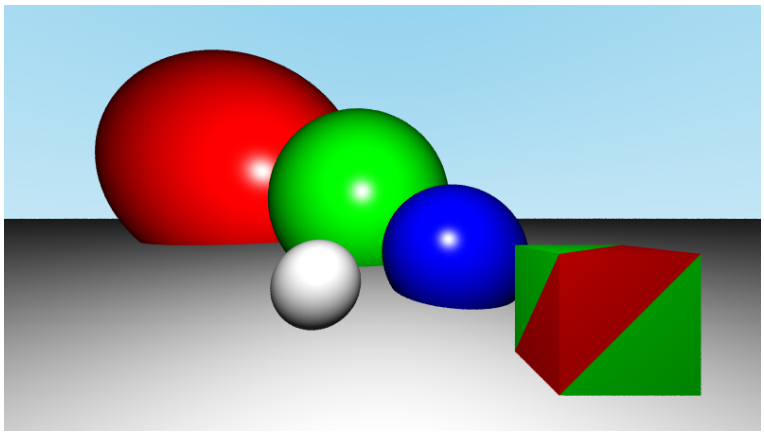


Figura 2.11: Esferas e cubo em um plano avaliado pelo modelo Ambient Shading

2.5 RAY TRACING

Os modelos de *shading* anteriores são úteis e rápidos de serem calculados, porém não levam em consideração todos aspectos da iluminação, reflexão e refração que ocorrem na realidade. Então, o algoritmo envolvido na técnica de *Ray Tracing* serve para acrescentar esses detalhes visuais na imagem gerada. Para descobrir a cor do pixel é necessário várias iterações de raios com o cenário, como eles refletem ou refratam quando ocorre uma interseção com alguma superfície e a quantidade de luz que recebem. Assim, nesse método existe um raio inicial da mesma forma que antes, origem na posição da câmera e indo em direção a algum pixel da tela, mas quando houver a interseção com alguma superfície em um ponto P dispara-se um raio de direção nova que dispersa pela cena e tem chance de colidir com outra superfície. Esse processo de gerar um novo raio sempre que ocorrer uma interseção pode ocorrer infinitamente caso a cena seja um ambiente fechado, portanto é comum utilizar um inteiro *profundidade* de limitante para a iteração não ser infinita, com uma *profundidade* de 10 o raio do pixel será dispersado, criando novos raios, até no máximo 10 vezes pela cena.

Além do material difuso existente, será acrescentado três materiais que adicionam novos comportamentos para as superfícies:

- O material *metálico* com a capacidade de refletir o raio, causando reflexos e podendo criar superfícies refletores como um espelho. Sendo apresentado na seção 2.5.1.
- O material *dielétrico* em que o raio refrata ao passar de um meio físico para o outro, como ocorre na realidade ao observar objetos de vidro ou que estejam numa piscina (envolvendo refração do ar para água). Sendo apresentado na seção 2.5.2.

- O material *emissor* que torna sua superfície em uma fonte luminosa. Sendo apresentado na seção 2.5.3.

2.5.1 Reflexão

O processo de adicionar o material refletor é direto, adiciona-se a reflexão especular ideal(reflexão de espelho), como ocorre na situação da figura 2.12, o observador ao olhar para um espelho perfeito da direção *e* enxerga o que está na direção *r* como se estivesse vendo da superfície.

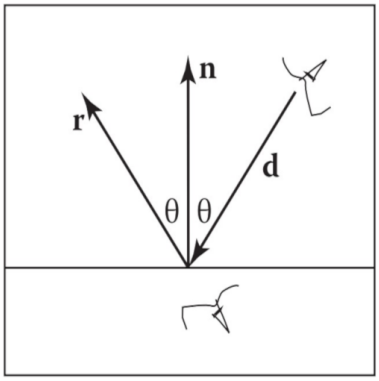


Figura 2.12: Reflexão ao olhar para um espelho perfeito. O observador olhando na direção *d* verá qualquer coisa que o observador "abaixo da superfície"iria ver na direção *r*. Imagem adaptada do livro (Shirley et al., 2009)

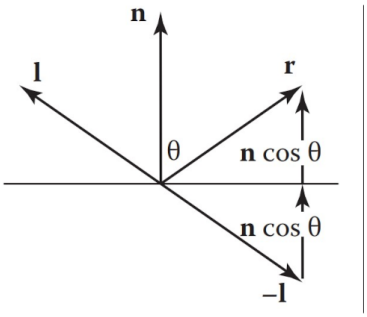


Figura 2.13: Geometria para calcular vetor *r*. Imagem adaptada do livro (Shirley et al., 2009)

