

# Computação Gráfica

WGSL & Raytracing 1

# WGSL - WebGPU Shading Language

- Linguagem moderna de shaders para web (maior acesso a hardware)
- Pode ser usada com JavaScript(JS), TypeScript(TS) e Rust
- Suporta Compute Shaders
- Fortemente tipada, ou seja, tipos de dados devem ser explicitamente definidos.
- A sintaxe e o design do WGSL foram feitos para ser mais seguros e menos propensos a erros comparado a outras linguagens de shaders: Rust based
- Links uteis:
  - <https://webgpufundamentals.org/webgpu/lessons/webgpu-wgsl.html>
  - <https://www.w3.org/TR/WGSL/>
  - <https://google.github.io/tour-of-wgsl/>

# WGSL vs GLSL

| GLSL  | WGSL   |
|---|--|
| OpenGL/WebGL  | WebGPU   |
| Tipada, mas com certa flexibilidade.  | Linguagem com <b>tipagem forte</b> .   |
| Tende a ser mais direta, com sintaxes menos rigorosas, mas também menos estruturadas. | Mais modular e estrita quanto à declaração de variáveis e funções.   |
| Amplamente usado em gráficos desktop, dispositivos móveis e web através do WebGL.     | Projetado especificamente para ser usado em navegadores modernos e com suporte a multiplataforma.                            |
| Mais antigo e largamente usado.   | Leva em consideração boas práticas de design moderno.  |
| Usado com WebGL, que pode ser mais lento ou ter menos controle de hardware moderno.   | Aproveita as vantagens do WebGPU, oferecendo melhor desempenho e controle de hardware, especialmente em computação paralela. |

# WGSL vs GLSL - Exemplos

## GLSL

```
float foo(vec2 uv, float k)
{
    return k + length(uv);
}
```

```
float foo(vec2 uv, float k)
{
    float o = length(uv) + k;
    return o;
}
```

## WGSL

```
fn foo(uv : vec2f, k : f32) -> f32
{
    return k + length(uv);
}
```

```
fn foo(uv : vec2f, k : f32) -> f32
{
    var o = length(uv) + k;
    return o;
}
```

## Ponteiros (Pointers):

### GLSL não suporta

## WGSL

```
fn foo(uv: ptr<function, vec2f>) -> f32
{
    return length(uv);
}

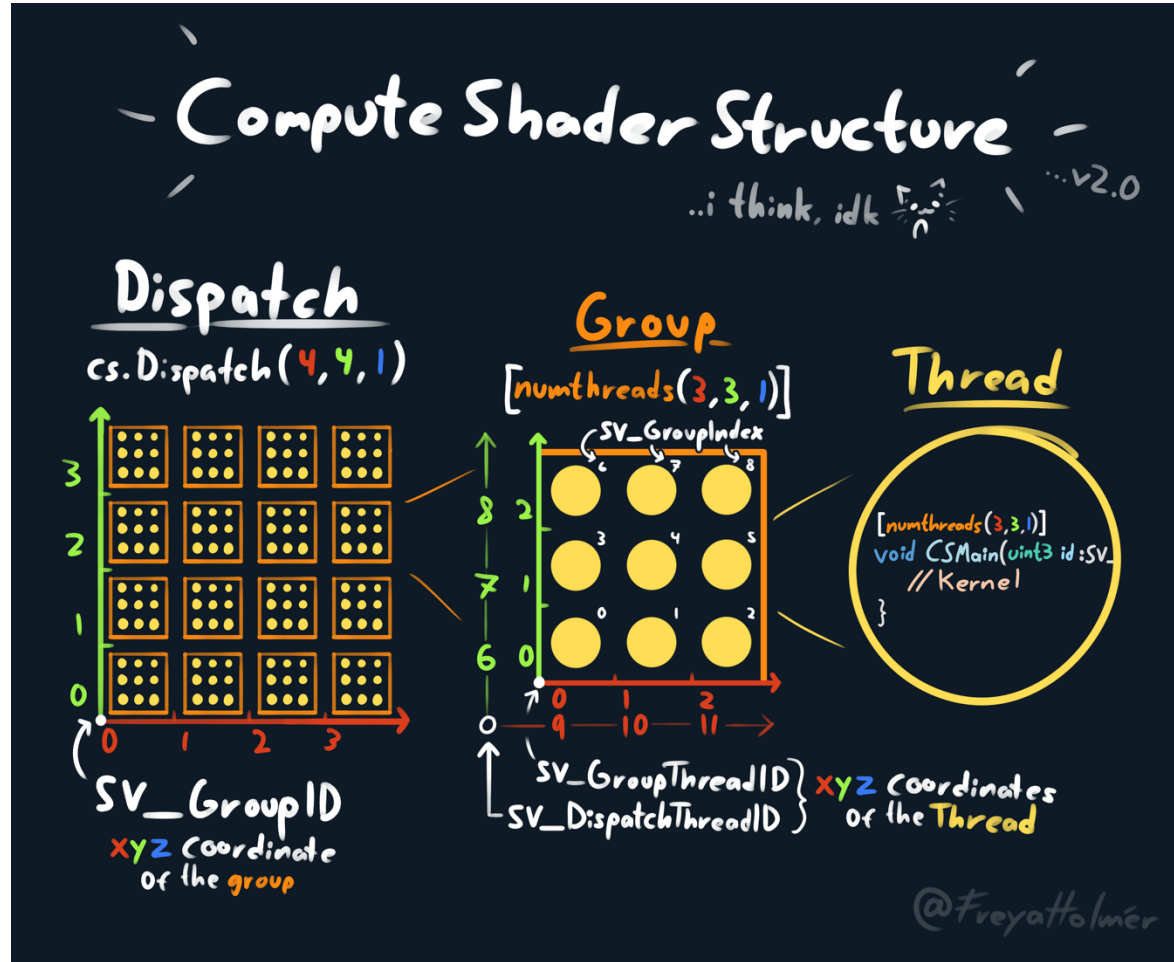
foo(&uv);
```

