

Projeto 1: Algoritmo de Traçado de Raio

Aluno: William Ramirez Ruiz

Abril 2023

1 Lista

A seguir são descritos os pontos desenvolvidos para este trabalho, divididos por aplicação.

1.1 Aplicação básica

Renderizar uma cena 3D usando o algoritmo de traçado de raios:

- A cena deve ser criada com instanciação de esferas, caixas e/ou planos.
- A cena deve ser iluminada por uma ou mais fontes de luz pontual.
- A cena deve ser renderizada com iluminação direta, usando o modelo Phong e com cálculo de sombras.
- A cena deve poder ser renderizada com múltiplas amostras (distribuição uniforme) por píxel.

1.2 Aplicação Adicional

- Aplicação de transformações de modelagem na instanciação de formas geométricas.
- Instância de geometria representada por triângulos (sem estrutura de aceleração).
- Instanciação de objetos reflexivos.
- Instanciação de objetos refratários.

2 Implementação

2.1 Detalhes

Para este trabalho, decidi começar com o Python para criar uma possível implementação em C++, devido à minha pouca experiência como programador.

O objetivo disso era explorar em profundidade alguns aspectos relacionados à renderização usando o modelo de iluminação Phong, por exemplo, como calcular nossas coordenadas na câmera virtual, como calcular a contribuição da cor, levando em conta as componentes difusas, refletivas e especulares ao avaliar o modelo Phong. Python minimiza esses cálculos, então pode ser interessante explorar esse tópico.

Para este trabalho, comecei com o que foi visto na disciplina e no livro guia [1]. Além disso, segui o livro [2] para aprender vários aspectos do c++. O código resultante do trabalho está disponível no seguinte link:

<https://github.com/williamalbert94/Renderizacion-c-visual-studio-2023>.

2.2 Bibliotecas em c++

Para a tarefa de renderização, talvez seja necessário utilizar algumas bibliotecas externas, como GLFW. Em alguns casos, o uso dessas bibliotecas no Visual Studio envolverá apenas copiar e colar e importar os scripts a serem usados. No entanto, para bibliotecas como GLFW, entre outras, será necessário vincular os arquivos a serem importados nas configurações do C++, indicando o caminho do "include" do pacote em C++ -> Geral -> Diretórios de inclusão adicionais. Em seguida, devemos vincular a biblioteca como um todo através da pasta lib/x64, em C++ -> Vinculador -> Geral, nas pastas de bibliotecas adicionais. Finalmente, será necessário definir a entrada como uma macro.

2.3 Implementação: Psudocodigo

Seguindo [2, 1] decidi criar um código iterativo que começa criando um array tridimensional (largura, altura, canal) que corresponderia ao nosso filme. Depois disso, iríamos percorrer o array para dimensão 1 e dimensão 2.

Posteriormente, através de uma função "Render", iremos calcular a cor de saída de cada pixel em nosso array utilizando o modelo de iluminação de Phong. Nossa dinâmica se concentrará inicialmente em calcular as coordenadas de nossa câmera virtual no sistema de coordenadas global, para lançar um raio através dela. Em seguida, verificamos se esse raio interage com algum objeto e, posteriormente, avaliamos a contribuição do material usando o modelo de Phong, retornando um valor de 3 dimensões que representará a cor para nosso pixel no filme.

2.4 Gerando um raio

Para nossa implementação, devemos ter uma noção bastante forte de como configurar nossa câmera virtual em relação ao sistema de coordenadas global. Com isso, devemos definir qual será nosso ponto de origem no espaço, qual será nosso campo de visão (FOV) e calcular o vetor de saída para nosso raio.

A posição da câmera pode ser definida por um vetor de três componentes, que especifica as coordenadas x, y e z da posição da câmera no espaço. A direção de visão da câmera é geralmente definida por um vetor que aponta para onde a câmera está olhando. Esse vetor é normalizado, o que significa que suas componentes têm magnitude igual a 1. O vetor pode ser definido como um ponto no espaço (para a posição da câmera) e um segundo ponto no espaço para a direção da câmera.

```

function GENERATERAY( $nx, ny$ )
   $\Delta v \leftarrow f \tan \frac{\theta}{2}$ 
   $\Delta u \leftarrow \frac{\Delta vw}{h}$ 
   $p \leftarrow (-\Delta u + 2\Delta unx, -\Delta v + 2\Delta vny, -f, 1)$ 
   $o \leftarrow V^{-1}(0, 0, 0, 1)$ 
   $t \leftarrow V^{-1}p$ 
  return Ray( $o$ , normalize( $t - o$ ))
end function