Esse vetor *r* é encontrado através da equação 2.13, uma variante da reflexão de iluminação de Phong, que se baseia na observação da geometria presente na figura 2.13.

$$r = d - 2(d \cdot n)n \tag{2.13}$$

Entretanto, nessa reflexão o material pode não refletir sempre como um espelho perfeito, deslocando a direção do raio *r* em uma certa quantidade, e assim gerando uma aparência de lustrosidade. Esse desvio pode ser realizado com um valor escalar chamado de rugosidade (ou *fuzz*) e geralmente se encontra entre 0 e 1, quanto mais próximo de 0 mais parecido com um espelho, por outro lado, quanto mais próximo de 1 mais lustroso o material fica.

2.5.2 Refração

Outro tipo de objeto especular é o *dielétrico*, um material com característica transparente que refrata a luz e também a reflete em determinadas condições, materiais como vidro, água, diamante e ar são dielétricos. Enquanto um raio refletido colide com uma superfície e espalha em uma nova direção, um raio refratado altera sua direção ao passar de um material para outro, como é observado num lápis com metade dentro da água que parece estar quebrado. A quantidade que um raio refratado se curva é determinada pelo índice refrativo do material, um valor calculado pela quantidade de luz que altera de direção ao sair do vácuo e entrar no material.

O processo de refração é descrito pela equação 2.14 proveniente da Lei de Snell, onde θ e ϕ são os ângulos em relação à normal da superfície, e η e η' são os índices refrativos dos materiais.

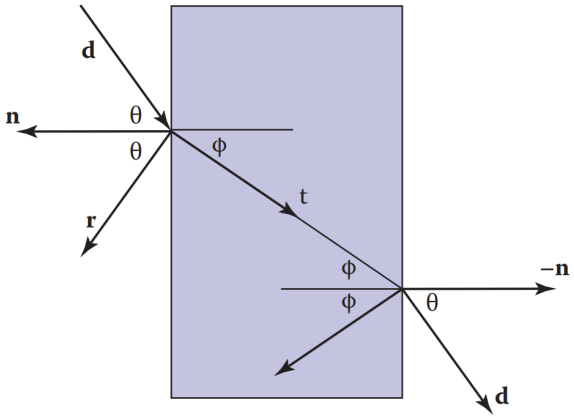


Figura 2.14: Geometria de uma superfície dielétrica, onde t é o vetor que representa o raio refratado dentro do material dielétrico. Imagem adaptada do livro (Shirley et al., 2009)

$$\eta \sin\theta = \eta' \sin\phi \tag{2.14}$$

Para saber a direção do raio refratado, utiliza-se a geometria apresentada na Figura 2.14, e é necessário encontrar o ângulo ϕ que é obtido isolando a equação 2.14 em função de $\sin\phi$. Então, para determinar o comportamento que o raio refratado t irá trazer para o pixel, separa-se dele suas partes perpendicular e paralela em relação à normal inversa da superfície $-n$, obtendo respectivamente, $t_\perp$ e $t_\parallel$. Sendo

$$t = t_\perp + t_\parallel$$

$$t_\perp = \frac{\eta}{\eta'}(d + \cos\theta n)$$

$$t_\parallel = -\sqrt{1 - |t_\perp|^2 n}$$

Assim, como a normal da superfície n e o raio incidente d são ambos vetores normalizados, o cosseno de θ é obtido pelo produto escalar entre $-d$ e n.

$$\cos\theta = -d \cdot n$$

$$t_\perp = \frac{\eta}{\eta'}(d + (-d \cdot n)n)$$

Quando um raio incidente de um meio com índice refrativo η passa para outro meio de índice refrativo η', a solução pode não ser possível de ser encontrada através da Lei de Snell caso o valor de η' seja menor que η, pois isso só seria possível se o $\sin\phi$ fosse maior que 1. Essa situação é conhecida como reflexão interna total, um exemplo desse comportamento ocorre na figura 2.15, e essa reflexão que pode chegar na mesma de um espelho perfeito é obtido através das equações de Fresnel, porém por ser uma fórmula complexa, existe uma maneira na computação gráfica mais simples e eficiente de conseguir isso utilizando a aproximação de Schlick:

$$R(0) = R_0 + (1 - R_0)(1 - \cos\theta)^5$$

Onde R_0 é a refletância na normal incidente:

$$R_0 = \left(\frac{\eta' - 1}{\eta' + 1}\right)^2$$

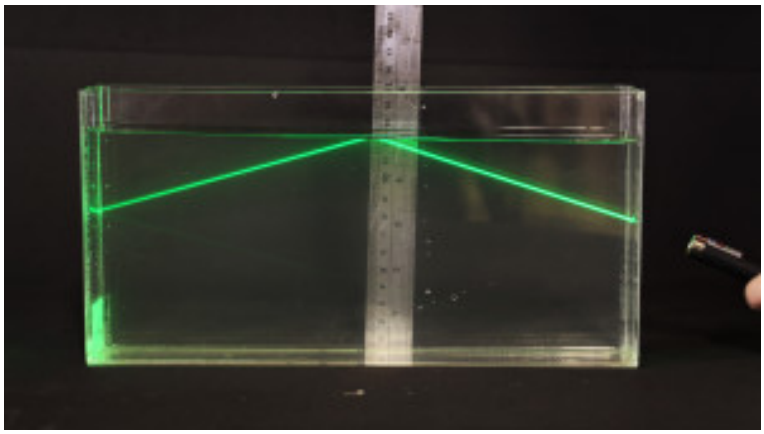


Figura 2.15: Reflexão interna total na água. Imagem do Departamento de Física da UFMG

2.5.3 Luz e Sombras

Uma maneira de representar sombras criadas pela luz é considerá-la um ponto luminoso no espaço, e para saber se um determinado ponto na cena está sendo iluminado cria-se um raio com origem nesse ponto da superfície indo na direção da luz, caso não tenha interseção o objeto está sendo iluminado, porém, caso contrário se trata de um ponto na sombra, pode-se observar essa situação na figura 2.16. Esse raio que sai de um ponto qualquer p e vai em direção à fonte luminosa I é chamado de raio sombra.

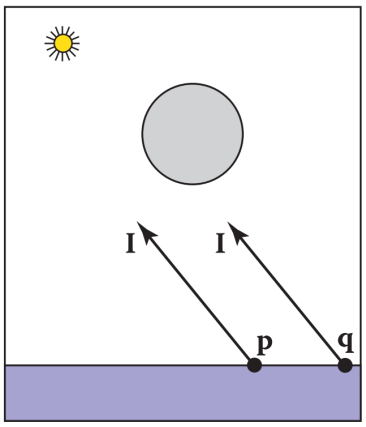


Figura 2.16: Geometria de um raio sombra. Onde o raio com origem no ponto p não se encontra na sombra, mas já o raio saindo do ponto q se intersecciona com um objeto e portanto está na sombra. Imagem adaptada do livro (Shirley et al., 2009)

O problema dessa abordagem é que a luz real não é um único ponto, e sim pontos infinitesimais, logo não existe apenas um vetor responsável por iluminar um ponto qualquer p, o mais correto é uma fonte de luz com vários raios luminosos partindo ao longo dela e viajando em várias direções pelo ambiente. Esses raios são tanto visíveis quanto invisíveis, o que produz o efeito conhecido como *penumbra* que é quando a direção da luz é parcialmente visível de um certo ponto, quando essa direção é totalmente bloqueada o objeto se encontra inteiro na sombra ou *umbra*, e a região quando o ponto está totalmente iluminado pode ser chamada de *anti-umbra*, tal fenômeno é melhor visualizado através da figura 2.17.

Portanto, será utilizado outra abordagem consistindo em criar uma nova componente luminosa, que pode ser atribuída a superfícies como se fosse um novo material. Quando um raio

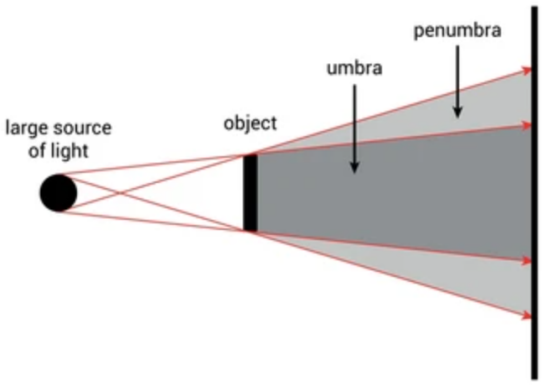


Figura 2.17: Uma fonte de luz com pontos infinitesimais iluminando um objeto qualquer, a região com a luz totalmente invisível se chama umbra, já quando ela está tanto parcialmente visível quanto invisível se chama penumbra

interseccionar a superfície contendo esse material *emissor*, calcula-se a cor do pixel levando em conta sua intensidade luminosa I, e assim o resultado da luz acaba sendo mais preciso, pois agora a fonte de luz possui uma área com vários pontos de iluminação.

