

# Computação Gráfica

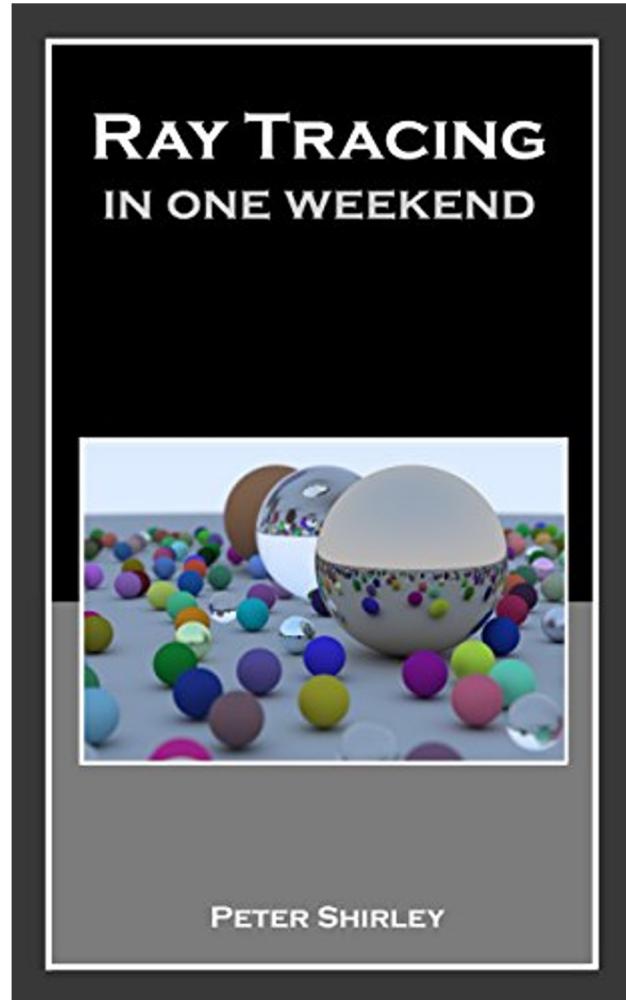
Aula 25: Raytracing

## Ray Tracing

Esta parte do curso foi baseada no livro:

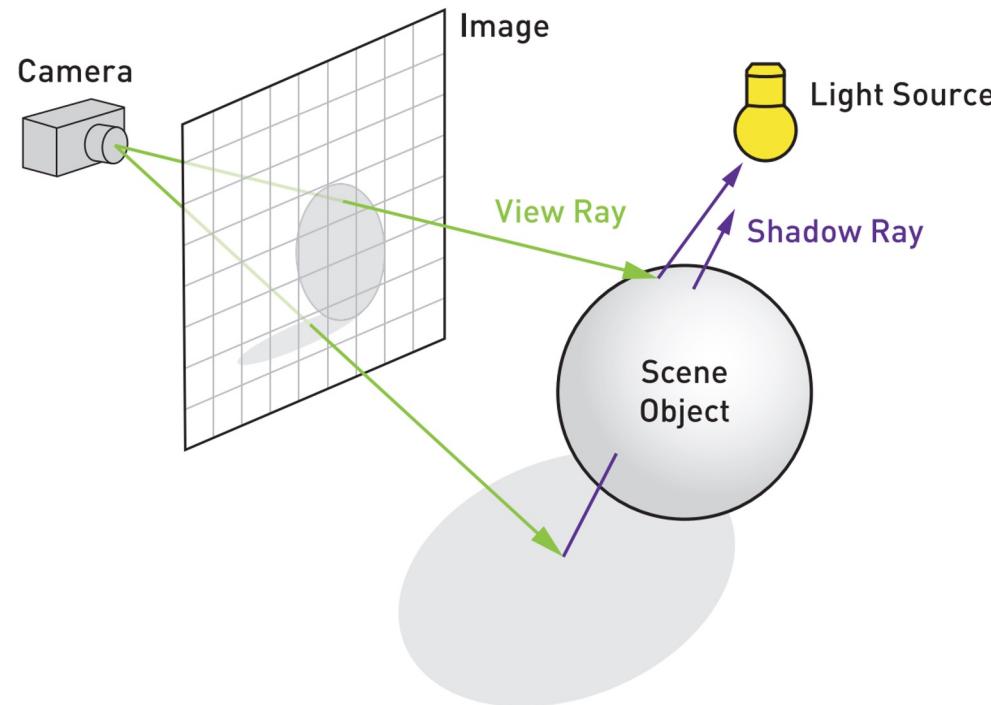
“Ray Tracing in One Weekend”, por Peter Shirley

URL (series): <https://raytracing.github.io/>

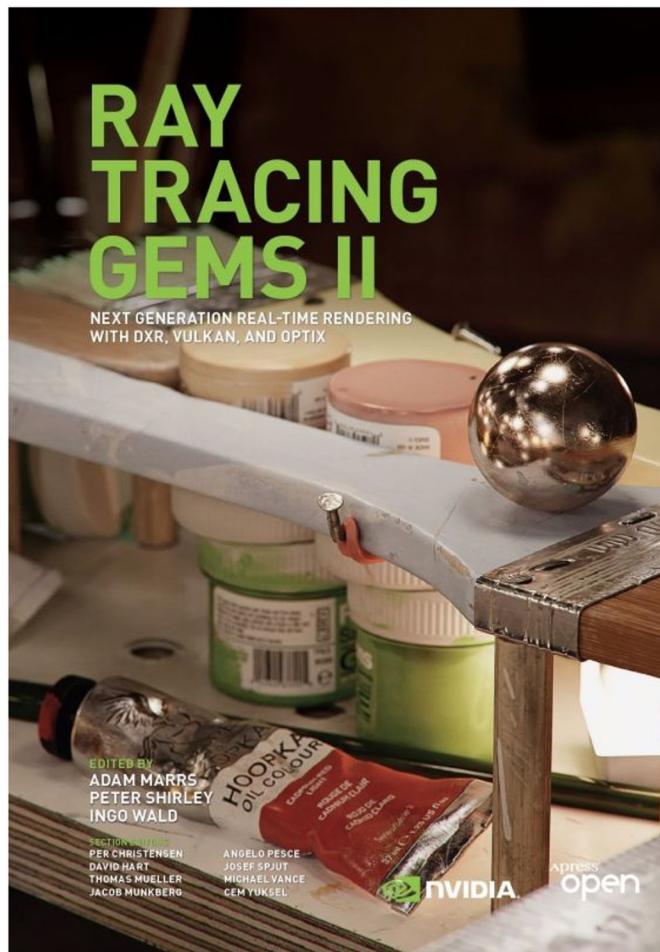
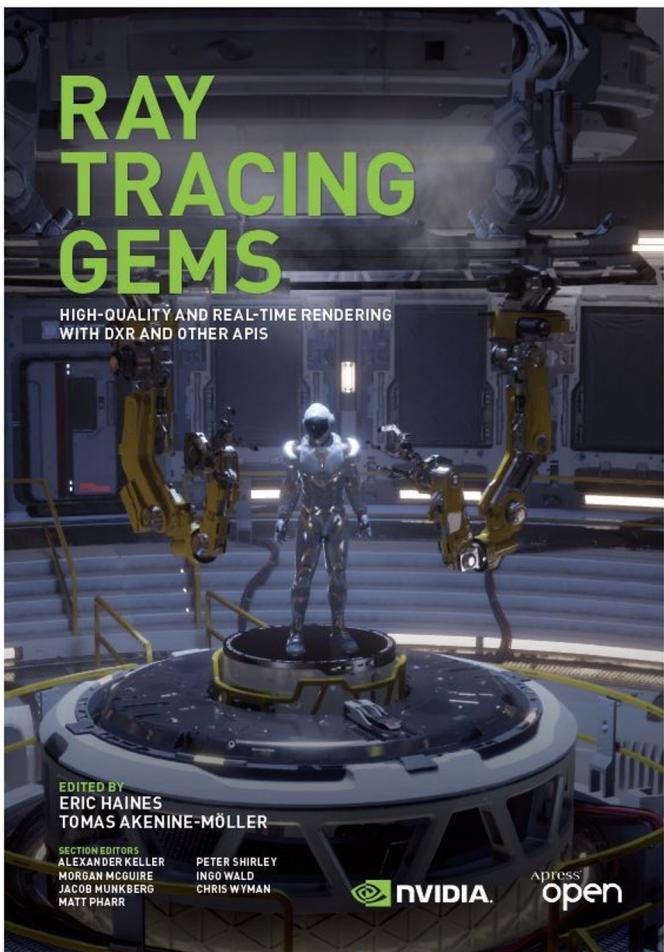


# Ray Tracing

O Ray Tracing usa um mecanismo de projeção de raios para reunir recursivamente as contribuições de luz de objetos reflexivos e refrativos.



# Alguns Livros Interessantes



# Onde Ray Tracing é usado

Ray Tracing é computacionalmente muito pesado, contudo com os avanços em recursos computacionais está se tornando viável até para aplicações em tempo real.

**Toy Story 4 (offline)**



**Resident Evil Village (realtime)**



"some frames in *Monsters University* took a reported 29 hours each"

fonte: Future of Gaming : Rasterization vs Ray Tracing vs Path Tracing

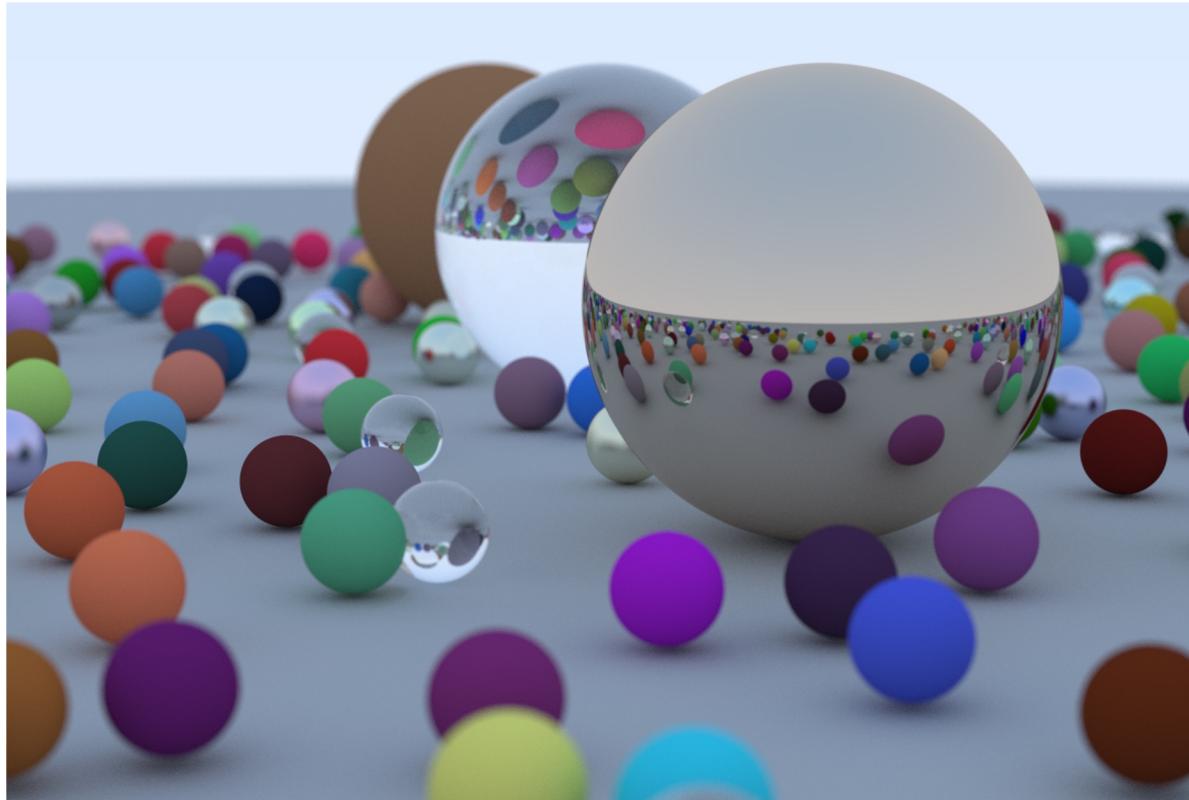
# Path Tracer

Um Path Tracer é um método de renderização baseada em Monte Carlo para produzir imagens de cenas tridimensionais. Na prática, o livro usado trata de um Path Tracer.