# Compute Shader

- **Propósito Geral:** Realiza operações computacionais diretamente na GPU, como física, simulação, processamento de dados, etc., sem precisar gerar gráficos.
- **Independente de Pipeline Gráfico:** Desvinculado ao pipeline gráfico tradicional (vertex/fragment shaders), sendo independentemente da renderização.
- **Grupos de Trabalho (Workgroups):** Divide a tarefa em grupos de trabalho para distribuir a carga computacional de forma eficiente entre os núcleos da GPU.
- **Entrada/Saída de Dados:** Recebe e escreve dados diretamente na memória da GPU (buffers, imagens), podendo manipular grandes volumes de dados.
- **Aplicações Diversas:** Usado para tarefas como física de partículas, inteligência artificial, simulação de fluidos, processamento de imagem, machine learning, etc.
- **Exemplo:** <https://gubebra.itch.io/multiple-species-physarum>
- Simulação de agentes (partículas, objetos)

**Resumindo:** Mais customizável, maleável, robusto e geral.

# Compute Shader



# Compute Shader em WGSL

## WGSL

```
const THREAD_COUNT = 16;
@compute @workgroup_size(THREAD_COUNT, THREAD_COUNT, 1)
fn render(@builtin(global_invocation_id) id : vec3u)
{
    ...
}
```

## Js

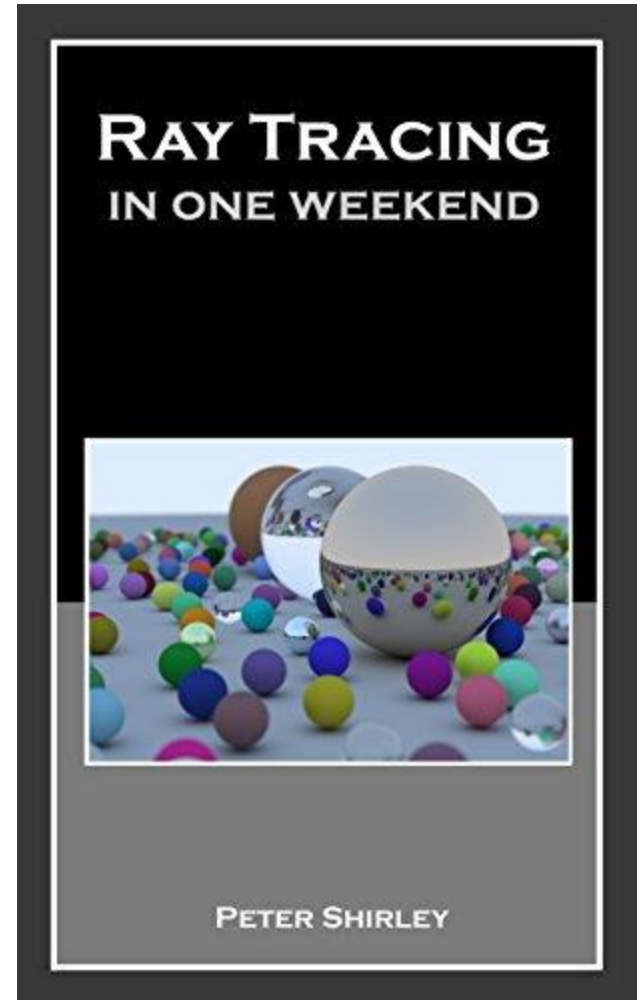
```
pass.dispatchWorkgroups(rez.x / THREAD_COUNT, rez.y / THREAD_COUNT, 1);
```