2.6 CONCLUSÃO

Utilizando essa teoria por trás do algoritmo de *Ray Tracing* apresentado em (Shirley et al., 2009) é possível renderizar uma cena básica com uma câmera, determina-se uma posição qualquer para ela e uma direção a ser observada, na frente dela se posiciona uma tela de pixels, criada a partir de suas dimensões horizontais e verticais. Então, para cada pixel dessa tela dispara-se um raio da posição de origem da câmera até uma coordenada dentro desse pixel, que percorre pela cena verificando se intersecciona com qualquer outro objeto, e caso encontre, uma estrutura contendo as informações dessa interseção é guardada para a etapa de *shading*. Nessa etapa de *shading* ou sombreamento do pixel, utilizando as informações da interseção, também se descobre se o raio continuará espalhando pelo cenário, se irá se refletir ou refratar, e em qual direção, e assim coletando mais informações por onde percorrer. Dessa forma, depois desse raio iterar pela cena, a cor que esse pixel deverá assumir pode ser calculada, esse processo se repete para cada pixel e no final dessa renderização uma imagem da cena observada é gerada.

No Capítulo 3 será explicado como essa teoria pode ser aplicada para renderizar cenas mediante um programa executável criado, possuindo algumas das superfícies geométricas descritas e os materiais também apresentados.

3 EXPERIMENTO

Para colocar a teoria da técnica de *Ray Tracing* em prática, um programa executável foi criado, consistindo em criar uma cena com algumas superfícies geométricas em que cada uma delas possui um material com algum comportamento escolhido. Como o principal objetivo era compreender os algoritmos envolvidos nesse processo, o foco desse experimento foi utilizar apenas o processador para renderizar as cenas criadas. Primeiramente feito de maneira sequencial, mas com a adição de paralelismo na execução posteriormente, quando se percebeu que o cálculo da coloração da amostra de um pixel independe da cor de outra amostra, permitindo assim utilizar todos os núcleos físicos do processador para acelerar o processo.

3.1 AMBIENTE DE TESTES

O experimento foi realizado por meio de um computador contendo um processador AMD *Ryzen* 7 5700X de 8 núcleos físicos com *clock* variando de 3.4GHz a 4.6GHz, 16GB de memória RAM 3000MHz em *dual channel*, uma placa de vídeo NVIDIA *GeForce* GTX 1660 com *clock* de 1.8GHz e 6GB de VRAM, e um sistema operacional GNU Linux Ubuntu.

3.2 IMPLEMENTAÇÃO

Como descrito no Capítulo 2, o processo para renderizar uma cena qualquer consiste em posicionar uma câmera num ponto *P* e dele enviar raios pra cada um dos pixels, acrescentando um pequeno desvio entre cada coordenada do pixel para uma melhor aleatorização do raio gerado, o que também garante um efeito de *anti-aliasing*[1].

Utilizando essa ideia, o Algoritmo 4 é responsável por renderizar a tela, ele cria um raio da origem da câmera que vai na direção de um pixel qualquer e continua percorrendo a cena, repetindo esse procedimento para cada uma das amostras (Linhas 1 a 9). Após terminar essa amostragem de raios, nas linhas 10 e 11, gera-se uma média das colorações calculadas e esse pixel da tela nas coordenadas (x,y) é colorido pela função *WritePixel*, que salva num arquivo o seu RGB[2]. O programa ao fim da execução gera um arquivo de imagem no formato PPM, conhecido também como *Portable Pixmap*, sendo uma maneira simples e direta de representar a informação guardada em formato texto.

[1] Conhecido também como antisserrilhamento, é um método para redução do serrilhamento que ocorre ao desenhar formas geométricas, serve para misturar pixels de um objeto mais próximo de um mais distante, ou seja, numa fronteira de duas cores distintas, teremos alguns pixels de tonalidades intermediárias, melhorando a aparência da renderização.

[2] Red, Green e Blue; representa a quantidade de vermelho, verde e azul que combinadas formam uma cor.

Algoritmo 4 Render

Require: *image_height*, *image_width*, *max_depth*, *samples*, *World*
Ensure: imagem da renderização

for all $y \in image\_height$ **do**
  for all $x \in image\_width$ **do**
    $pixel\_color \leftarrow$ Color(0, 0, 0)
    for all $s \in samples$ **do**
      $u \leftarrow \left(\frac{x + random\_double()}{image\_width - 1}\right)$
      $v \leftarrow \left(\frac{y + random\_double()}{image\_height - 1}\right)$
      $ray \leftarrow$ **GetRay**(u, v) // gera um raio da origem da câmera até o grid do pixel
      $pixel\_color \leftarrow pixel\_color +$ **RayColor**(ray, max_depth, $World$)
    end for
    $pixel\_color \leftarrow \left(\frac{pixel\_color}{samples}\right)$
    WritePixel(x, y, $pixel\_color$) // escreve a cor obtida no pixel (x,y)
  end for
end for