```

Figure 1: Geração de raios em função de nx y ny

As equações utilizadas para calcular os vetores u , v e n são as seguintes, seguindo o pseudocódigo apresentado na Figura 1:

$$fov_{rad} = f * \frac{\arccos(-1)}{128} \quad (1)$$

$$n = no * \left(\frac{1}{(2 * \tan(\frac{fov_{rad}}{2.0}))} \right) \quad (2)$$

$$uo = vo * no \quad (3)$$

$$u = \frac{uo}{||uo||} \quad (4)$$

As equações utilizadas para gerar o raio são:

$$p = u * normx + v * normy + camera + n \quad (5)$$

$$raio = rayo(po, po - camera) \quad (6)$$

Onde vo e no são vetores passados como parâmetros da construção da cena, fov_{rad} é o ângulo de visão da câmera em radianos e $||uo||$ é a norma do vetor uo .

O campo de visão é o ângulo de abertura da câmera, geralmente é medido em graus ou radianos e define a extensão da cena que será capturada pela câmera. É possível definir o campo de visão através de uma fórmula trigonométrica com base no tamanho da imagem de saída e na distância focal da câmera (Ver Figura 2 e Eq. 2, divisor).

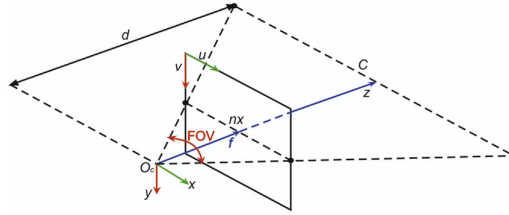


Figure 2: Pinhole camera model, tirado de [3].

Depois de definir a posição, direção de visão e campo de visão da câmera, podemos usar essas informações para calcular o raio que será lançado. Isso é feito usando uma equação matemática que leva em conta a posição da câmera, a direção de visão e o

campo de visão (ver Eq. 5 e Eq.6). Com o raio calculado, já é possível testar se o raio intersecta algum objeto na cena.

2.5 Traçando um raio

Depois de gerar o raio, devemos verificar se ele intersecta alguns objetos, a fim de calcular a cor de saída para o pixel específico, usando o modelo de iluminação Phong. Esse processo pode ser definido pelo pseudocódigo na Figura 3.

```
function TRACERAY(ray)
    hit  $\leftarrow$  ComputeIntersection(ray)
    if hit then
        if hit.light then
             $r \leftarrow hit.t$ 
             $c \leftarrow hit.light.P/r^2$ 
        else
             $c \leftarrow hit.material.Eval(this, hit, ray.o)$ 
        return c
    end if
end if
end function
```

Figure 3: Algoritmo de traçado de raios

Na primeira linha, a função `ComputeIntersection(ray)` é chamada para determinar se o raio *ray* intersecta algum objeto na cena. Se a interseção for encontrada, entra-se no loop *if hit then* que verifica se o objeto intersectado é uma fonte de luz (*hit.light*). Se for o caso, então o valor da cor *c* é calculado como a intensidade da luz da fonte dividida pelo quadrado da distância entre a fonte de luz e o ponto de interseção. Este cálculo é feito usando a fórmula $c = hit.light.P/r^2$. Caso contrário, se o objeto intersectado não for uma fonte de luz, a função `Eval` do material do objeto intersectado é chamada para determinar a cor do objeto nesse ponto. Este cálculo é feito usando a fórmula $c = hit.material.Eval(this, hit, ray.o)$. Finalmente, a cor calculada *c* é retornada.

O modelo de Phong considera três componentes de luz em um ponto: a componente ambiente, a componente difusa e a componente especular (Figura 4). A componente ambiente é a luz geral que ilumina todos os objetos na cena. A componente difusa é a luz que é espalhada uniformemente pela superfície em todas as direções. A componente especular é a luz que é refletida pela superfície em uma direção específica, que depende da posição da fonte de luz e da direção do raio de visualização.

O pseudocódigo (Figura 4) começa inicializando a cor final *ret* como preto e normalizando o vetor normal. Em seguida, calcula a componente ambiente usando a cor ambiente do material e a potência da luz. As variáveis *diffuse* e *specularHighlight* são inicializadas como preto, para serem somadas mais tarde.

Em seguida, há um loop que percorre todas as fontes de luz presentes na cena. Para cada fonte de luz, é calculada a direção do raio de luz que aponta para o ponto *p*. É então verificado se há algum objeto na cena que obstrui a luz, ou seja, se há sombras. Se não houver sombras, o cálculo da componente difusa e especular é feito. Caso contrário, apenas a componente ambiente é considerada.