# Ray Tracing

Esta parte do curso foi baseada no livro:

"Ray Tracing in One Weekend", por Peter Shirley

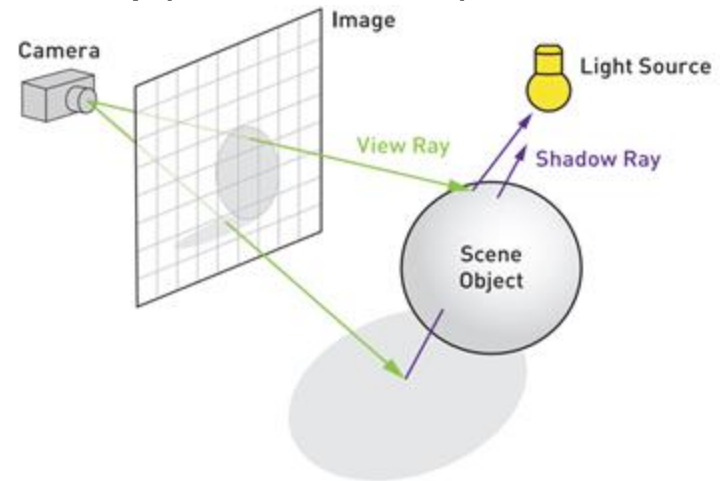
**URL (series):** <https://raytracing.github.io/>



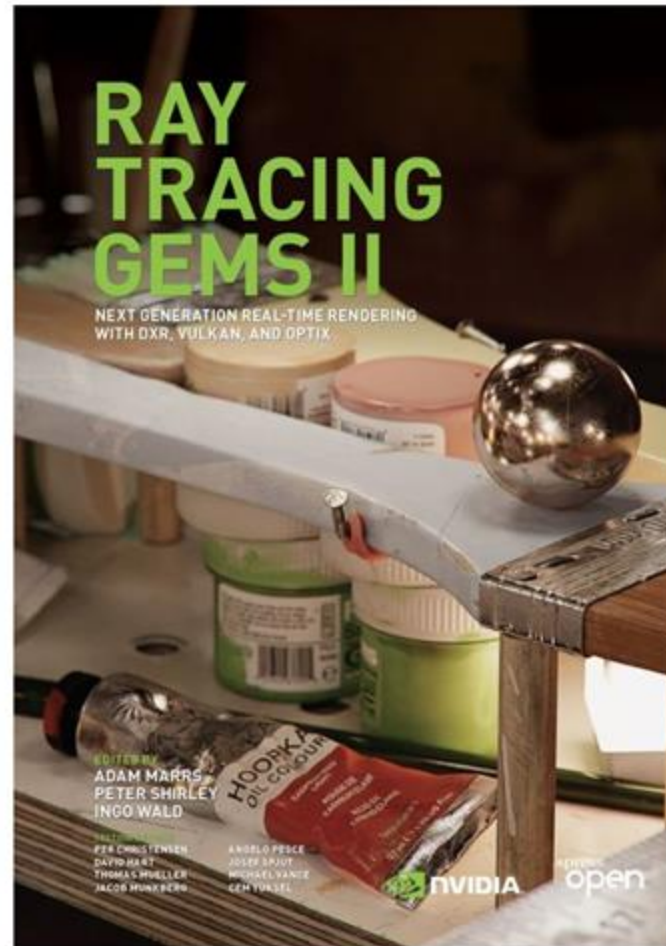


# Ray Tracing – o que é

- **Ray Tracing** é uma técnica de renderização usada para criar imagens altamente realistas, simulando o comportamento reverso dos raios de luz.
- A técnica funciona ao **rastrear fontes de luz** partindo de uma câmera e verificando onde esses raios atingem objetos no ambiente.
- Cada interação de um raio com uma superfície tem cálculos de **reflexão, refração, sombreamento e sombras** com base nas propriedades do material e da iluminação da cena.
- Matemática para definir objetos (ex: geometrias) pode ser complexa.