A cor calculada pro pixel é realizada através da função *RayColor* que aplica uma técnica de *shading*, e sua implementação é descrita no Algoritmo 5. Esse método de coloração analisa se o raio passado como parâmetro intersecciona algum objeto da cena, verificando o tipo de material dessa superfície, e caso seja um material emissor retorna a cor dessa fonte luminosa. Porém, caso esse material seja difuso, metálico ou dielétrico, o raio será dispersado do ponto P onde ocorreu a interseção através dos métodos *LambertianScatter*, *MetallicScatter* e *DielectricScatter*, respectivamente. Todos esses materiais dispersam o raio, criando um novo saindo do ponto P e indo numa direção calculada pelas funções de *Scatter*.

Algoritmo 5 RayColor

Require: *ray*, *depth*, *World*
Ensure: *Color*
1: **if** *depth* = 0 **then**
2: **return** $Color(0,0,0)$ // atingiu a profundidade máxima de raios gerados
3: **end if**
4: **if** *ray* não teve interseção com nada em *World* **then**
5: **return** *background_color* // uma cor escolhida pro fundo da cena
6: **end if**
7: *HitRecord* ← informações da interseção
8: *color_from_emission* ← **Emitted**$(HitRecord)$ // retorna emissão luminosa da superfície interseccionada no ponto p
9: **if** *HitRecord.material* é emissor **then**
10: **return** *color_from_emission* // interseção com fonte luminosa, não dispersa o raio
11: **end if**
12: *attenuation* ← $Color(0,0,0)$
13: **if** *HitRecord.material* é difuso **then**
14: *attenuation* ← **LambertianScatter**(*ray*, *HitRecord*)
15: **else if** *HitRecord.material* é metal **then**
16: *attenuation* ← **MetallicScatter**(*ray*, *HitRecord*)
17: **else if** *HitRecord.material* é dielétrico **then**
18: *attenuation* ← **DielectricScatter**(*ray*, *HitRecord*)
19: **end if**
20: *scattered* ← raio dispersado pelo material
21: *color_from_scatter* ← *attenuation* $*$ **RayColor**(*scattered*, *depth* − *1*, *World*)
22: **return** *color_from_emission* + *color_from_scatter*

Na linha 14 do Algoritmo 5 será realizada a dispersão do raio para um material difuso de Lambert, calculada através do Algoritmo 6, consistindo em (1) obter a cor desse material e (2) encontrar uma nova direção qualquer ao redor do ponto de interseção, criando-se um novo raio dispersado pela cena.

Algoritmo 6 LambertianScatter

Require: *ray*, *HitRecord*
Ensure: *attenuation*, *scattered*
1: *scatter_direction* ← *rec.normal* + *random_unit_vector*()
2: **if** *scatter_direction* é muito próximo de zero em todas direções **then**
3: *scatter_direction* ← *rec.normal* // quando direção do raio disperso é degenerada
4: **end if**
5: *scattered* ← **Ray**(*HitRecord.p*, *scatter_direction*) // novo raio saindo do ponto de interseção
6: *attenuation* ← cor do material no ponto *HitRecord.p*
7: **return** *attenuation*

Para o dispersor do material metálico, na linha 16 do Algoritmo 5, a implementação é dada pelo Algoritmo 7, e a ideia é utilizar a função *Reflect* para calcular a reflexão de um espelho e descobrir a nova direção do raio, podendo existir um desvio adicional numa direção aleatória caso o metal tenha rugosidade, também chamado de *fuzz*. Assim, o raio dispersado e a cor do material são encontrados.

Algoritmo 7 MetallicScatter

Require: *ray*, *HitRecord*
Ensure: *attenuation*, *scattered*
1: *reflected* ← **Reflect**(*unit_vector*(*ray.direction*), *HitRecord.normal*) // raio refletido na superfície
2: *reflected* ← *reflected* + *fuzz* ∗ *random_unit_vector*() // valor escalar fuzz controla como a reflexão se aproxima de um espelho
3: *scattered* ← **Ray**(*HitRecord.p*, *reflected*) // novo raio saindo do ponto de interseção
4: *attenuation* ← cor do material no ponto *HitRecord.p*
5: **return** *attenuation*

Já para o caso da dispersão dielétrica na linha 18 do Algoritmo 5, diferente dos anteriores, quando houver uma interseção o raio não absorve a cor do material dielétrico, apenas reflete ou refrata o raio incidente. Através do Algoritmo 8, ao passar do ar para o objeto ou do objeto para o ar, é verificado se o ângulo θ do raio incidente é capaz de refratar seguindo a Lei de Snell, caso não haja uma solução ou a refletividade do material seja maior que um valor *epsilon*, o raio sofrerá reflexão; e caso contrário ele refrata.