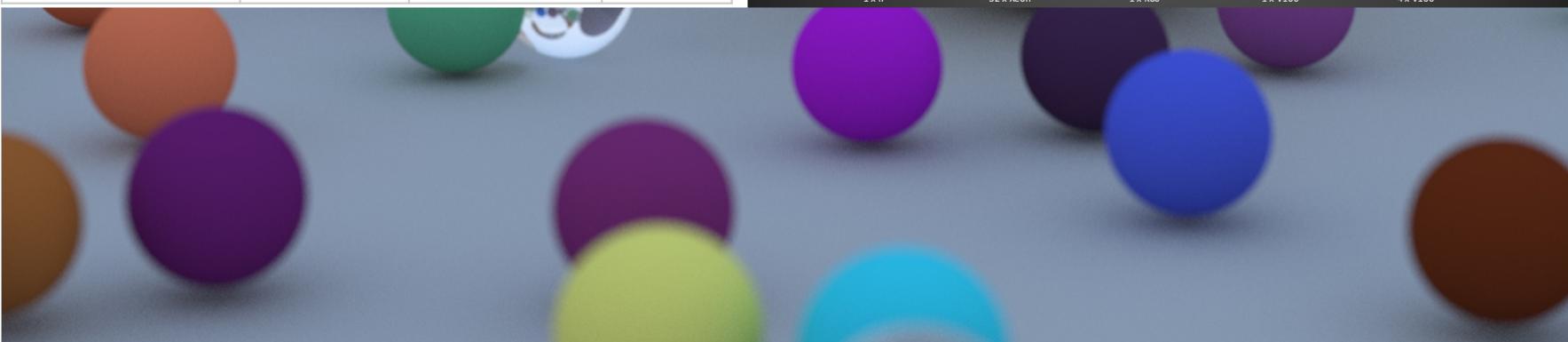
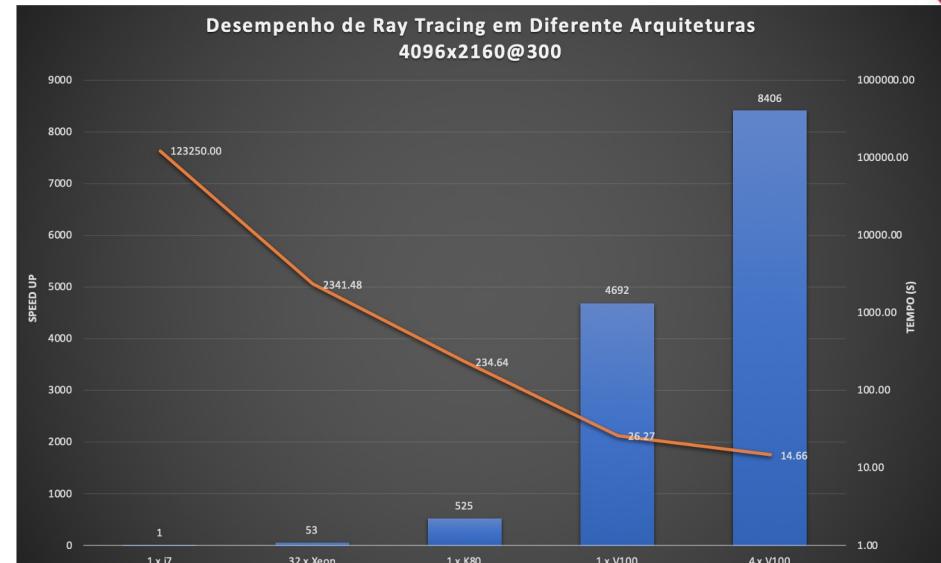
# Projeto

Vocês irão desenvolver um Path Tracer baseado na proposta do livro texto em GLSL.



# Desempenho do Algoritmo

Arquitetura	Speed up	Tempo (s)	Horas
1 x i7	1	123250.00	34.236
32 x Xeon	53	2341.48	0.650
1 x K80	525	234.64	0.065
1 x V100	4692	26.27	0.007
4 x V100	8406	14.66	0.004



# Raio (half-line / semirreta)

Raios do RayTracer

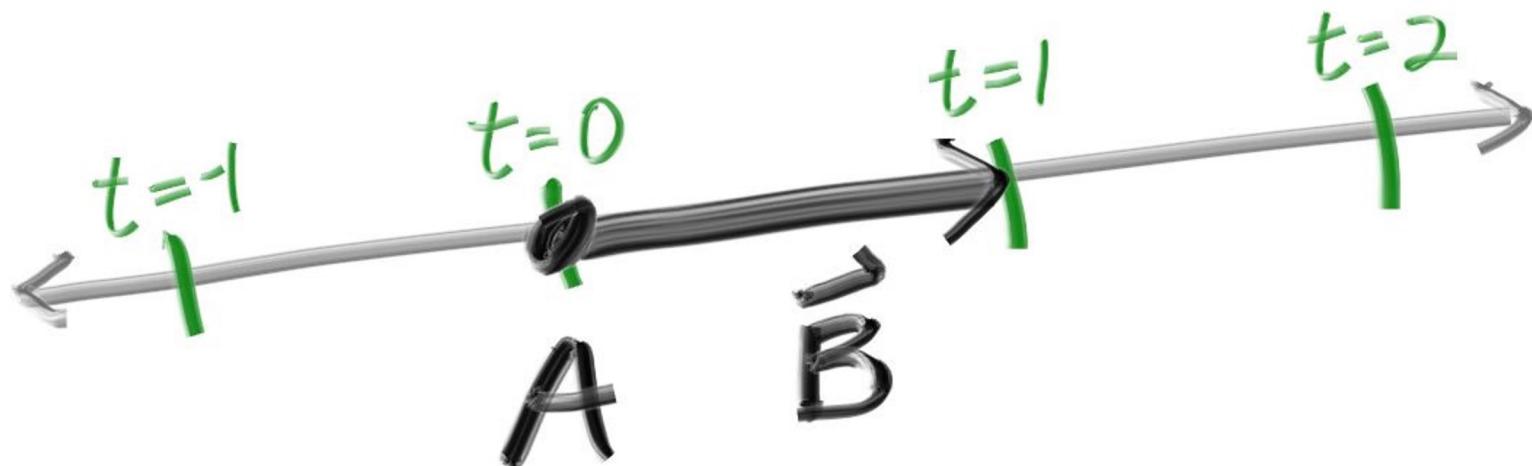
$$P(t) = A + tb$$

A é a origem do raio

b é um vetor que define a direção do raio

t é um parâmetro real

P(t) é o ponto sobre o raio



# Lançando os Raios na Cena

Vamos lançar raios pelos pixels e calcular a cor vista na direção desses raios. As etapas envolvidas são:

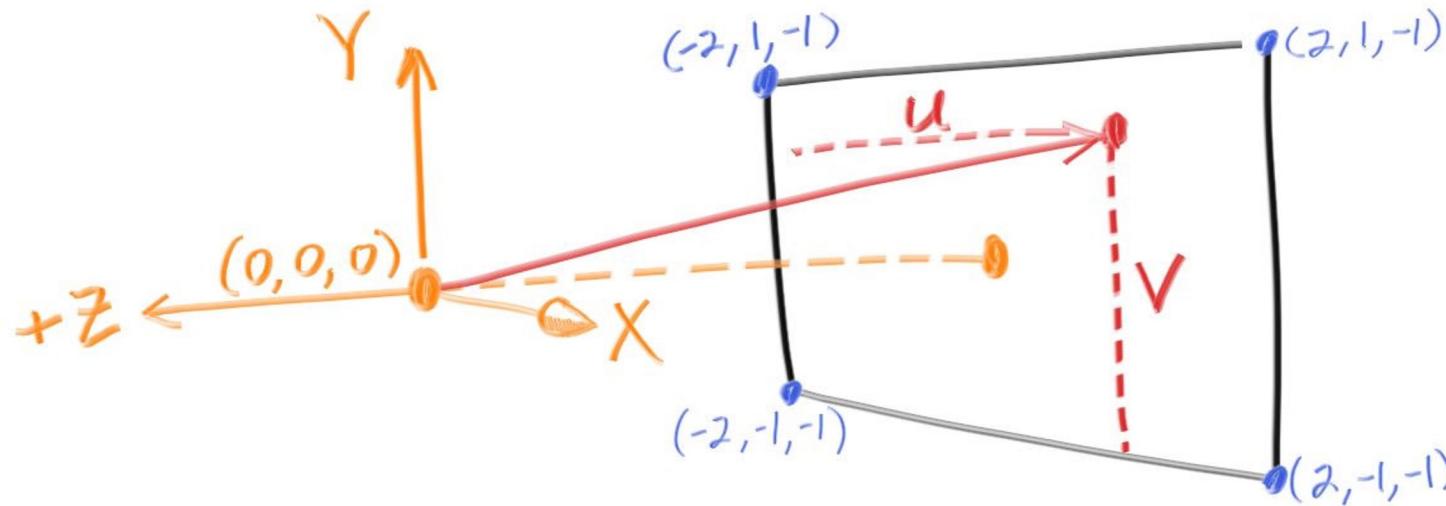
1. calcular o raio do ponto de vista até o pixel;
2. determinar quais objetos o raio faz a interseção;
3. calcular uma cor para esse ponto de interseção.

# Criando câmera

Vamos deixar o ponto de vista no  $(0,0,0)$  olhando para o Z negativo.