A componente difusa é calculada multiplicando a cor difusa do material, a potência da luz, o coeficiente de difusão do material e o produto escalar entre o vetor normal e

```

function EVALPHONG(normal, p, r, m, lights)
    ret  $\leftarrow$  Color(0.0, 0.0, 0.0, 1.0)
    normal  $\leftarrow$  normalize(normal)
    ambientColor  $\leftarrow$  m.getAmbient(p)  $\cdot$  LightPower
    diffuse  $\leftarrow$  Color(0.0, 0.0, 0.0, 1.0)
    specularHighlight  $\leftarrow$  Color(0.0, 0.0, 0.0, 1.0)
    for each light in lights do
        lightDirection  $\leftarrow$  normalize(light - p)
        lightRay  $\leftarrow$  Ray(p + lightDirection * EPS, lightDirection)
        hitObj  $\leftarrow$  intersect(lightRay)
        reflectiveness  $\leftarrow$  1000.0  $\cdot$  m.kr
        if hitObj = NULL or (normal  $\cdot$  lightDirection < 0) then
            diffuseColor  $\leftarrow$  m.getDiffuse(p)
            specularColor  $\leftarrow$  m.getSpecular(p)
            lDotNormal  $\leftarrow$  max(normal  $\cdot$  lightDirection, 0)
            diffuse  $\leftarrow$  diffuse + (diffuseColor  $\cdot$  LightPower  $\cdot$  m.kd  $\cdot$  lDotNormal)
            disFromLight  $\leftarrow$  normalize(r.v * -1 + lightDirection)
            specReflec  $\leftarrow$  max(disFromLight  $\cdot$  normal, 0)
            reflectivity  $\leftarrow$  pow(specReflec, reflectiveness)
            specularHighlight  $\leftarrow$  specularHighlight + specularColor  $\cdot$  LightPower  $\cdot$ 
            reflectivity
        end if
    end for
    finalColor  $\leftarrow$  ambientColor + diffuse + specularHighlight
    return finalColor
end function

```

Figure 4: Avaliação do modelo de iluminação de Phong.

a direção da luz. A componente especular é calculada considerando a cor especular do material, a potência da luz, o coeficiente de reflexão do material e o ângulo de reflexão. Finalmente, a cor final é calculada somando a componente ambiente, a componente difusa e a componente especular.

Como último passo, aplicamos nossa renderização (veja a Figura 5), que irá iterar sobre cada pixel nas 2 dimensões e nos permitirá definir o número de amostras para efeitos de supersampling (Antialias).

3 Experiencias

Esta seção replicou o que foi visto na disciplina, alcançando vários dos resultados estudados nas horas de estudo.

Inicialmente ajustamos as coordenadas da câmera para definir nosso zero (0,0,0), e nos certificamos de que nossa cena esteja dentro do alcance de visão de nossa câmera, para que ao gerar o feixe, uma delas consiga "acertar" com o objeto. A função Render neste caso terá apenas 1 amostra, o modelo phong não irá considerar a luz ambiente, e a radiância não está sendo calculada através da condicional que nos diz se existe algum objeto no caminho do ponto avaliado até a luz fontes.

Na Figura 6, podemos ver qualitativamente que nosso código não está calculando se existe um objeto no caminho do nosso raio além da esfera, omitindo o cálculo da

```

function RENDER(film, camera, scene)
   $n \leftarrow \text{film.SampleCount}()$ 
  for each pixel (i, j) in film do
     $c \leftarrow 0$ 
    for  $k \leftarrow 1$  to  $n$  do
       $(x_n, y_n) \leftarrow \text{film.GetSample}(i, j)$ 
       $\text{ray} \leftarrow \text{camera.GenerateRay}(x_n, y_n)$ 
       $c \leftarrow c + \text{TraceRay}(\text{ray}, \text{scene})$ 
    end for
    film.SetValue(i, j, c/n)
  end for
end function

```

Figure 5: Algoritmo para renderizar uma cena usando um modelo de câmera e filme.