# Alguns Livros Interessantes



# Onde Ray Tracing é usado

Ray Tracing é computacionalmente “muito pesado”. Contudo, com os avanços em recursos computacionais, essa técnica está se tornando viável até para aplicações em tempo real.

- Filmes/Animações
- Softwares de modelagem 3D: Blender / Maya
- Jogos: Reflexos / Sombras / AO / Lightmaps

## 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

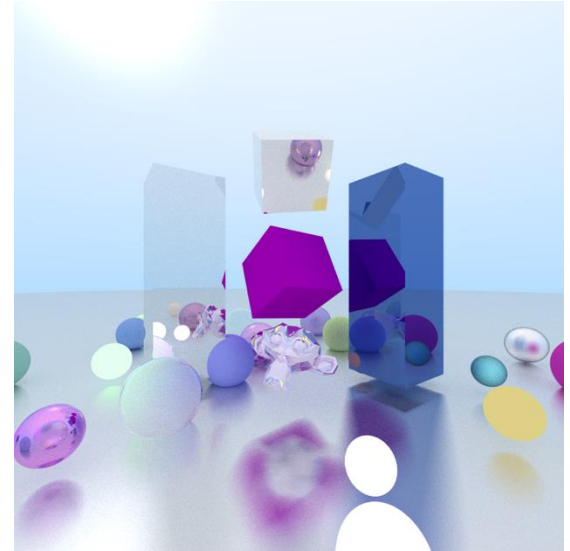
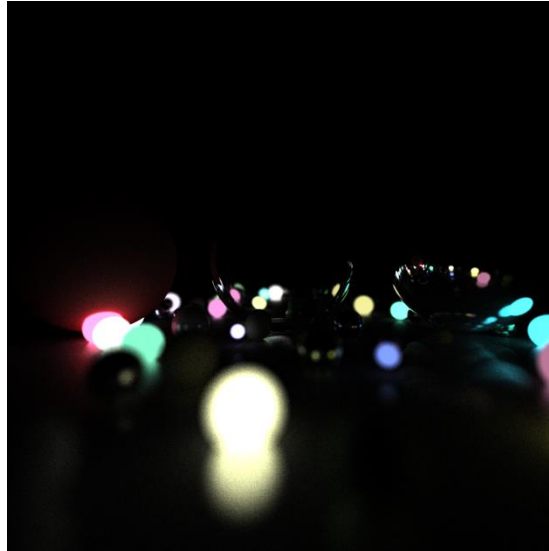
# Path Tracer

Um Path Tracer é um método de renderização baseada em Monte Carlo para produzir imagens de cenas tridimensionais. O livro de referência usa na prática Path Tracer.



# Projeto

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

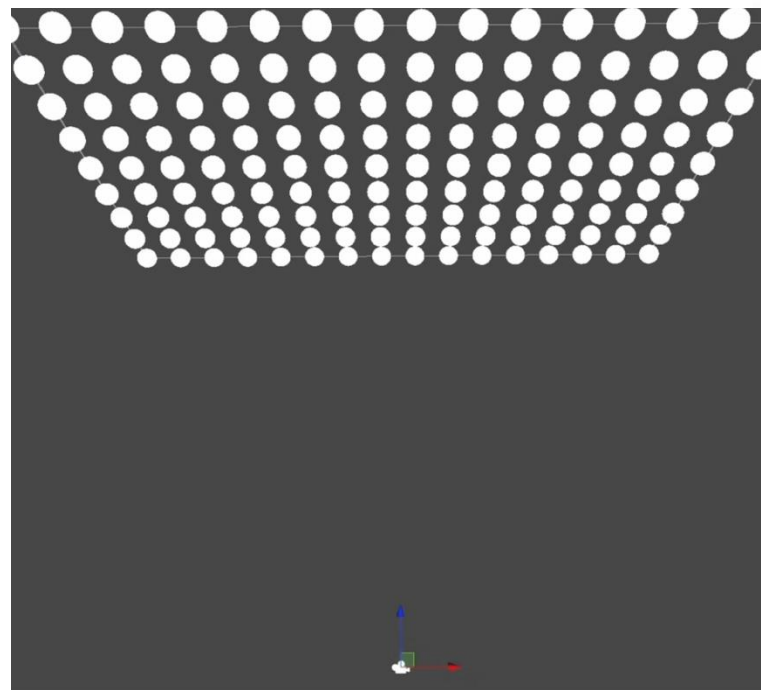




# Path Tracer

Ideia básica do algoritmo:

- Crie  $x$  raios que saem da câmera virtual da cena
- Defina um número  $y$  máximo de reflexões (bounces)
- Para cada raio e a cada bounce, veja onde ele bateu (objetos na cena)
- Se bater em algo, pegue a cor e material do objeto
- Calcule a nova direção do raio após a colisão baseado no tipo de material, e multiplique a cor do pixel pela cor do material
- Se não bater em nada, retorne com a cor do ambiente
- Faça a média de cor com base no número  $x$  (**raios por pixel**)
- Acumule as cores (progressive rendering) e faça a transformação de espaço de cores (linear para gamma)



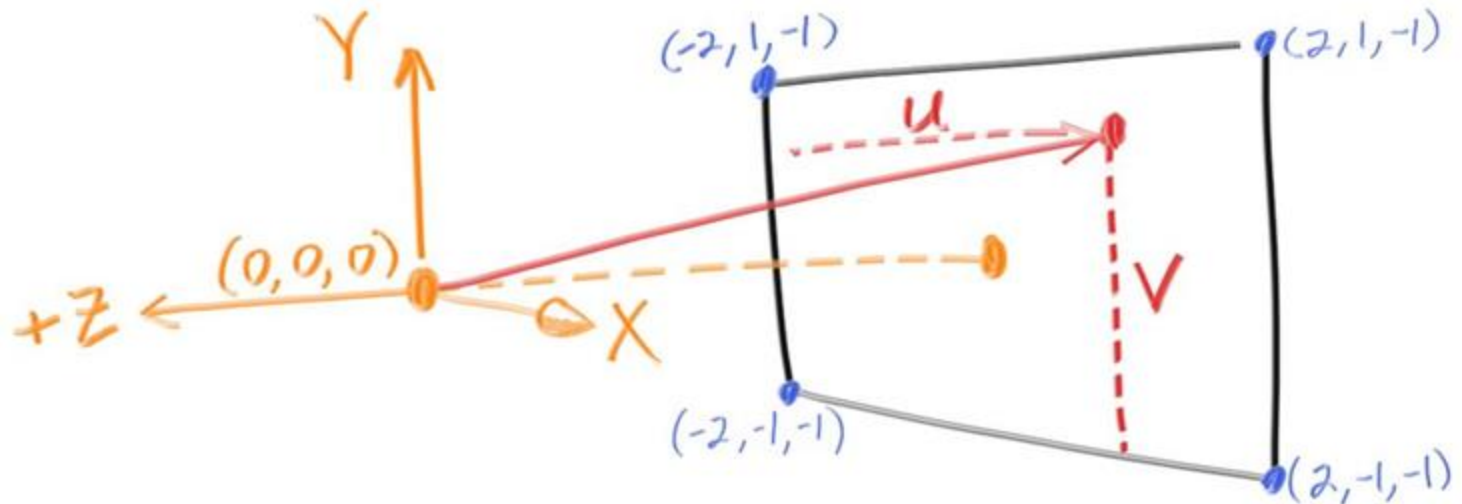
Fonte animação: [link](#)

# Criando câmera

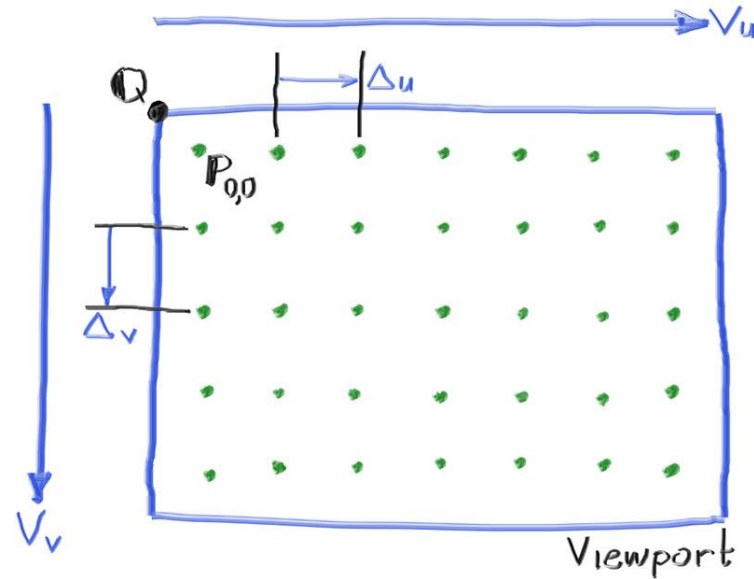
Vamos deixar o ponto de vista no  $(0,0,0)$  olhando para o Z negativo.