Varredura:

- Em CPU teríamos de passar por um loop pixel por pixel
- Em GPU cada pixel automaticamente roda em paralelo



# Interpolações

Iremos fazer uma interpolação linear do branco para o azul para o fundo de tela, representando o céu.

$$\text{blendedValue} = (1 - t) \cdot \text{startValue} + t \cdot \text{endValue}$$

t variando de 0 a 1



```
struct Ray{
    vec3 origin, direction;
};

vec3 color(Ray r){
    vec3 unit_direction = normalize(r.direction);
    float t = 0.5 * (unit_direction.y + 1.0);
    return mix(vec3(1.0), vec3(0.5,0.7,1.0), t);
}

void mainImage( out vec4 fragColor, in vec2 fragCoord ) {
    vec2 uv = fragCoord/iResolution.xy;

    float aspect = iResolution.x/iResolution.y;
    vec3 lower_left_corner = vec3(-1.0*aspect,-1.0, -1.0);
    vec3 horizontal = vec3(2.0*aspect, 0.0, 0.0);
    vec3 vertical = vec3(0.0,2.0, 0.0);
    Ray r = Ray(vec3(0,0,0), lower_left_corner+uv.x*horizontal+uv.y*vertical);

    fragColor = vec4(color(r), 1.0);
}
```

# Resultado no Shadertoy

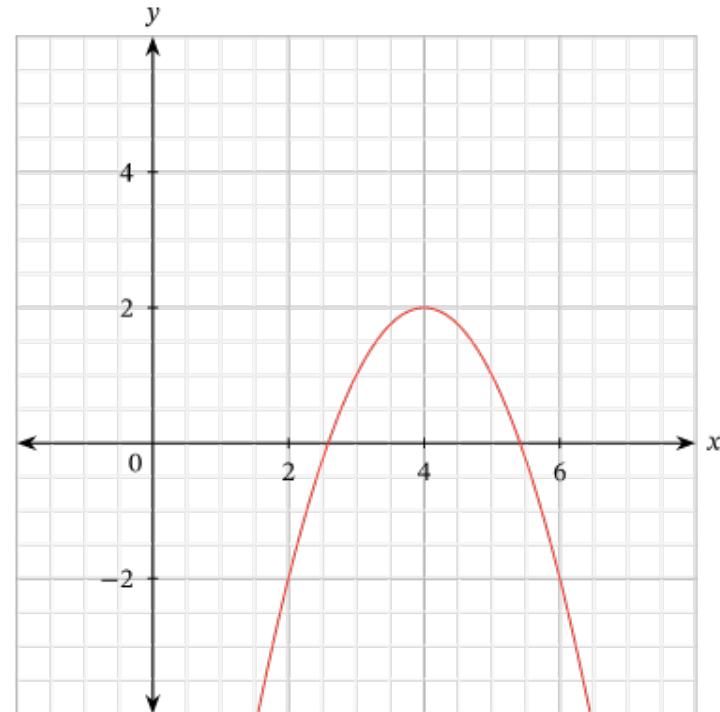


# Fórmula de Bhaskara

Método para encontrar as raízes reais de uma equação do segundo grau através de seus coeficientes.

$$\Delta = b^2 - 4ac$$

$$X = \frac{-b \pm \sqrt{\Delta}}{2a}$$



# Adicionando uma Esfera

Esferas são objetos comuns em RayTracer devido à simplicidade de calcular a intersecção de um raio com uma esfera.

$$x^2 + y^2 + z^2 = R^2$$

Assim, para um ponto  $(x, y, z)$  sobre a esfera a equação acima precisa ser verdadeira.

Para um ponto dentro da esfera  $x^2 + y^2 + z^2 < R^2$

Para um ponto fora da esfera  $x^2 + y^2 + z^2 > R^2$

# Adicionando uma Esfera

Para uma esfera centrada em  $(C_x, C_y, C_z)$ :

$$(x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2 = r^2$$

Para um ponto P na superfície, podemos calcular o vetor  $(P-C)$ , logo:

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

Assim a equação da esfera na forma vetorial é:

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

# Adicionando uma Esfera

Desejamos saber se o raio  $\mathbf{P}(t) = \mathbf{A} + t\mathbf{b}$  intersecta a esfera, assim:

$$(\mathbf{P}(t) - \mathbf{C}) \cdot (\mathbf{P}(t) - \mathbf{C}) = r^2$$

Ou expandindo:

$$(\mathbf{A} + t\mathbf{b} - \mathbf{C}) \cdot (\mathbf{A} + t\mathbf{b} - \mathbf{C}) = r^2$$

$$t^2\mathbf{b} \cdot \mathbf{b} + 2t\mathbf{b}(\mathbf{A} - \mathbf{C}) + (\mathbf{A} - \mathbf{C}) \cdot (\mathbf{A} - \mathbf{C}) - r^2 = 0$$

# Adicionando uma Esfera

Podemos usar o Bhaskara para encontrar as raízes da equação de segundo grau.

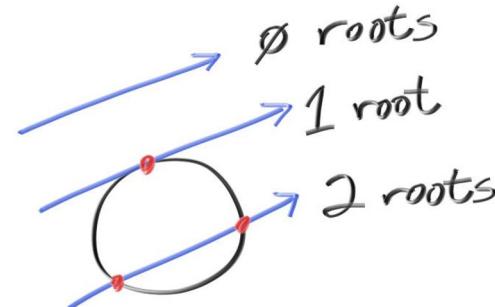
$$\Delta = b^2 - 4ac \quad X = \frac{-b \pm \sqrt{\Delta}}{2a}$$

$$t^2 \mathbf{b} \cdot \mathbf{b} + 2t \mathbf{b}(\mathbf{A} - \mathbf{C}) + (\mathbf{A} - \mathbf{C}) \cdot (\mathbf{A} - \mathbf{C}) - r^2 = 0$$

$\underbrace{\phantom{000}}_a \quad \underbrace{\phantom{000}}_b \quad \underbrace{\phantom{000000}}_c$

Se o delta:

- positivo: temos duas raízes
- zero: temos uma raiz
- negativo: não temos solução



# Uma esfera vermelha

```
bool hit_sphere(vec3 center, float radius, Ray r){  
    vec3 oc = r.origin - center;  
    float a = dot(r.direction,r.direction);  
    float b = 2.0 * dot(oc, r.direction);  
    float c = dot(oc,oc) - radius*radius;  
    float discriminat = b*b - 4.0*a*c;  
    return (discriminat > 0.0);  
}  
  
vec3 color(Ray r){  
    if(hit_sphere(vec3(0.,0.,-1.), 0.5, r)){  
        return vec3(1.,0.,0.);  
    }  
    vec3 unit_direction = normalize(r.direction);  
    float t = 0.5 * (unit_direction.y + 1.0);  
    return mix(vec3(1.0), vec3(0.5,0.7,1.0), t);  
}
```

$$t^2 \mathbf{b} \cdot \mathbf{b} + 2t \mathbf{b}(\mathbf{A} - \mathbf{C}) + (\mathbf{A} - \mathbf{C}) \cdot (\mathbf{A} - \mathbf{C}) - r^2 = 0$$

# Desenhando a esfera



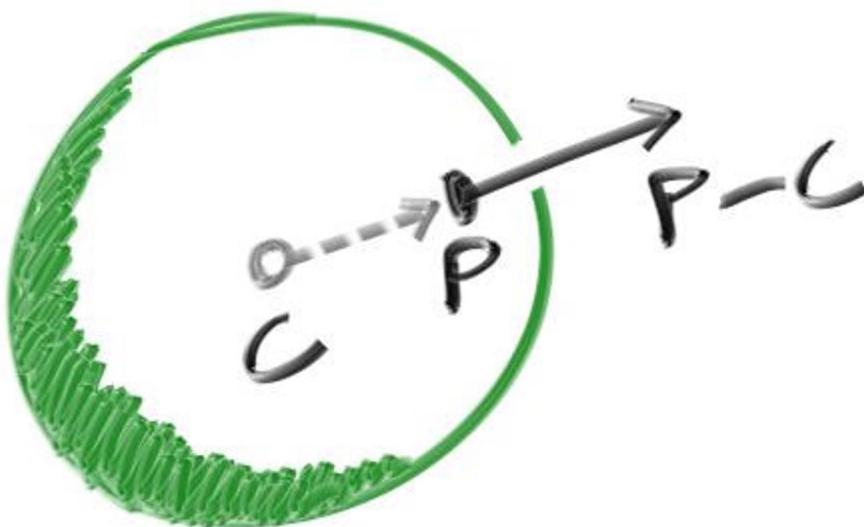
**Dois problemas:**

- Uma esfera atrás da câmera também apareceria
- Não estamos tratando os tons de cores devido a efeitos de iluminação

# Normais na Superfície

Para descobrir o vetor normal em um ponto na superfície de uma esfera, basta subtrair esse ponto do centro da esfera:

$$\vec{N} = (\mathbf{P} - \mathbf{C})$$



# Calculando as Normais

```
float hit_sphere(vec3 center, float radius, Ray r){  
    vec3 oc = r.origin - center;  
    float a = dot(r.direction,r.direction);  
    float b = 2.0 * dot(oc, r.direction);  
    float c = dot(oc,oc) - radius*radius;  
    float discriminat = b*b - 4.0*a*c;  
    if (discriminat < 0.0)  
        return(-1.0);  
    else  
        return( (-b - sqrt(discriminat)) / (2.0*a));  
}  
  
vec3 point_at_parameter(Ray r, float t) {  
    return(r.origin + t*r.direction);  
}  
  
vec3 color(Ray r){  
    float t = hit_sphere(vec3(0.,0.,-1.), 0.5, r);  
    if(t > 0.0) {  
        vec3 N = normalize(point_at_parameter(r,t) - vec3(0.0, 0.0, -1.0));  
        return 0.5*(N+vec3(1.0));  
    }  
    vec3 unit_direction = normalize(r.direction);  
    t = 0.5 * (unit_direction.y + 1.0);  
    return mix(vec3(1.0), vec3(0.5,0.7,1.0), t);  
}
```

# Esfera exibindo valores das normais

Os valores das normais ( $x, y, z$ ) foram mapeados para as cores na superfície ( $r, b, g$ ).



# Simplificando o código de intersecção de raios

$$t^2 \mathbf{b} \cdot \mathbf{b} + \boxed{2t\mathbf{b}(\mathbf{A} - \mathbf{C})} + \boxed{(\mathbf{A} - \mathbf{C}) \cdot (\mathbf{A} - \mathbf{C})} - r^2 = 0$$

- Primeiro: o produto escalar de um vetor por ele mesmo é igual ao quadrado do módulo daquele vetor.
- Segundo: que o valor de  $b$  é uma multiplicação de 2, assim podemos colocar em evidência esse fator 2 e simplificar.

$$\begin{aligned}& \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \\&= \frac{-2h \pm \sqrt{(2h)^2 - 4ac}}{2a} \\&= \frac{-2h \pm 2\sqrt{h^2 - ac}}{2a} \\&= \frac{-h \pm \sqrt{h^2 - ac}}{a}\end{aligned}$$

# Abstração para vários objetos intersectáveis

Vamos ter organizar o algoritmo para renderizar diversos objetos, contudo GLSL não é uma linguagem Orientada a Objetos, como o livro de referência propõe. Assim vamos criar uma lista global.

```
struct Sphere {  
    vec3 center;  
    float radius;  
};  
  
Sphere world[] = Sphere[2] (  
    Sphere(vec3(0.0, 0.0, -1.0), 0.5),  
    Sphere(vec3(0.0, -100.5, -1.0), 100.0)  
,
```

# Objetos intersectáveis

Vamos criar uma estrutura para organizar os objetos intersectáveis (no momento esferas).

Vamos testar se a intersecção ocorre entre um  $t_{\min}$  e  $t_{\max}$ .

Temos de limitar a intersecção mais próxima da origem do raio.

```
struct hit_record {
    float t;
    vec3 p, normal;
};
```

# Testando esfera pelo ponto de contato

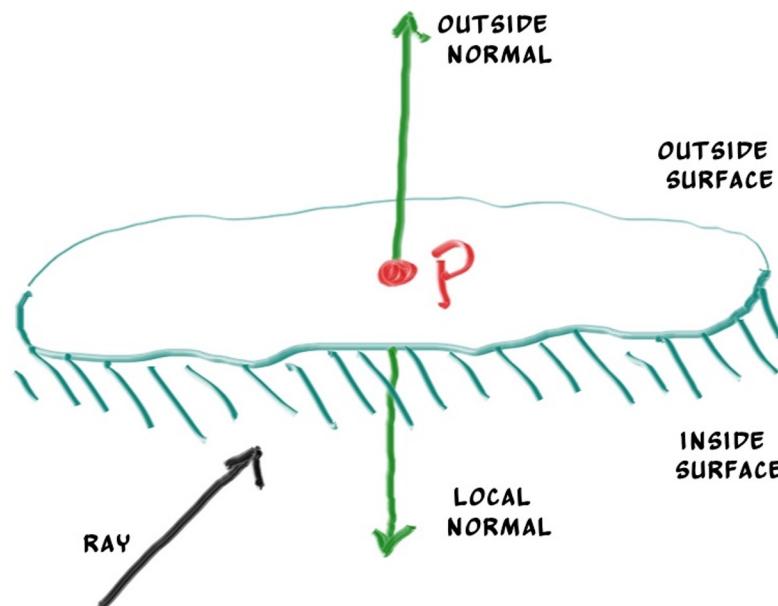
Vamos atualizar a rotina da esfera para usar a nova estrutura.

```
bool hit_sphere(Sphere s, Ray r, float t_min, float t_max, out hit_record rec) {
    vec3 oc = r.origin - s.center;
    float a = dot(r.direction, r.direction);
    float half_b = dot(oc, r.direction);
    float c = dot(oc, oc) - s.radius*s.radius;
    float discriminat = half_b*half_b - a*c;
    if (discriminat < 0.0) return(false);

    float sqrtD = sqrt(discriminat);
    float root = (-half_b - sqrtD) / a;
    if (root < t_min || t_max < root) {
        root = (-half_b + sqrtD) / a;
        if (root < t_min || t_max < root) return false;
    }
    rec.t = root;
    rec.p = point_at_parameter(r, rec.t);
    rec.normal = (rec.p - s.center) / s.radius;
    return(true);
}
```

# Face frontal e traseira

Os raios lançados podem intersectar as superfícies dos objetos pelo lado interno ou externo, para obter a direção basta fazer uma operação de produto escalar dos vetores.