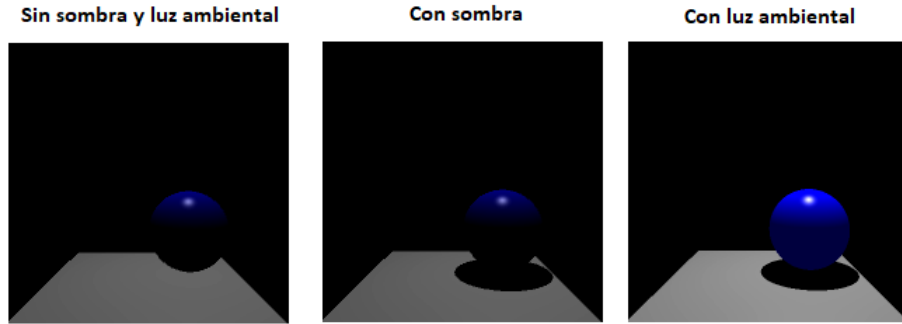


Figure 6: Mudança no componente de radiância em nossa cena.

backface. Posteriormente, calculamos se o raio colide com algum outro objeto na cena e calculamos a radiância, se não houver colisão, retorna $(0,0,0)$. Isso permitirá que calculemos a contribuição de radiância neste caso no plano.

Outro aspecto relevante é como calculamos a componente de saída de PhongEval. Nós podemos indicar que existe uma luz ambiente e a consideramos na contribuição de radiância no modelo, assim como mostrado na Figura 6. Para este caso, foi considerado um fator de atenuação para a luz ambiente e uma intensidade para o raio em proporção de 1.

Depois de adicionarmos o cálculo de sombras e a componente de luz ambiente, devemos definir alguns aspectos relacionados ao nosso material. Inicialmente, devemos definir se ele é difuso, refletivo ou refrativo, definindo sua cor e os atributos kr , kd e kt , respectivamente.

- kr (coeficiente de reflexão) é um número que representa a quantidade de luz que um material reflete. Um valor alto de kr significa que o material reflete muita luz e parece brilhante e especular, como um espelho.
- kd (coeficiente de difusão) é um número que representa a quantidade de luz que um material absorve e dispersa em todas as direções. Um valor alto de kd

significa que o material é muito absorvente e não reflete muita luz diretamente. Em vez disso, a luz se dispersa e se difunde em todas as direções, dando uma aparência opaca e fosca ao material.

- kt (coeficiente de transmissão) é um número que representa a quantidade de luz que um material transmite através de sua superfície. Um valor alto de kt significa que o material é muito transparente e permite que a luz o atravesse. Um valor baixo de kt significa que o material é muito opaco e não permite que a luz passe através dele. Em geral, materiais transparentes têm um valor alto de kt , enquanto materiais opacos têm um valor baixo ou nulo de kt .

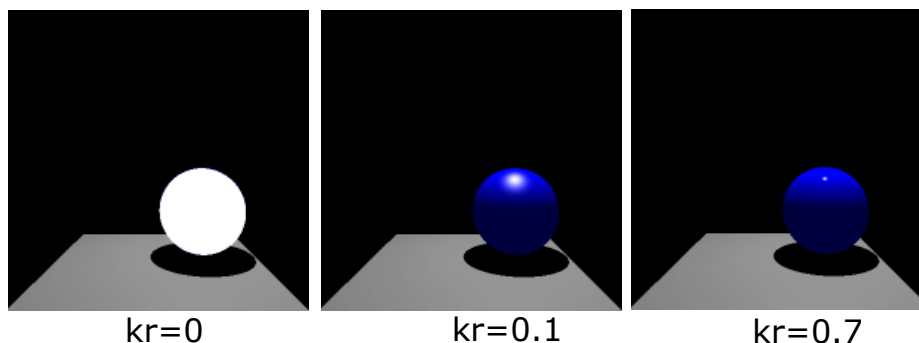


Figure 7: Mudança no componente total ao considerar vários valores de reflexão.

A Figura 7 mostra como a componente total da cor vai mudando à medida que o valor de kr varia de baixo para alto. Para este caso, o material foi considerado difuso, de cor azul com um valor de $kr=0.1$, com uma contribuição na componente especular igual a 1.

Até este ponto, nosso algoritmo realiza uma renderização básica sem cálculo de reflexão e refração, portanto, alguns outros aspectos podem ser melhorados, como o ruído associado, usando anti-aliasing. Como se mostra na Figura 5, renderizar nossa cena com várias amostras, considerando uma distribuição uniforme, pode nos levar a evitar vários efeitos de dente de serra em nossa renderização. Ao renderizar com múltiplas amostras uniformes, múltiplos raios são gerados para cada pixel, com pequenas variações em sua direção. Isso permite que a imagem seja amostrada em diferentes posições e reduz o efeito de aliasing. Além disso, ao fazer a média dos resultados dos raios, uma imagem mais suave e detalhada é obtida, reduzindo ainda mais os efeitos de aliasing.