Algoritmo 8 DielectricScatter

Require: *ray*, *HitRecord*
Ensure: *attenuation*, *scattered*
1: *attenuation* ← *Color*(*1*, *1*, *1*) // a superfície não absorve nada
2: *ri* ← *refraction_index*
3: **if** *HitRecord* com um raio entrando na superfície **then**
4: $ri \leftarrow \frac{1}{ri}$ // índice de refração do ar pro material
5: **end if**
6: *unit_direction* ← **unit_vector**(*ray.direction*())
7: $\cos\theta \leftarrow \mathbf{min}((-unit\_direction) \cdot HitRecord.normal, 1.0)$
8: $\sin\theta \leftarrow \sqrt{1.0 - \cos\theta^2}$
9: *reflectance* abaixo utilizando aproximação de Schlick
10: **if** $ri * \sin\theta > 1.0$ **or** $reflectance(\cos\theta, ri) > epsilon$ **then**
11: *direction* ← **reflect**(*unit_direction*, *HitRecord.normal*) // direção do raio ao refletir
12: **else**
13: *direction* ← **refract**(*unit_direction*, *HitRecord.normal*, *ri*) // direção do raio ao refratar
14: **end if**
15: *scattered* ← **Ray**(*HitRecord.p*, *direction*) // novo raio saindo do ponto de interseção

Depois que o primeiro raio disparado da câmera atinge um objeto com algum dos materiais implementados acima, ele absorve um pouco da cor *attenuation* desse material, se for possível, e um novo raio *scattered* é dispersado desse ponto de interseção pela cena, repetindo-se o processo até que a profundidade máxima seja alcançada.

3.3 RESULTADOS

Alguns resultados da implementação[3] como 3.1, 3.2 e 3.3 mostram renderizações possíveis do algoritmo de *Ray Tracing* implementado.

Apesar da técnica de iluminação utilizada apresentar resultados melhores para o sombreamento, ela gera um problema de ruído na imagem (também conhecido como *noisy*) que é

[3]Disponível em: *https://github.com/sekaikei/rtrender*

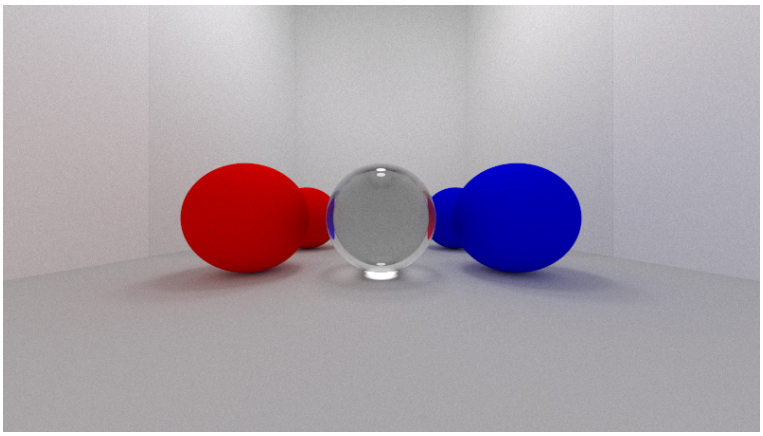


Figura 3.1: Esferas dentro de uma caixa com uma parede espelho atrás. Imagem renderizada utilizando 10.000 amostras e profundidade 10