# Tratando a face do objeto

```
vec3 set_face_normal(Ray r, vec3 outward_normal) {
    bool front_face = dot(r.direction, outward_normal) < 0.0;
    return front_face ? outward_normal : -outward_normal;
}

bool hit_sphere(Sphere s, Ray r, float t_min, float t_max, out hit_record rec){
    vec3 oc = r.origin - s.center;
    float a = dot(r.direction,r.direction);
    float half_b = dot(oc, r.direction);
    float c = dot(oc,oc) - s.radius*s.radius;
    float discriminat = half_b*half_b - a*c;
    if (discriminat < 0.0) return(false);

    float sqrt_d = sqrt(discriminat);
    float root = (-half_b - sqrt_d) / a;
    if (root < t_min || t_max < root) {
        root = (-half_b + sqrt_d) / a;
        if (root < t_min || t_max < root) return false;
    }
    rec.t = root;
    rec.p = point_at_parameter(r,rec.t);
    vec3 outward_normal = (rec.p - s.center) / s.radius;
    rec.normal = set_face_normal(r, outward_normal); return(true);
    return(true);
}
```

# Tratando todos os objetos

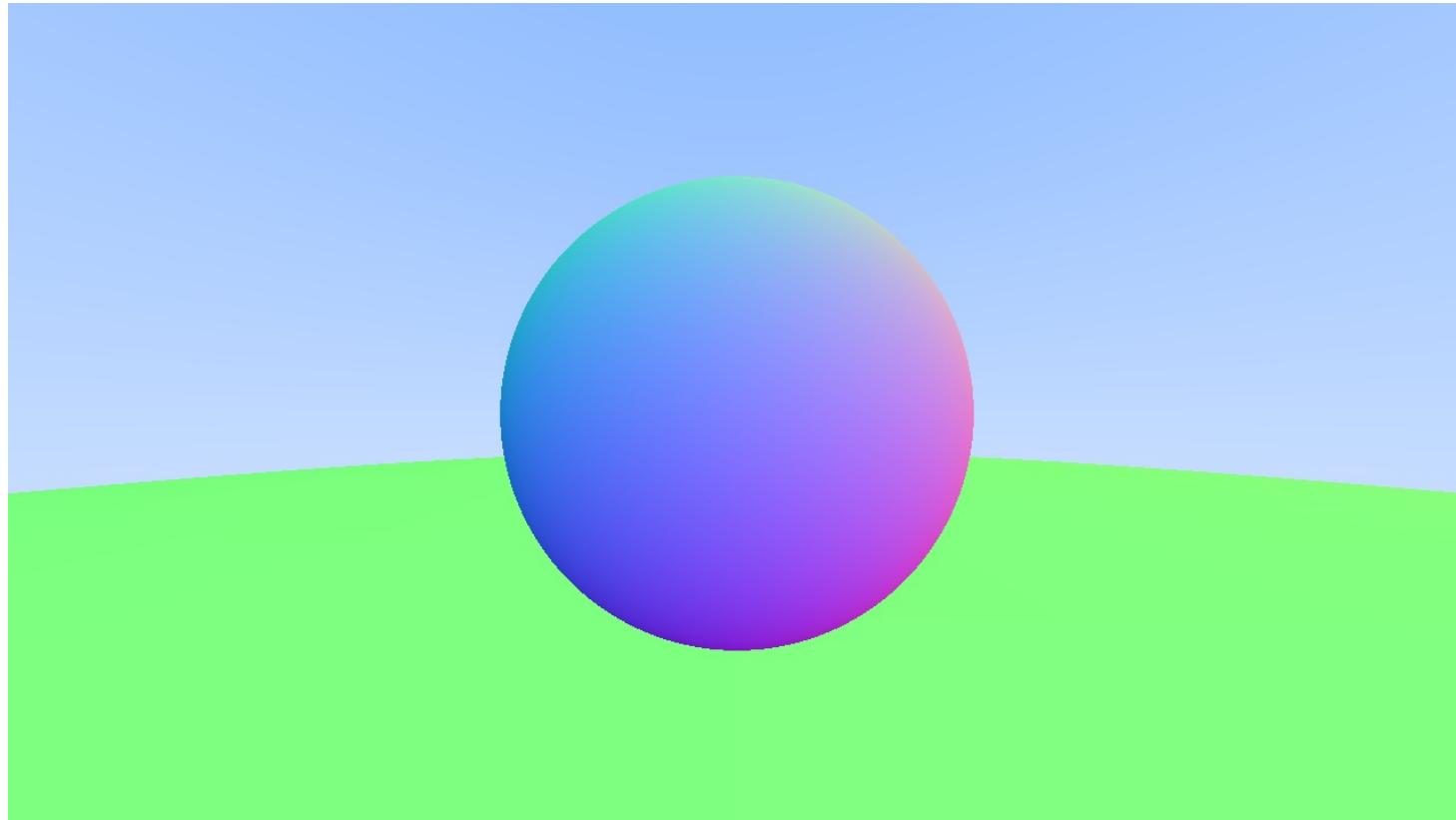
Vamos atualizar as rotinas para desenhar várias esferas.

```
bool hitable_list(Ray r, float t_min, float t_max, inout hit_record rec) {
    hit_record temp_rec;
    bool hit_anything = false;
    float closest_so_far = t_max;
    for(int i=0; i<world.length(); i++) {
        if(hit_sphere(world[i], r, t_min, closest_so_far, temp_rec)) {
            hit_anything = true;
            closest_so_far = temp_rec.t;
            rec = temp_rec;
        }
    }
    return hit_anything;
}

vec3 color(Ray r){
    hit_record rec;
    if(hitable_list(r, 0.0, 10000.0, rec)) {
        return 0.5*(rec.normal+vec3(1.0));
    }
    vec3 unit_direction = normalize(r.direction);
    float t = 0.5 * (unit_direction.y + 1.0);
    return mix(vec3(1.0), vec3(0.5,0.7,1.0), t);
}
```

# Resultado da organização do código

Se tudo correu bem, você deve ter a seguinte imagem.



# Antialiasing

Imagen atual está gerando artefatos nas bordas dos objetos.  
Vamos fazer um antialiasing para reduzir esse problema.



Vamos lançar vários raios por pixel?

# Gerador de Números Aleatórios

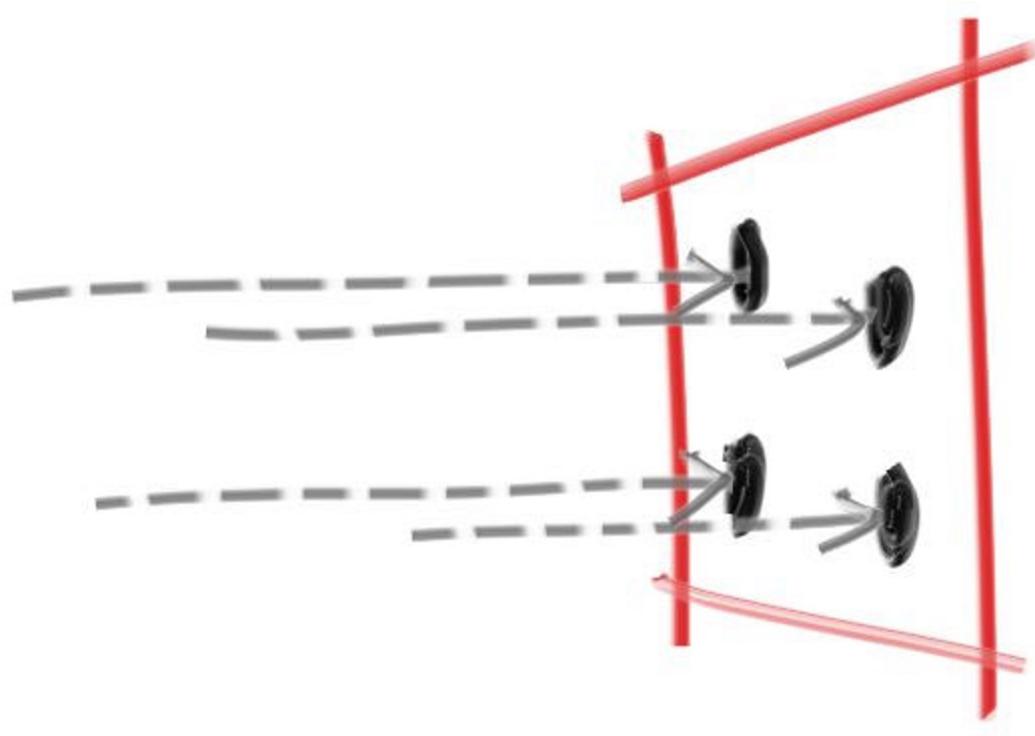
Precisamos de uma rotina que gere números reais na faixa:  
 $0 \leq r < 1$

O GLSL não possui uma função de números aleatórios de forma nativa. Assim temos de fazer uma. A seguinte função é uma variação das funções de números aleatórios do Book of Shaders.

```
vec2 rand(float co) {
    vec2 tmp = vec2( sin(co*12.9898) ,
                     sin(co*78.233)) ;
    return fract(tmp * vec2(43758.5453, 72649.2791)) ;
}
```

# Várias amostras por pixel

Vamos lançar vários raios por pixel e depois fazer uma média.