Como mostra a Figura 8 e a Figura 9, a renderização com várias amostras diminui significativamente o aliasing presente na renderização.

Com o passo anterior, já teríamos uma renderização aceitável, porém pouco realista, sendo necessário adicionar ao nosso EvalPhong o cálculo da componente reflexiva, criando materiais reflexivos e adicionando objetos novos à cena através da instanciação por triângulos. A instanciação por triângulos nos permite gerar objetos mais complexos, tendo total controle sobre os aspectos da forma.

Para avaliar a refletividade e refração, através dos atributos do material, retornamos que tipo de material este possui e seus valores de kd , kr e kt . Com isso, através de uma flag, calculamos as componentes relacionadas à reflexão e refração. Para a

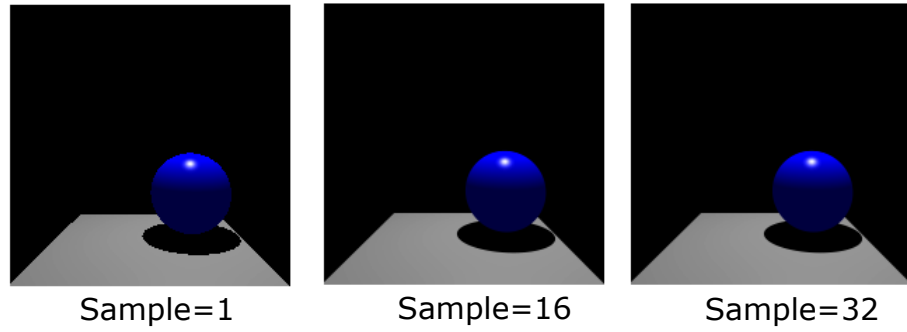


Figure 8: Renderização com várias amostras.

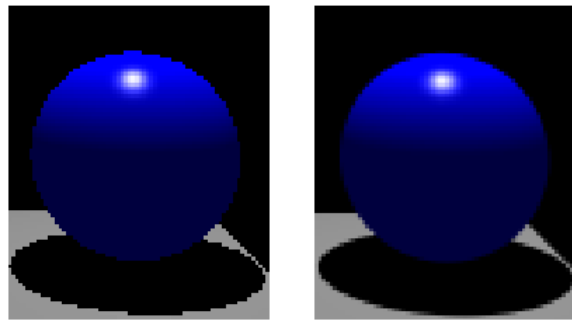


Figure 9: Zoom do objeto renderizado com varias amostras, $n=2$ vs $n=16$.

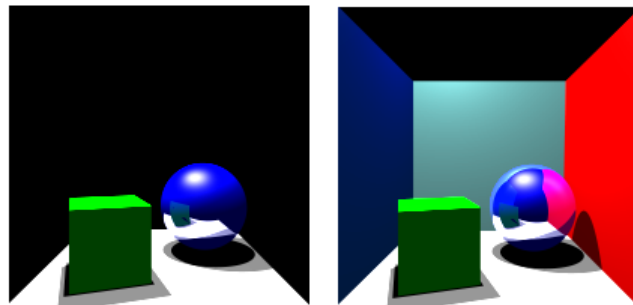


Figure 10: Reflexão de um cubo em uma esfera, definindo aleatoriamente um material.

reflexão, é lançado um raio na direção da reflexão e é calculada a cor do objeto que este raio atinge. Este processo é repetido várias vezes, criando uma série de reflexões em cascata, até que o raio não atinja nenhum objeto na cena ou que se alcance um número máximo de reflexões (recursivamente).

Para a refração, é lançado um raio na direção da refração, calculando a lei de Snell para determinar a direção do raio refratado. É calculada a cor do objeto que este raio atinge e o processo é repetido de maneira similar à reflexão. Para fins práticos foi definido que o meio era o ar.

A quantidade de reflexão e refração no objeto é determinada pelos atributos de material kr e kt , respectivamente. Estes valores indicam quanto de luz é refletida e refratada em comparação com a quantidade de luz que é absorvida ou difundida no objeto.

Como um primeiro teste, foi criado um material com cor azul (0,0,1), uma contribuição especular de (0.2,0.2,0.2,1.0), $kr = 0.6$, $kd = 0.1$ e $kt = 0.1$. Isso gerou uma representação bastante irreal de uma esfera metálica polida, onde podemos ver como outras primitivas (cubo sem componente reflexiva alguma) são refletidas sobre a esfera azul. Ao adicionar mais primitivas, é mais evidente como este material é bastante irreal em relação ao que conhecemos na realidade (ver Figura 10).