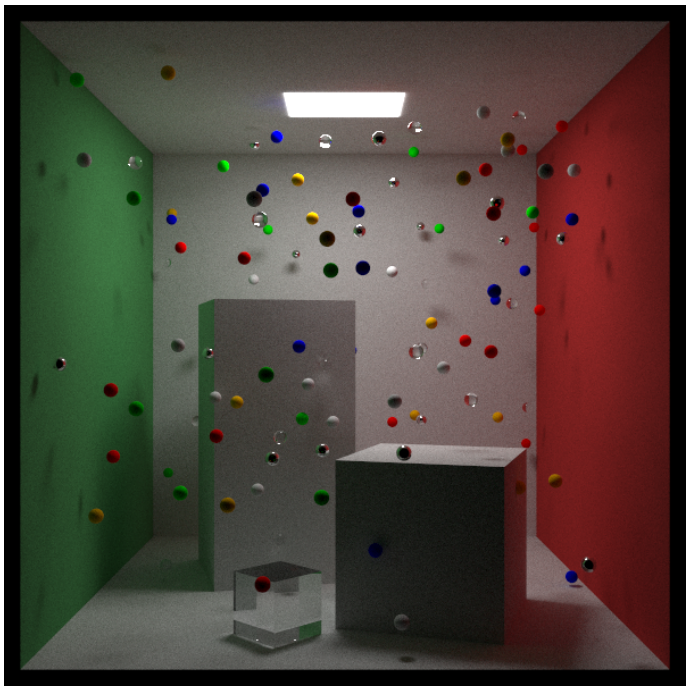


Figura 3.2: Renderização de uma Cornell Box com três cubos e cem esferas dentro. Imagem renderizada utilizando 10.000 amostras e profundidade 10

consequência de nem sempre ser possível um raio atingir a fonte de luz, sendo necessário várias amostras de raios por pixel para solucionar isso. É possível verificar esse comportamento na Figura 3.4 ao renderizar a cena da Figura 3.3 com apenas 1000 amostras.

Durante esse experimento, foi observado que o tempo de execução dessas renderizações é elevado, para gerar a Figura 3.1 levou aproximadamente 15 minutos; para gerar a Figura 3.2 levou aproximadamente 25 minutos; e para gerar a Figura 3.3 levou aproximadamente 25 minutos;.

3.4 CONCLUSÃO

Com o experimento realizado foi possível concluir que o algoritmo de *Ray Tracing* é um processo computacionalmente caro e o tempo para a renderização das cenas foi diretamente proporcional à quantidade de amostras de raio por pixel; o tanto de profundidade que um raio

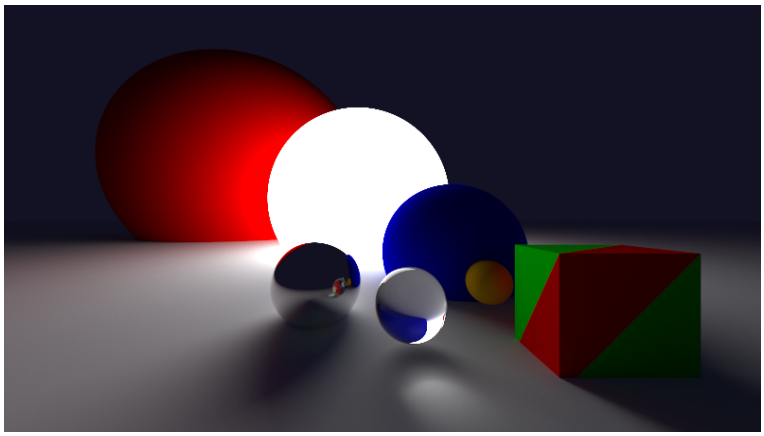


Figura 3.3: Esferas e cubo num plano com iluminação no centro. Imagem renderizada utilizando 100.000 amostras e profundidade 10

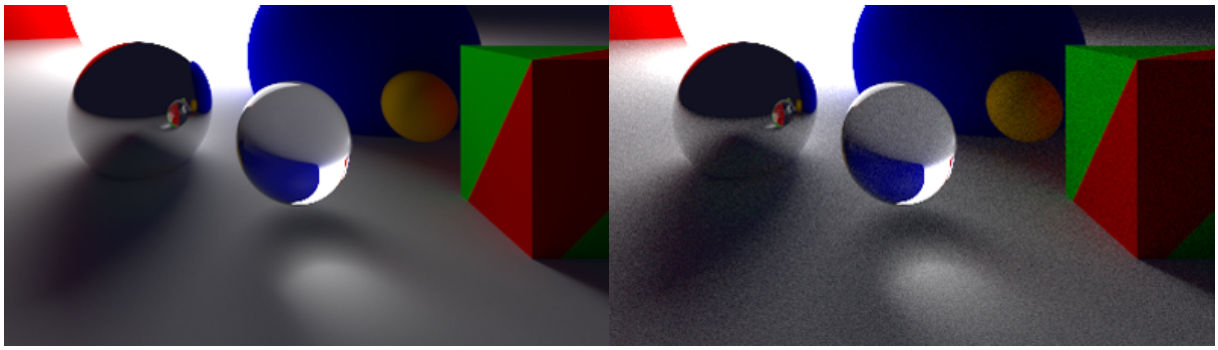


Figura 3.4: Ruído em esferas e cubo num plano com iluminação no centro. Ambas imagens renderizadas com profundidade 10, a imagem da esquerda foi renderizada utilizando 100.000 amostras, enquanto a imagem da direita foi renderizada com apenas 1.000 amostras

dispersa; e a complexidade da cena, isto é, quantas superfícies, qual forma geométrica delas e qual material possuem.