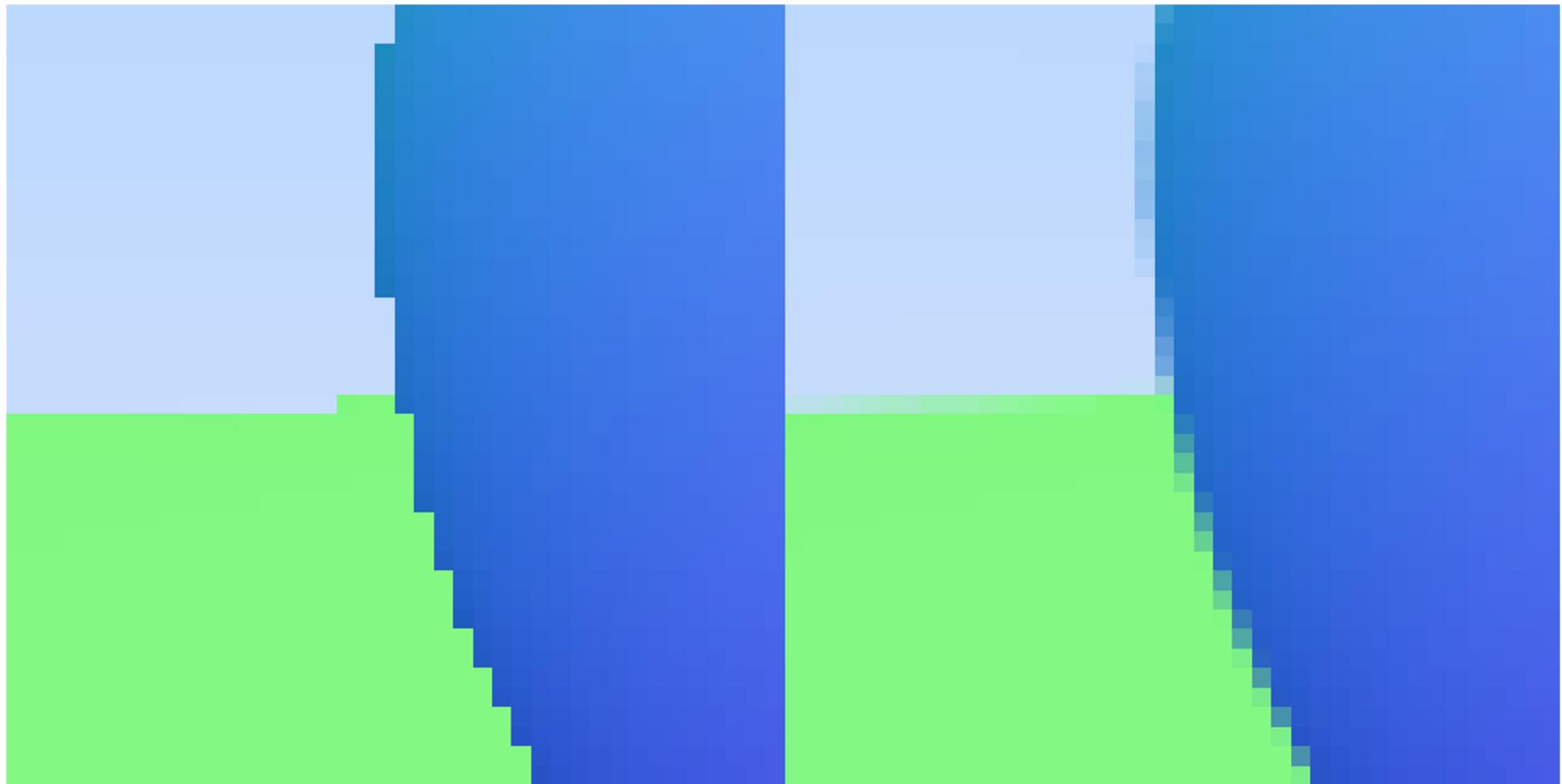


# Lançando vários raios

```
Ray get_ray(vec2 uv) {
    float aspect = iResolution.x/iResolution.y;
    vec3 lower_left_corner = vec3(-1.0*aspect,-1.0, -1.0);
    vec3 horizontal = vec3(2.0*aspect, 0.0, 0.0);
    vec3 vertical = vec3(0.0,2.0, 0.0);
    Ray r = Ray(vec3(0,0,0), lower_left_corner+uv.x*horizontal+uv.y*vertical);
    return r;
}

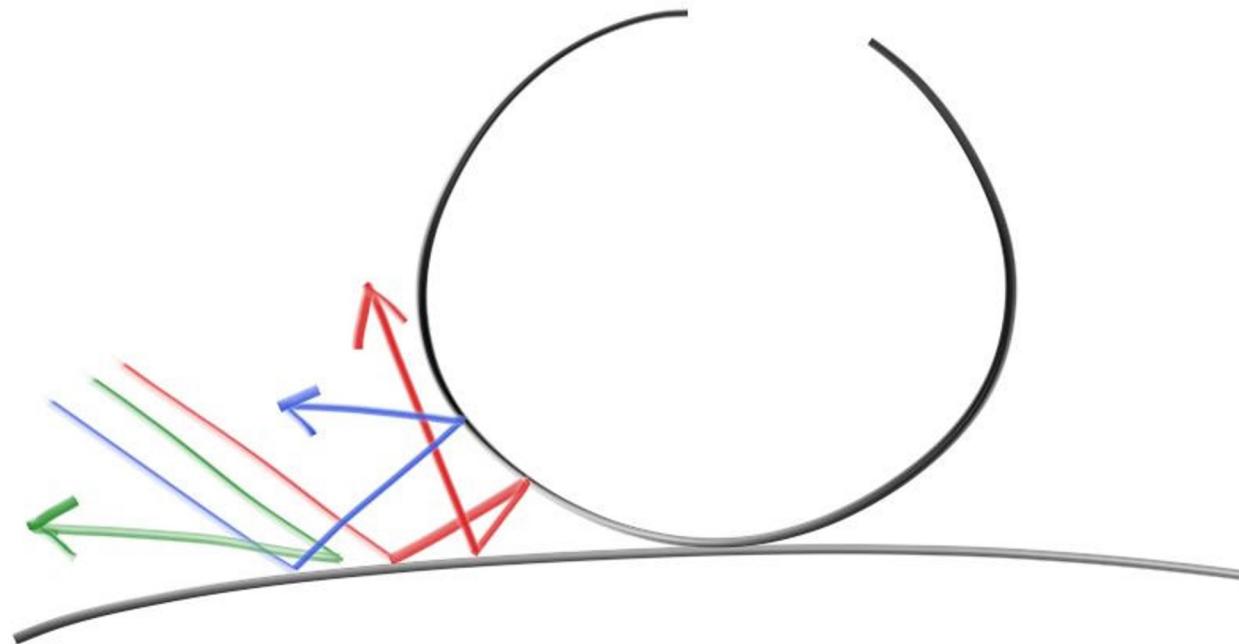
const float samples_per_pixel = 10.0;
void mainImage( out vec4 fragColor, in vec2 fragCoord ) {
    vec2 uv = fragCoord/iResolution.xy;
    vec3 col = vec3(0.0);
    for(float s=0.0; s<samples_per_pixel; ++s) {
        vec2 delta;
        delta = rand(s) / iResolution.xy;
        Ray r = get_ray(uv+delta);
        col += color(r);
    }
    col /= samples_per_pixel;
    fragColor = vec4(col, 1.0);
}
```

# Antes e depois do Antialiasing



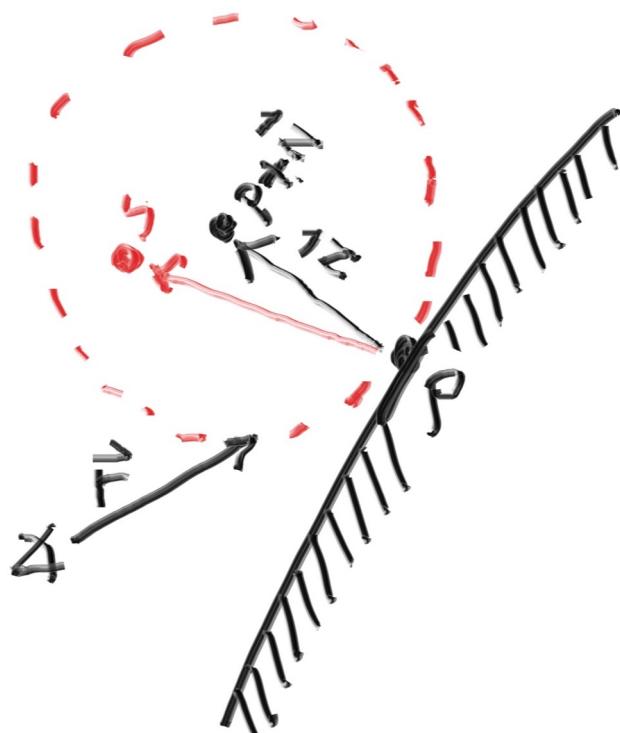
# Materiais difusos

Objetos difusos que não emitem luz apenas assumem a cor do ambiente. A luz que reflete em uma superfície difusa tem sua direção com certa aleatoriedade.



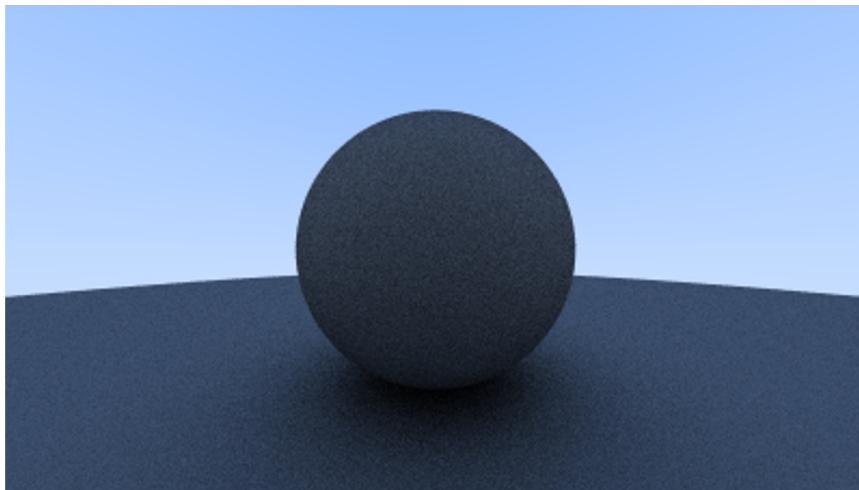
# Descobrindo onde refletir

Imagine o volume de uma esfera do mesmo lado do ponto de intersecção do raio. Ache um ponto dentro dessa esfera e lance o novo raio de reflexão nessa direção.

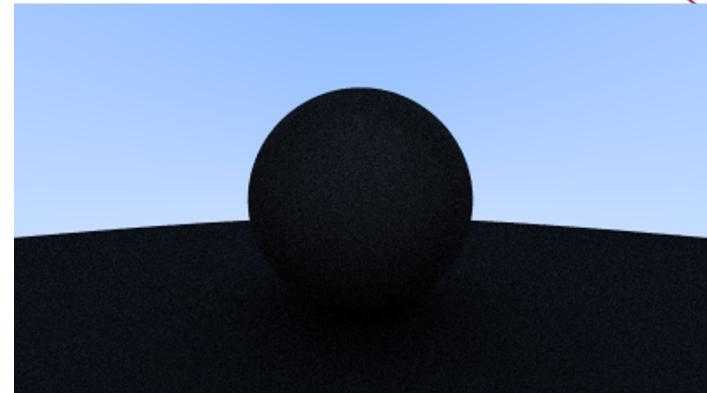


# Resultado

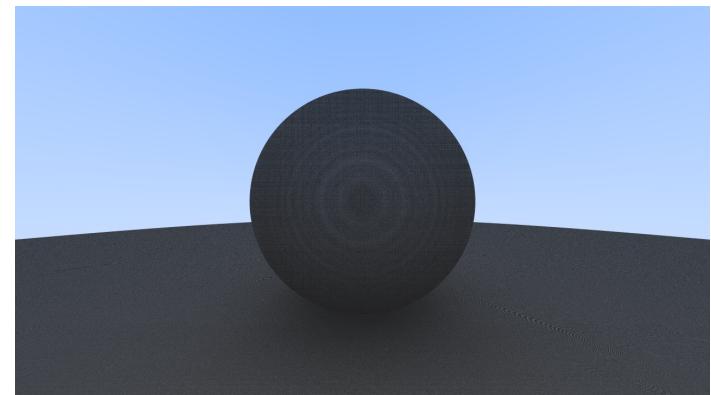
Parece que tem algo estranho.