Consequentemente, considerou-se explorar outros tipos de materiais, por exemplo, um mais próximo a um espelho. Este foi definido com $kr = 1.0$, $kd = 0.0$ e $kt = 0.0$, sem definir uma cor própria, criando um espelho ideal. Isso nos levou ao que se vê na Figura 11.

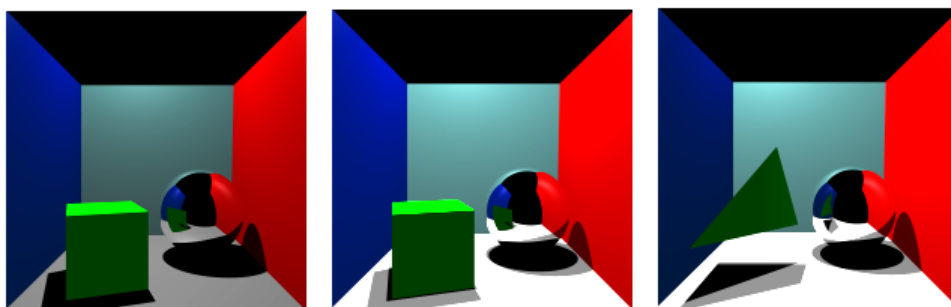


Figure 11: raio-objeto.

Na Figura 11, vemos nossa cena final, onde partimos de uma cena com apenas uma luz pontual, depois passamos para uma cena com 2 luzes pontuais e finalmente, por instanciação de objetos, criamos uma figura trapezoidal, que está suspensa no ar. Como podemos ver, o novo material é um espelho perfeito e nos permite observar com maior clareza o efeito de reflexão em nossa cena final.

Entre alguns efeitos relevantes, podemos ver que a esfera na parte superior tem uma curvatura com componente de luz. A luz que passa pela área com índice de refração mais alto curva-se mais e sai da esfera em uma direção diferente, o que faz com que essa área pareça mais brilhante do que o resto da esfera. Em relação à reflexão, podemos ver como a esfera tem um efeito de olho de peixe, refletindo de forma distorcida várias das primitivas na cena.

Finalmente, através do resultado obtido em nossa cena final, podemos concluir

que o modelo de iluminação Phong é altamente relevante na aplicação de iluminação direta, no entanto, o custo computacional foi bastante elevado, levando a resultados intermediários. Durante a execução, foi observado que a criação de primitivas pode ser facilitada através da instância de triângulos, permitindo-nos criar representações complexas diferentes em nossa cena. Outro aspecto importante foi como calculamos a reflexão e refração levando em conta a nossa backface, durante a implementação, este foi o desafio mais significativo, uma vez que o cálculo incorreto resultava em ultrapassar o intervalo de valores de cor, levando a uma primitiva totalmente negra.

Dentro de algumas limitações, também foi observado como o modelo produz sombras duras e abruptas, sem considerar a luz que é indiretamente refletida em uma cena. Além disso, o modelo de Phong não considera os efeitos de dispersão de luz em meios como ar, água ou fumaça. Esses meios podem distorcer e espalhar a luz de maneiras complexas, o que pode afetar significativamente a aparência de uma cena. Com outras texturas, o modelo Phong estava limitado, pois ele tinha alguns desafios ao lidar com texturas complexas ou materiais translúcidos, como vidro.

4 Considerações de implementação

A classe *escena* tem um construtor que recebe como parâmetros um ponto de vista, uma direção de visão, um campo de visão e um tamanho de imagem. Também há várias funções para adicionar objetos e luzes à cena, obter o próximo objeto ou luz da cena e gerar um raio para um pixel específico na imagem.

No construtor, o campo de visão é calculado em radianos e o vetor de direção de visão é normalizado dividindo-o por duas vezes a tangente do campo de visão dividido por dois. Em seguida, o vetor horizontal u da câmera é calculado usando o produto cruz entre o vetor de direção de visão e o vetor vertical v . O vetor u é normalizado e definido como um membro da classe.

Na função *Generarrayo*, as coordenadas do pixel são normalizadas e, em seguida, o ponto p no espaço 3D correspondente ao pixel na imagem 2D é calculado usando a posição da câmera, os vetores de direção e horizontal e as coordenadas normalizadas do pixel. Depois, um raio é criado e retornado, originando-se no ponto p e apontando na direção da posição da câmera para p .