4 CONCLUSÃO

Neste trabalho iniciou-se informando brevemente a utilidade da área da computação gráfica no setor de entretenimento, alguns desafios enfrentados e técnicas responsáveis pela renderização, dando um destaque para a teoria envolvendo as técnicas de *Ray Casting* e *Ray Tracing* no Capítulo 2, expondo os elementos utilizados para se realizar uma renderização. Depois o Capítulo 3 apresentou passos necessários para criar um programa executável, capaz de renderizar cenas e gerar uma imagem na saída.

O estudo mostrou as principais técnicas que o *Ray Tracing* aplica para gerar imagens mais realistas, o que enriquece produções utilizando CGI, porém, esse algoritmo não é muito eficiente para aplicações em tempo real. Dessa forma, a computação deve ser feita de forma paralela para ganhar velocidade e é imprescindível a utilização de uma unidade gráfica exclusiva (GPU) para esse processamento. Existem muitas pesquisas relacionadas em avaliar essa melhoria, uma delas foi observada no experimento realizado no artigo (Castaño-Díez et al., 2008), em que se comparou a utilização de GPU e CPU no processamento de imagens, tendo como conclusão que uma placa de vídeo pode acelerar o tempo da aplicação em algumas vezes[1]. Com os tempos da renderização no experimento do Capítulo 3 foi possível perceber que não é viável utilizar somente a CPU, e portanto, seria interessante utilizar algumas bibliotecas de renderização gráfica como OpenGL e DirectX para auxiliar no processo de *Ray Tracing*, pois a utilização de uma GPU traz algumas melhorias como (1) maior capacidade de processamento paralelo, (2) possuir centenas de núcleos de processamento a mais que uma CPU, mesmo eles sendo menos potentes, (3) algoritmos com alto grau de localidade, devido à memória compartilhada, e (4) utilização do pipeline gráfico de renderização. Também, para placas gráficas mais modernas, existe ainda a tecnologia RT integrada em *hardware* acelerando os cálculos.

Muitas informações foram abstraídas para que a implementação de um renderizador *Ray Tracing* fosse possível, as mais evidentes sendo a incapacidade de ser uma aplicação em tempo real; falta de modelos 3D modernos sendo renderizados; e materiais muito simples, pois o comum é um objeto possuir várias características ao mesmo tempo, por exemplo, meio difuso e meio metálico. Portanto, para trabalhos futuros, uma aplicação RT fazendo uso da GPU poderia ser implementada, buscando se aproximar mais dos simuladores ou jogos em tempo real que atualmente utilizam essa técnica.

[1] Essa razão de melhoria é conhecida como *speedup* e representa quantas vezes um algoritmo paralelo é mais rápido quando comparado com sua versão sequencial.

REFERÊNCIAS

Blinn, J. F. e Newell, M. E. (1976). Texture and reflection in computer generated images. *Proceedings of the 3rd annual conference on Computer graphics and interactive techniques*.

Castaño-Díez, D., Moser, D., Schoenegger, A., Pruggnaller, S. e Frangakis, A. (2008). Performance evaluation of image processing algorithms on the gpu. *Journal of structural biology*, 164:153–60.

Goldstein, R. A. e Nagel, R. (1971). 3-d visual simulation. *SIMULATION*, 16(1):25–31.

Peter Shirley, Trevor David Black, S. H. (2024a). Ray tracing in one weekend. `https://raytracing.github.io/books/RayTracingInOneWeekend.html`. Acessado em 01/08/2024.

Peter Shirley, Trevor David Black, S. H. (2024b). Ray tracing: The next week. `https://raytracing.github.io/books/RayTracingTheNextWeek.html`. Acessado em 01/08/2024.

Phong, B. T. (1975). Illumination for computer generated pictures. *Seminal graphics: pioneering efforts that shaped the field*.

Prunier, J.-C. (2009). Scratchapixel 4.0 - computer graphics. `https://www.scratchapixel.com/index.html`. Acessado em 01/08/2024.

Roth, S. D. (1982). Ray casting for modeling solids. *Computer Graphics and Image Processing*, 18(2):109–144.

Shirley, P., Ashikhmin, M. e Marschner, S. (2009). *Fundamentals of computer graphics*. AK Peters/CRC Press.

Snyder, J. M. e Barr, A. H. (1987). Ray tracing complex models containing surface tessellations. *ACM SIGGRAPH Computer Graphics*, 21(4):119–128.