Autor



ou



Professor

**Shadow Acne - Os raios intersectam diretamente com a própria geometria.  
ou seja, um  $t=0.0001$  de inicio de raio resolve!**

# Geração de Números aleatórios para raios

Geração de números aleatórios em GPU é complicado. Se for para gerar múltiplos raios mais complicado ainda . A proposta de nimitz é bem interessante:

<https://www.shadertoy.com/view/Xt3cDn>



# Lançando vários raios refletidos

```
vec3 frand;
vec3 rand = vec3(0.5, 0.5, 0.5);
vec2 rand2(){
    vec3 rand = vec3(
        mod(rand.y*(195.1*frand.z+371.2*frand.x+508.3*frand.y), 1.0),
        mod(rand.x*(573.9*frand.z+736.4*frand.y+914.5*frand.x), 1.0),
        rand.z
    );
    return fract(rand).xy-0.5;
}

vec3 random_in_unit_sphere(){
    rand = vec3(
        mod(rand.z*((57.1*frand.z)+(57.1*frand.x)+(759.7*frand.y))*3.83,2.0),
        mod(rand.x*((41.1*frand.z)+(65.7*frand.x)+(621.9*frand.y))*1.57,2.0),
        mod(rand.y*((33.1*frand.z)+(45.3*frand.x)+(557.2*frand.y))*2.34,2.0)
    );
    return (rand-vec3(1.0));
}
```

# Guardando o número aleatório

```
const float samples_per_pixel = 100.0;
void mainImage( out vec4 fragColor, in vec2 fragCoord ) {
    vec2 uv = fragCoord/iResolution.xy;
    vec3 col = vec3(0.0);
    for(float s=0.0; s<samples_per_pixel; ++s) {
        frand = vec3(fragCoord.xy, s);
        vec2 delta;
        delta = rand2() / iResolution.xy;
        Ray r = get_ray(uv+delta);
        col += color(r);
    }
    col /= samples_per_pixel;
    fragColor = vec4(col, 1.0);
}
```

# Funções Recursivas (CPU)

```
color ray_color(const ray& r, const hittable& world) {
    hit_record rec;

    if (world.hit(r, 0, infinity, rec)) {
        point3 target = rec.p + rec.normal + random_in_unit_sphere();
        return 0.5 * ray_color(ray(rec.p, target - rec.p), world);
    }

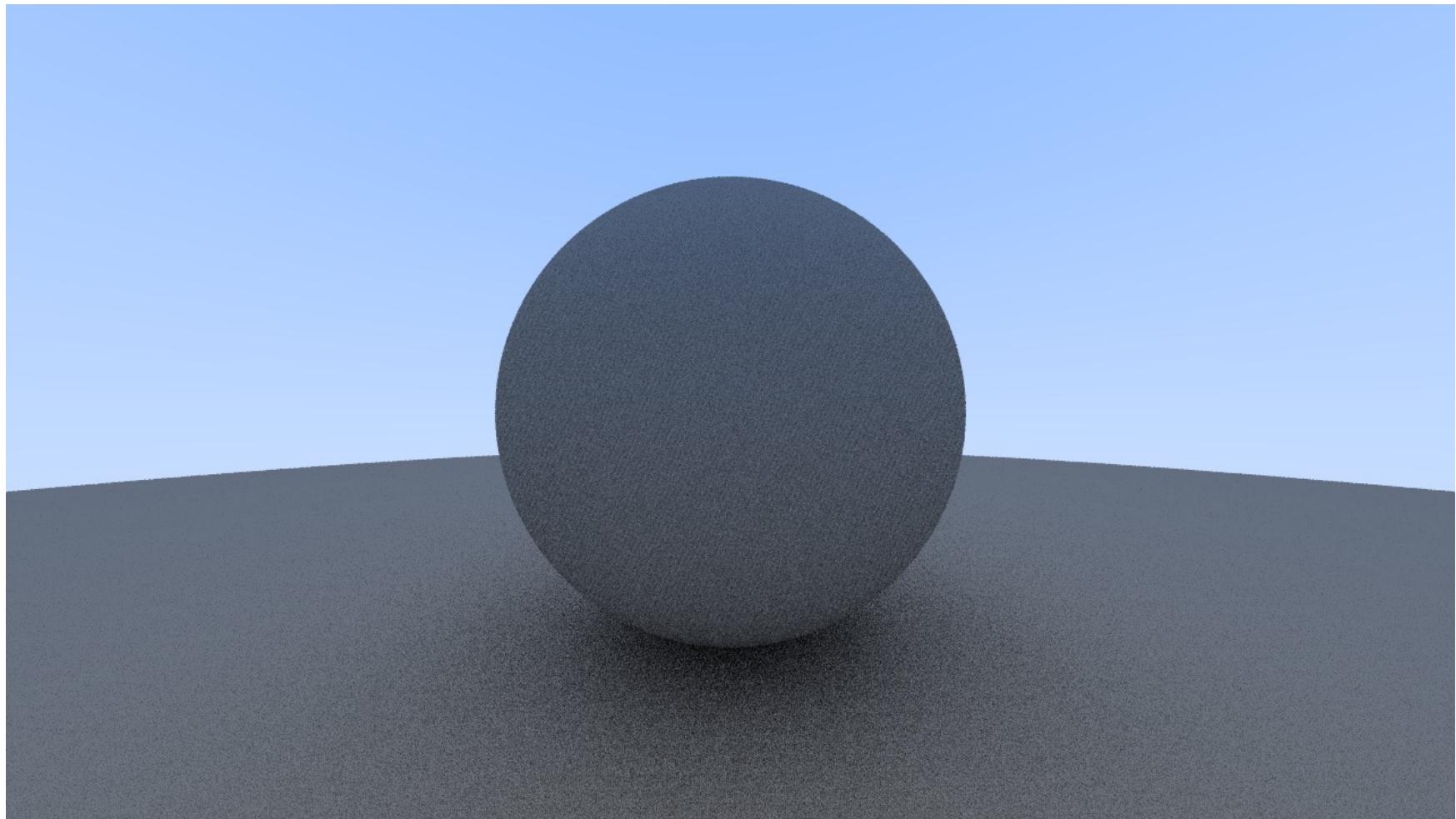
    vec3 unit_direction = unit_vector(r.direction());
    auto t = 0.5*(unit_direction.y() + 1.0);
    return (1.0-t)*color(1.0, 1.0, 1.0) + t*color(0.5, 0.7, 1.0);
}
```

do livro: RayTracingInOneWeekend

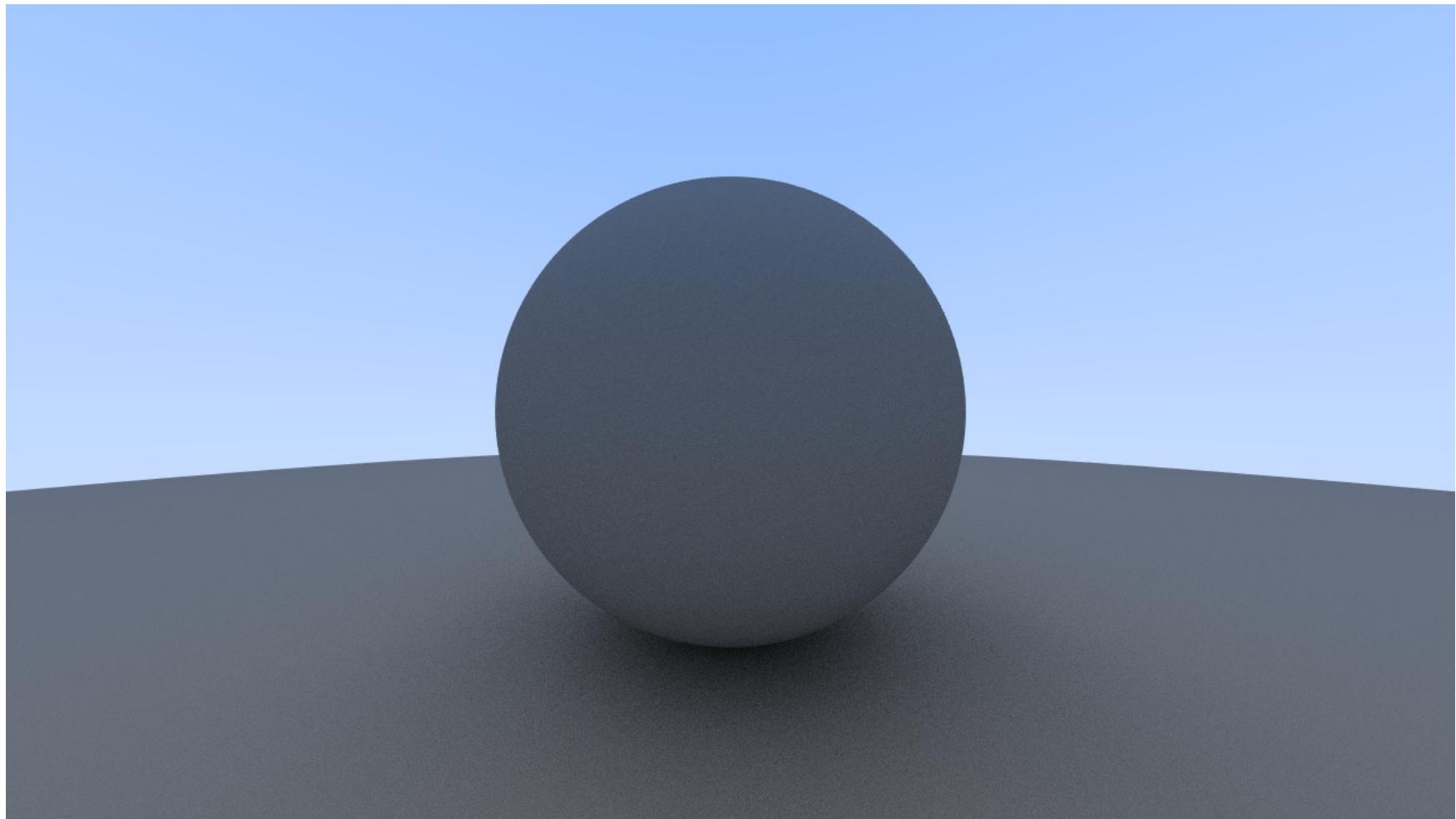
# Funções Recursivas (GPU)

```
const int max_depth = 10;
vec3 color(Ray r){
    hit_record rec;
    vec3 col = vec3(1.0);
    Ray raio = r;
    for(int i=0; i < max_depth; i++) {
        if(hitable_list(raio, 0.0001, 100.0, rec)) {
            vec3 target = rec.p + rec.normal + random_in_unit_sphere();
            col *= 0.5;
            raio = Ray(rec.p, target - rec.p);
        } else {
            vec3 unit_direction = normalize(r.direction);
            float t = 0.5 * (unit_direction.y + 1.0);
            col *= mix(vec3(1.0), vec3(0.5,0.7,1.0), t);
            return col;
        }
    }
    return col;
}
```