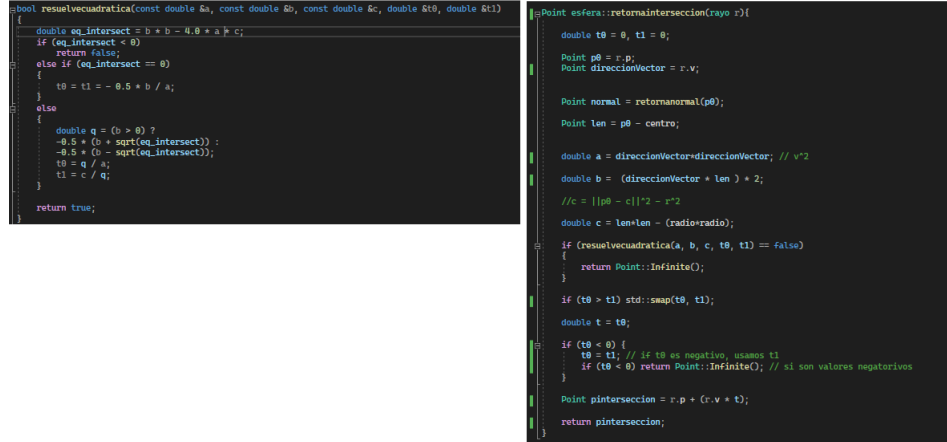
Tendo isso em mente, para nossa implementação em C++, é necessário definir uma primitiva na cena e um plano, criando um material e um atributo de cor. Além disso, devemos ter clareza sobre como avaliaremos a contribuição desse material usando o modelo de iluminação de Phong (ver Figura 4).

Como podemos ver, c é composto por 3 componentes (*ambient* + *diffuse* + *specular*), que nos retornarão diferentes resultados que veremos mais adiante.

Para implementar nossa renderização em C++, é necessário considerar as funções *EvalPhong* e *ComputeIntersection*. A última é uma função da classe *Objeto* que, a partir das coordenadas definidas para luz, câmera e primitiva, retorna se o raio que sai da câmera intersecta a primitiva. Começamos definindo o objeto e seus atributos, como o material e as componentes kr , kt e kd . A ideia é que, uma vez definido o objeto, computemos se o raio intersecta o objeto. A equação para isso é de segundo grau, $at^2 + bt + c = 0$, onde t é definido pela Eq. 7.

$$t = -b \frac{\sqrt{b^2 - 4ac}}{(2a)} \quad (7)$$

Na qual $a = \hat{d}\hat{d}$, $b = 2\hat{d}(o - c)$ e $c = (o - c)(o - c) - r^2$. Em nossa implementação, devemos evitar a auto-interseção e saber qual é o nosso backface (ver Figura 12).



```

bool resolvecuadratica(const double &a, const double &b, const double &c, double &t0, double &t1)
{
    double eq_intersect = b * b - 4.0 * a * c;
    if (eq_intersect < 0)
        return false;
    else if (eq_intersect == 0)
    {
        t0 = t1 = -0.5 * b / a;
    }
    else
    {
        double q = (b > 0) ?
            -0.5 * (b + sqrt(eq_intersect)) :
            -0.5 * (b - sqrt(eq_intersect));
        t0 = q / a;
        t1 = c / q;
    }
    return true;
}

// Point esfera::retornainterseccion(rayo r){
    double t0 = 0, t1 = 0;
    Point p0 = r.p;
    Point direccionVector = r.v;

    Point normal = retornanormal(p0);
    Point len = p0 - centro;

    double a = direccionVector * direccionVector; // v^2
    double b = (direccionVector * len) * 2;
    // c = ||p0 - c||^2 - r^2
    double c = len * len - (radio * radio);

    if (resolvecuadratica(a, b, c, t0, t1) == false)
    {
        return Point::Infinito();
    }

    if (t0 > t1) std::swap(t0, t1);

    double t = t0;

    if (t0 < 0) {
        t0 = t1; // if t0 is negative, swap t1
        if (t0 < 0) return Point::Infinito(); // si son valores negativos
    }

    Point pinterseccion = r.p + (r.v * t);
    return pinterseccion;
}

```

Figure 12: Interseção raio-objeto.

Na função *esfera::retornainterseccion(rayo)*, é verificado se o discriminante da equação quadrática é negativo, indicando que não há interseção real, e se t_0 e t_1 são negativos, indicando que os pontos de interseção estão atrás da direção do raio. Em ambos os casos, é retornado um ponto "infinito" para indicar que não há interseção. Esse processo garante que o raio não intersecte a esfera dentro de seu próprio volume, que é uma forma de evitar a auto-interseção e saber se temos um backface.

Uma vez definida a classe objeto juntamente com material para calcular se nosso raio intersecta o objeto, podemos criar a função mostrada na Figura 3 (chamada de traçar raio). Como esta função mostra, começamos computando se o raio intersecta e se essa interseção é com o objeto ou não. Se não for, o componente hit.light em nosso caso seria um array de [0,0,0], já que ao não intersectar o objeto, a contribuição é 0 (ver Figura 13). Com isso, simplesmente retornamos o mesmo componente. Se intersectar o objeto, Hit.light será um array maior que 0 e podemos calcular a contribuição em nosso array c através de "EvalPhong".

Apenas precisaríamos adicionar à nossa cena, através do script, objetos em forma de esferas ou triângulos para criar nossa cena com uma ou várias primitivas (objetos criados a partir de esferas e triângulos).

Considerando as classes anteriores, como Render e PhongEval, podemos ver como o processo de renderização vai mudando ao considerar vários fatores, como gerar várias amostras uniformes por pixel, colocar uma condição para ver a incidência da luz sobre o objeto e uma superfície (condição de sombra), a definição do material e o cálculo da reflexão e refração.

O código de renderização foi definido com 3 Loop 2 para cada dimensão e o último para supersampling (ver Figura 14). Quanto PhongEval definido apenas como indicado no pseudocódigo acima.

```

Color raytracer::trazarayo(ray r, int depth){
    Color c=Color(0.0,0.0,0.0,0.0);

    objeto* hit = intersecta(r);
    if (hit != NULL) {
        Point hit_light = hit->retornainterseccion(r);
        if (hit_light.x==0 && hit_light.y==0 && hit_light.z==0) {
            c = c;
        }
        else {
            Material* mat = hit->retornamaterial();
            c = EvalPhong(hit->retornanormal(hit_light), hit_light, r, mat, hit);
            hit->retornamaterial();
        }
    }
    return c;
}

escena::escena(Point * vo, Point * no, double f, int No){
    delta_v = vo;
    n = no;

    //calculo el fov en rad
    fov_rad = f * acos(-1) / 180.0;

    *n = (*n) * (1 / (2 * tan(fov_rad / 2.0)));
    Point uo = delta_v->cross(*n);
    uo.normalize();
    delta_u = new Point(uo.x, uo.y, uo.z);
    C = Point(0, 0, 0);
    N = No;

    index0 = 0;
    index1 = 0;
}

rayo escena::generarayo(double xo, double yo){
    double normx = (xo/N)-0.5;
    double normy = (yo/N)-0.5;
    Point p = (*delta_u)*normx + (*delta_v)*normy + (*camara) + (*n);
    rayo r = rayo(p, (p -(*camara)));
    return r;
}

```

Figure 13: Funciones relevantes para render (generar rayo/trazar rayo).

```

for(int i = 0; i < width; i++){
    for(int j = 0; j < height; j++) {
        Color rad = raytracer->renderiza(i, j);
        std::cout << rad.r << rad.g << rad.b << std::endl;

        pixels[i][j][0] = rad.r;
        pixels[i][j][1] = rad.g;
        pixels[i][j][2] = rad.b;
    }
}

Color raytracer::renderiza(int x, int y){
    Color c = Color(0.0,0.0,0.0,0.0);
    for (int i = 1; i <= muestras; i++) {
        for (int j = 1; j <= muestras; j++) {
            double one, two;
            //randomes as 0.5
            one = double(i) - 0.5 + double(1) / double(muestras);
            two = double(j) - 0.5 + double(1) / double(muestras);
            rayo r = Escena->generarayo(one, two);
            c = c + trazarayo(r, 0);
        }
    }
    c = c/(double(muestras * muestras));
    return c;
}

```

Figure 14: Função de renderização.

References

- [1] J. D. Foley, F. D. Van, A. Van Dam, S. K. Feiner, and J. F. Hughes, *Computer graphics: principles and practice*, vol. 12110. Addison-Wesley Professional, 1996.
- [2] P. Shirley, “Ray tracing in one weekend,” *Amazon Digital Services LLC*, vol. 1, 2016.
- [3] S. Rende, A. D. Irving, T. Bacci, L. Parlagreco, F. Bruno, F. De Filippo, M. Montefalcone, M. Penna, B. Trabucco, R. Di Mento, *et al.*, “Advances in microcartography: A two-dimensional photo mosaicing technique for seagrass monitoring,” *Estuarine, Coastal and Shelf Science*, vol. 167, pp. 475–486, 2015.