# Shadow Acne (10 amostras)



# Shadow Acne (100 amostras)



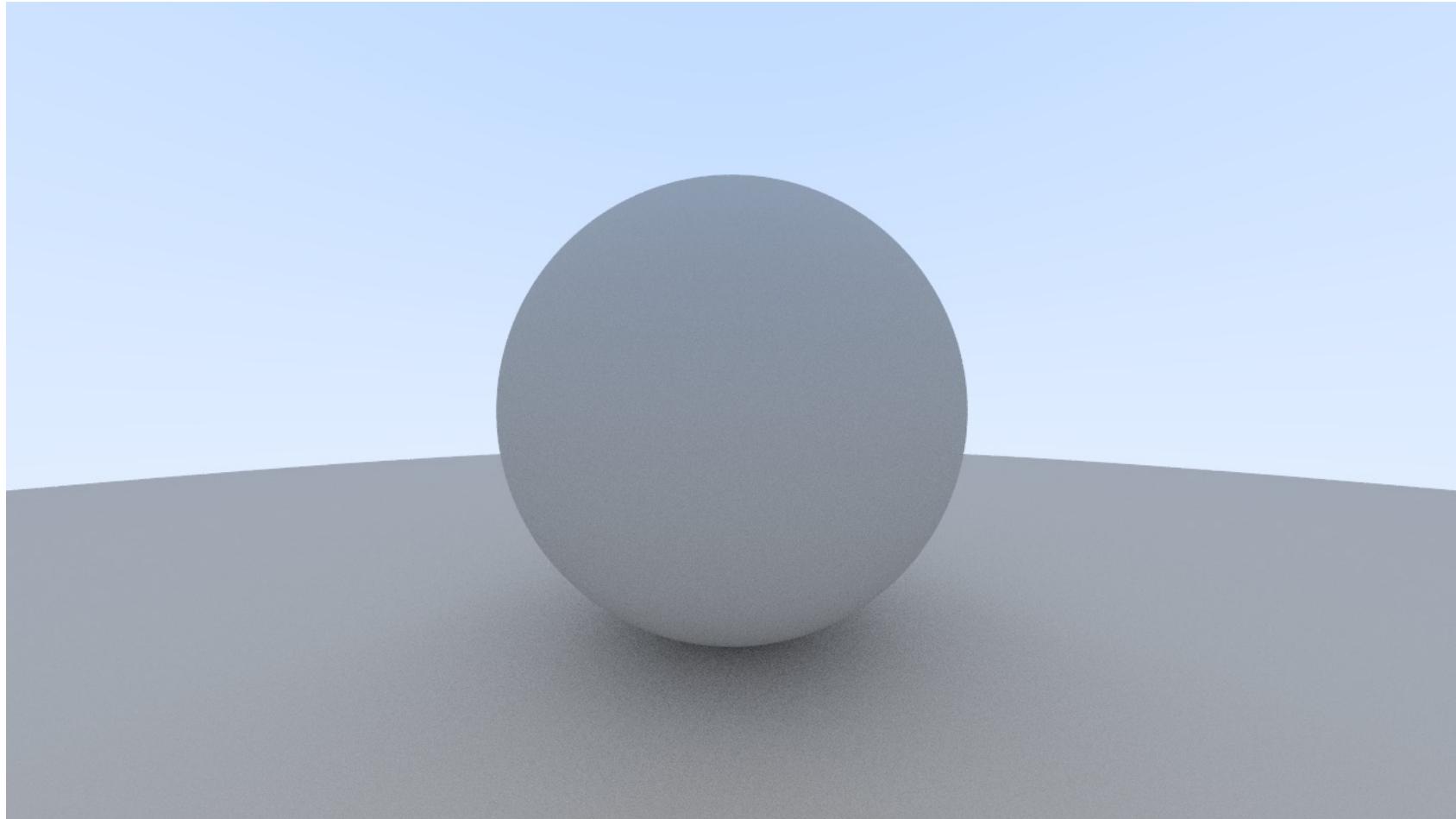
# Imagen corrigida pelo Gamma

Vamos usar um gamma = 2.0

Ou seja, vamos elevar o valor das cores em  $1/\text{gamma}$ .

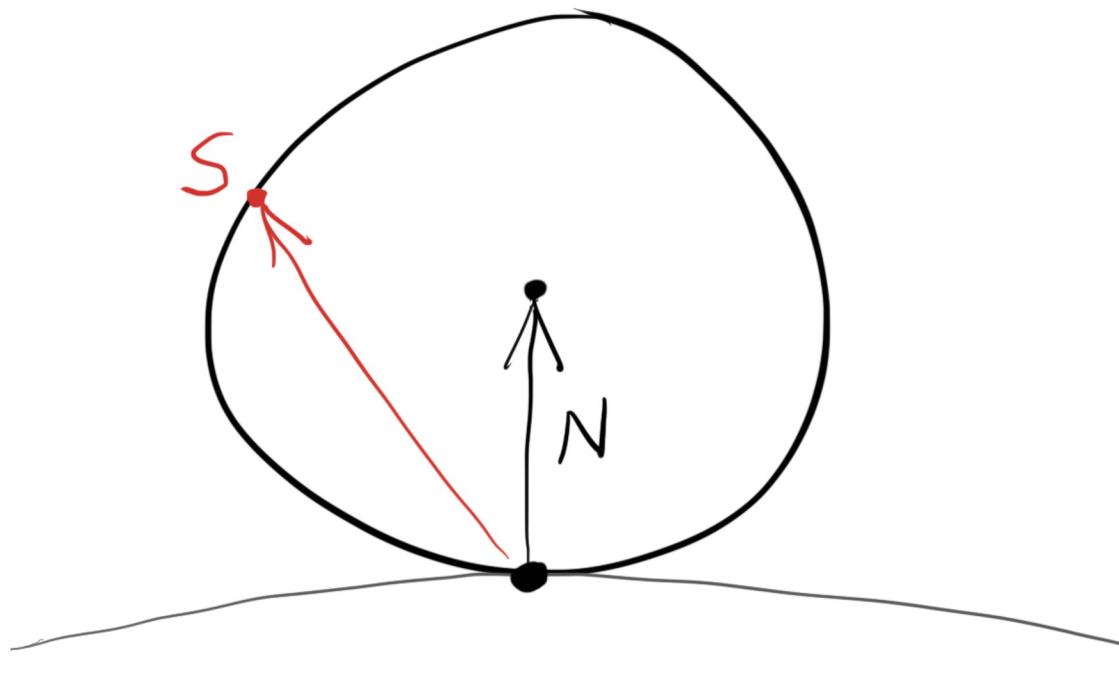
```
const float samples_per_pixel = 100.0;
void mainImage( out vec4 fragColor, in vec2 fragCoord ) {
    vec2 uv = fragCoord/iResolution.xy;
    vec3 col = vec3(0.0);
    for(float s=0.0; s<samples_per_pixel; ++s) {
        frand = vec3(fragCoord.xy, s);
        vec2 delta;
        delta = rand2() / iResolution.xy;
        Ray r = get_ray(uv+delta);
        col += color(r);
    }
    col /= samples_per_pixel;
    float gamma = 2.0;
    col = pow(col, vec3(1.0/gamma));
    fragColor = vec4(col, 1.0);
}
```

# Imagen com correção do Gamma



# Reflexão Lambertiana

A distribuição atual de raios atual segue uma função  $\cos^3(\phi)$ , sendo  $\phi$  o angulo da normal. Porém, a distribuição Lambertiana segue uma função de distribuição  $\cos(\phi)$ . Assim, os pontos a serem lançados estarão na superfície dessa esfera de referência.

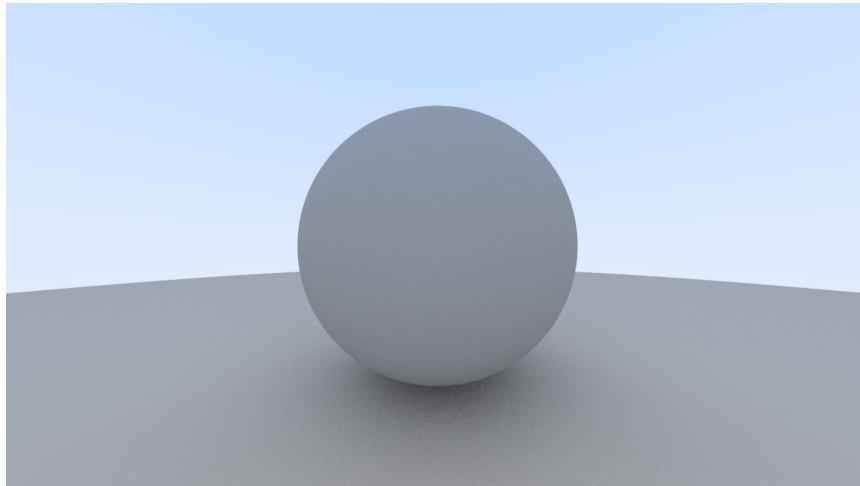


# Raios de forma uniforme

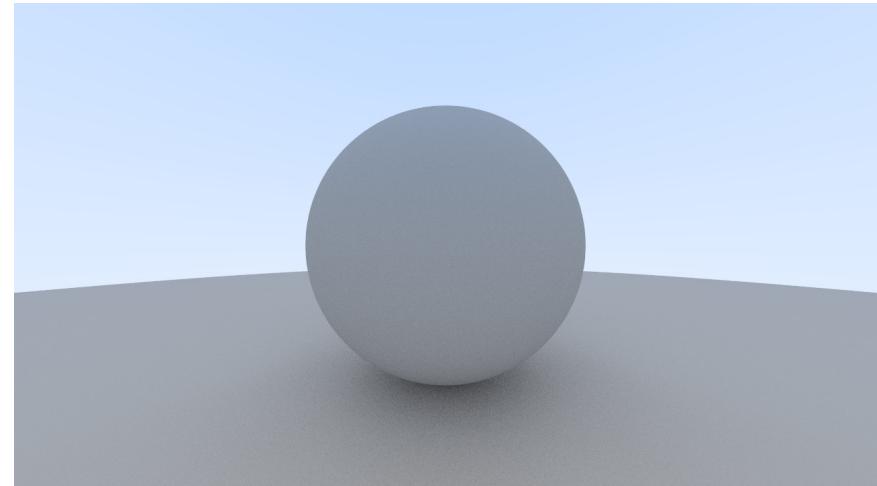
```
vec3 random_in_unit_sphere(){
    rand = vec3(
        mod(rand.z*((57.1*frand.z)+(57.1*frand.x)+(759.7*frand.y))*3.83,2.0),
        mod(rand.x*((41.1*frand.z)+(65.7*frand.x)+(621.9*frand.y))*1.57,2.0),
        mod(rand.y*((33.1*frand.z)+(45.3*frand.x)+(557.2*frand.y))*2.34,2.0)
    );
    return (rand-vec3(1.0));
}

vec3 random_unit_vector() {
    return normalize(random_in_unit_sphere());
}
```

# Resultado da Reflexão Lambertiana



Antes



Depois

- As sombras são menos pronunciadas após a mudança
- Ambas as esferas ficam mais claras na aparência após a mudança

# Espalhamento por Hemisfera

A última proposta apresentada ainda não é a ideal, pois os raios devem se espalhar em todas as direções de forma uniforme.

A próxima proposta é lançar um raio de uma distância unitária do ponto de intersecção e só garantir que ele está no mesmo hemisfério da normal.

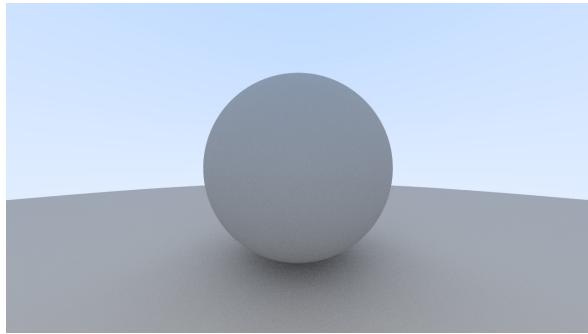
# Raios de forma uniforme

```
vec3 random_in_unit_sphere() {
    rand = vec3(
        mod(rand.z*((57.1*frand.z)+(57.1*frand.x)+(759.7*frand.y))*3.83,2.0),
        mod(rand.x*((41.1*frand.z)+(65.7*frand.x)+(621.9*frand.y))*1.57,2.0),
        mod(rand.y*((33.1*frand.z)+(45.3*frand.x)+(557.2*frand.y))*2.34,2.0)
    );
    return (rand-vec3(1.0));
}

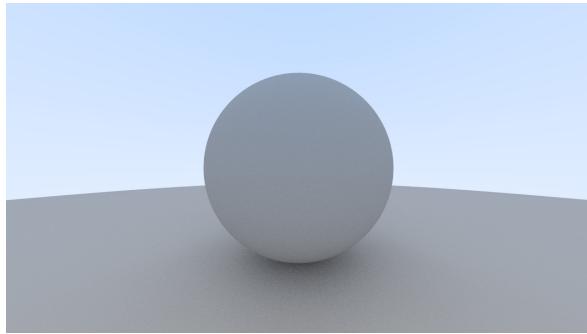
vec3 random_in_hemisphere(vec3 normal) {
    vec3 in_unit_sphere = random_in_unit_sphere();
    if (dot(in_unit_sphere, normal) > 0.0)
        return in_unit_sphere;
    else
        return -in_unit_sphere;
}

...
if(hitable_list(raio, 0.0001, 100.0, rec)) {
    vec3 target = rec.p + random_in_hemisphere(rec.normal);
    col *= 0.5;
}
...
```

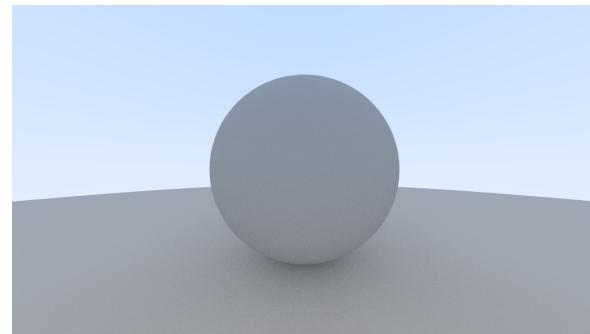
# Resultado da Reflexão Lambertiana Hemisférica



Inicial



Intermediário



Final

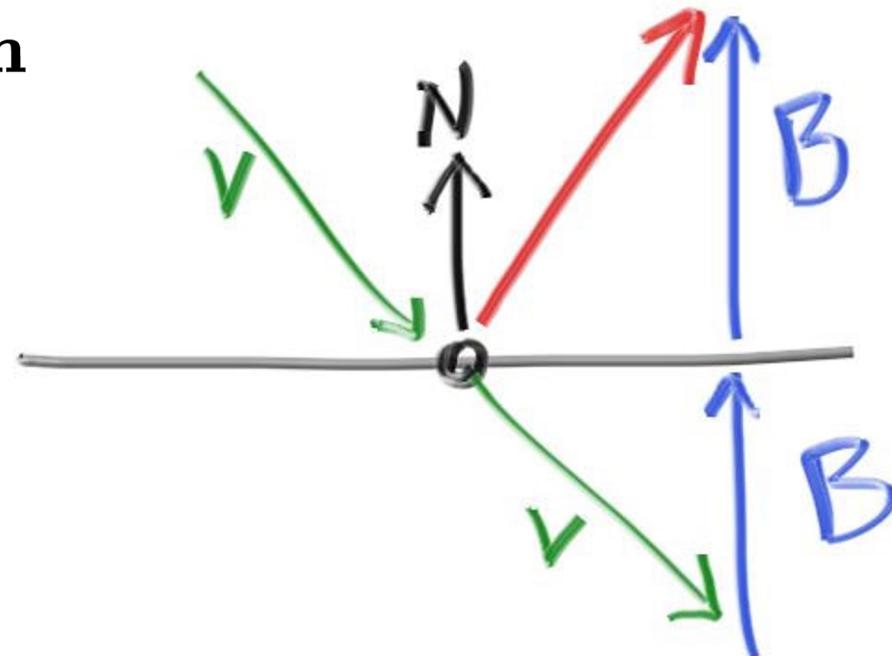
# Reflexão em Materiais Espelhados

Para calcular o vetor de reflexão (vermelho) somaremos o vetor de entrada ( $v$ ) por duas vezes  $B$ .

$B$  tem a direção e sentido de  $N$  com um comprimento de  $|v \cdot N|$ .

$$\mathbf{R} = \mathbf{v} - 2 \cdot (\mathbf{v} \cdot \mathbf{n}) \cdot \mathbf{n}$$

Em GLSL: `reflect()`



# Materiais 1/4

```
#define lambertian 0
#define metal 1

struct Material {
    vec3 albedo;
    int type;
};

struct hit_record {
    float t;
    vec3 p, normal;
    Material mat;
};

struct Sphere {
    vec3 center;
    float radius;
    Material mat;
};
```

# Materiais 2/4

```
bool hit_sphere(Sphere s, Ray r, float t_min, float t_max, out hit_record rec){  
    ...  
    rec.t = root;  
    rec.p = point_at_parameter(r, rec.t);  
    vec3 outward_normal = (rec.p - s.center) / s.radius;  
    rec.normal = normalize(set_face_normal(r, outward_normal));  
    rec.mat = s.mat;  
    return(true);  
}  
  
bool near_zero(vec3 e) {  
    float s = 1e-8;  
    return (abs(e[0]) < s) && (abs(e[1]) < s) && (abs(e[2]) < s);  
}
```

# Materiais 3/4

```
bool scatter(Material mat, Ray r_in, hit_record rec, out vec3 attenuation, out Ray scattered) {
    vec3 scatter_direction = rec.normal + random_unit_vector();
    if (near_zero(scatter_direction))
        scatter_direction = rec.normal;

    if(mat.type == 0) {
        scattered = Ray(rec.p, scatter_direction);
        attenuation = mat.albedo;
        return true;
    }

    if(mat.type == 1) {
        vec3 reflected = reflect(normalize(r_in.direction), rec.normal);
        scattered = Ray(rec.p, reflected);
        attenuation = mat.albedo;
        return(dot(scattered.direction, rec.normal) > 0.0);
    }

    return false;
}
```

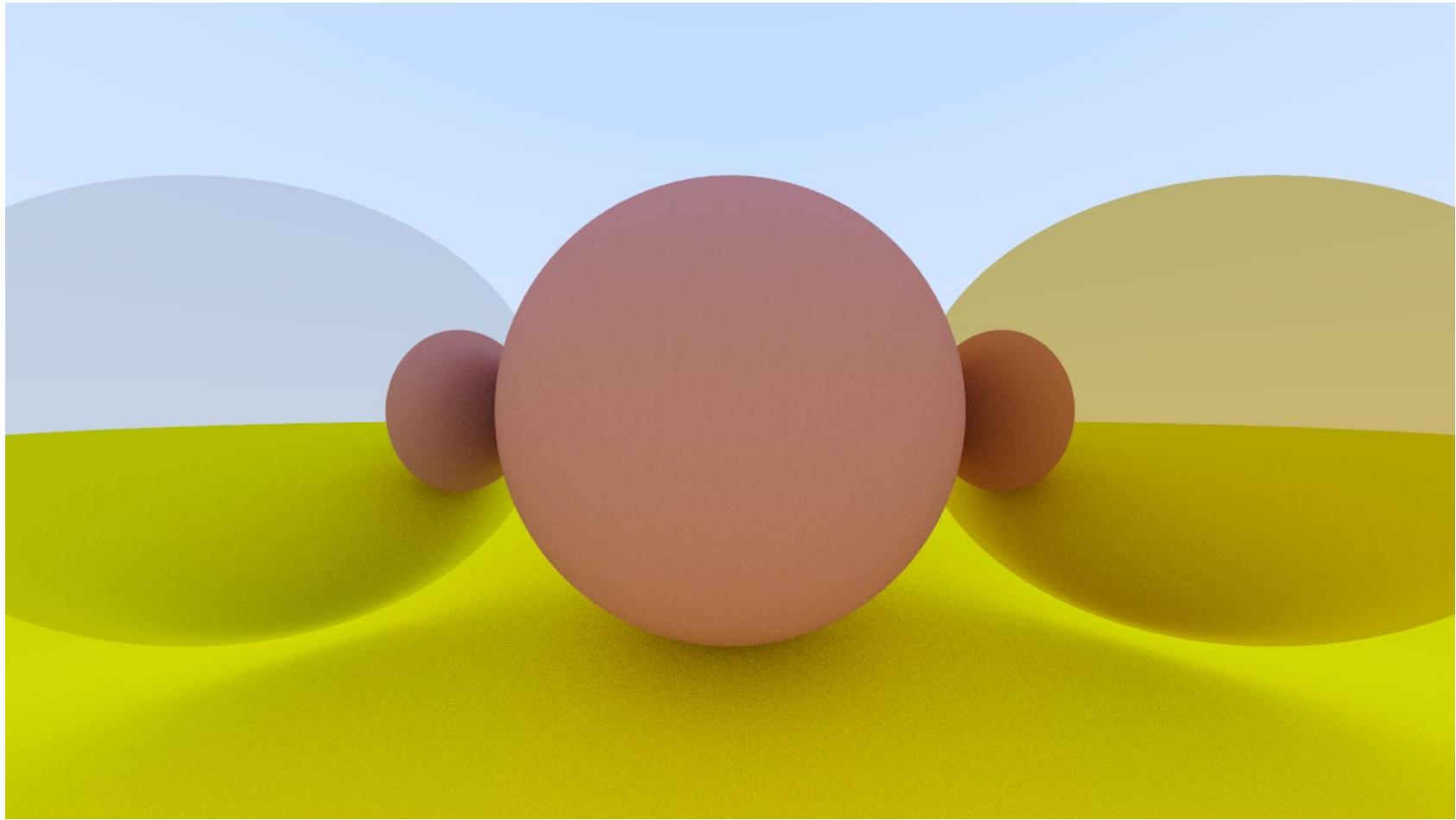
# Materiais 4/4

```
const int max_depth = 10;
vec3 color(Ray r){
    hit_record rec;
    vec3 col = vec3(1.0);
    Ray raio = r;
    for(int i=0; i < max_depth; i++){
        if(hitables_list(raio, 0.0001, 100.0, rec)) {
            Ray scattered;
            vec3 attenuation = vec3(0.0);
            if (scatter(rec.mat, r, rec, attenuation, scattered))
                raio = scattered;
            else break;
            col *= attenuation;
        } else {
            vec3 unit_direction = normalize(r.direction);
            float t = 0.5 * (unit_direction.y + 1.0);
            col *= mix(vec3(1.0), vec3(0.5,0.7,1.0), t);
            return col;
        }
    }
    return col;
}
```

# Nova Cena

```
Sphere world[] = Sphere[4](  
    Sphere(vec3( 0.0, -100.5, -1.0), 100.0, Material(vec3(0.8, 0.8, 0.0),0)),  
    Sphere(vec3( 0.0, 0.0, -1.0), 0.5, Material(vec3(0.7, 0.3, 0.3),0)),  
    Sphere(vec3(-1.0, 0.0, -1.0), 0.5, Material(vec3(0.8, 0.8, 0.8),1)),  
    Sphere(vec3( 1.0, 0.0, -1.0), 0.5, Material(vec3(0.8, 0.6, 0.2),1)))  
) ;
```

# Resultado das esferas metálicas



<https://www.shadertoy.com/view/DtV3Rc>

# Estratégias Avançadas de Ray Tracing

Os seguintes a frente são estratégias adicionais para implementar em um Ray Tracing.

Fica de lição de casa implementar as estratégias em GLSL.

# Projeto 2.3

Vocês deverão implementar uma nova geometria para o Ray Tracer atual. Cada um terá a sua geometria.



Torus



Cilindro



Cone



Anel



Gota



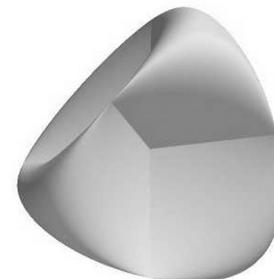
Pirâmide



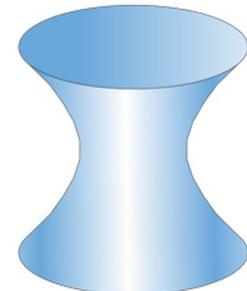
Taça



Parabolóide



Steiner



Hiperbolóide

# Computação Gráfica

Luciano Soares  
[<lpsoares@insper.edu.br>](mailto:lpsoares@insper.edu.br)