Varredura de pixels:

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



# Criando o raio da câmera



```
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);
```



# Raio (half-line / semirreta)

Raios do RayTracer

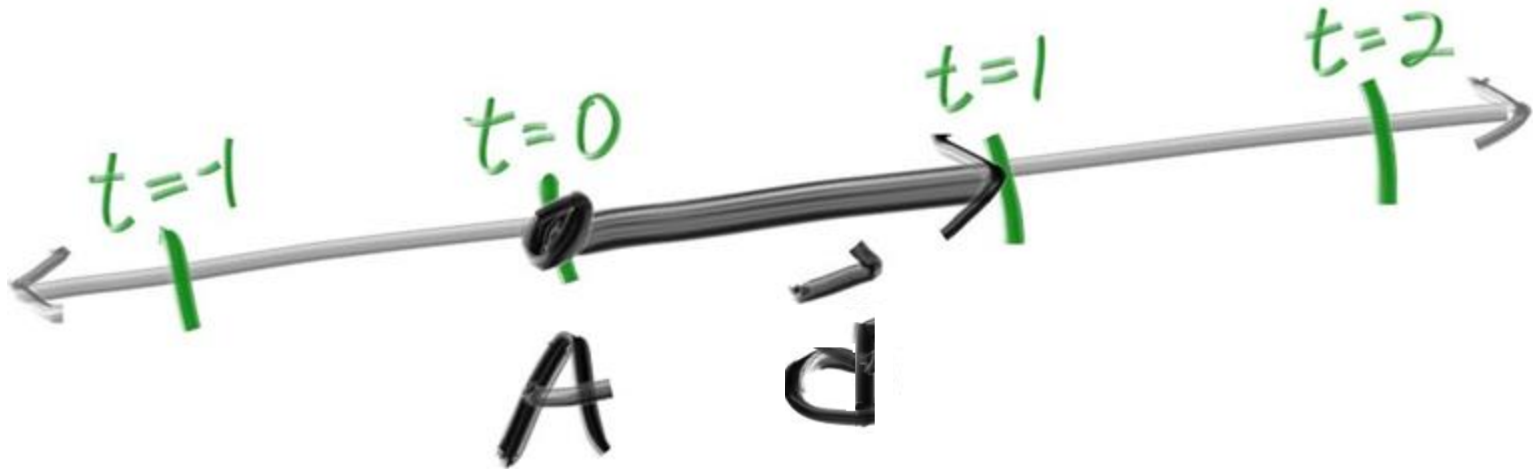
$$P(t) = A + td$$

A é a origem do raio

d é um vetor que define a direção do raio

t é um parâmetro real

P(t) é o ponto sobre o raio



# Adicionando uma Esfera

Esferas são objetos comumente vistos em RayTracers devido à simplicidade de calcular a intersecção de um raio com uma esfera. Equação de uma superfície esférica centrada na origem:

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

Assim, para um ponto  $(x, y, z)$  pertencente à superfície esférica, a equação acima precisa ser verdadeira.

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

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

# Adicionando uma Esfera

Para uma superfície esférica 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 superfície esférica 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

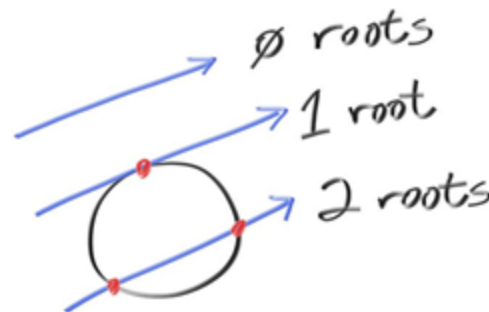
Precisamos encontrar as raízes da equação de segundo grau.

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

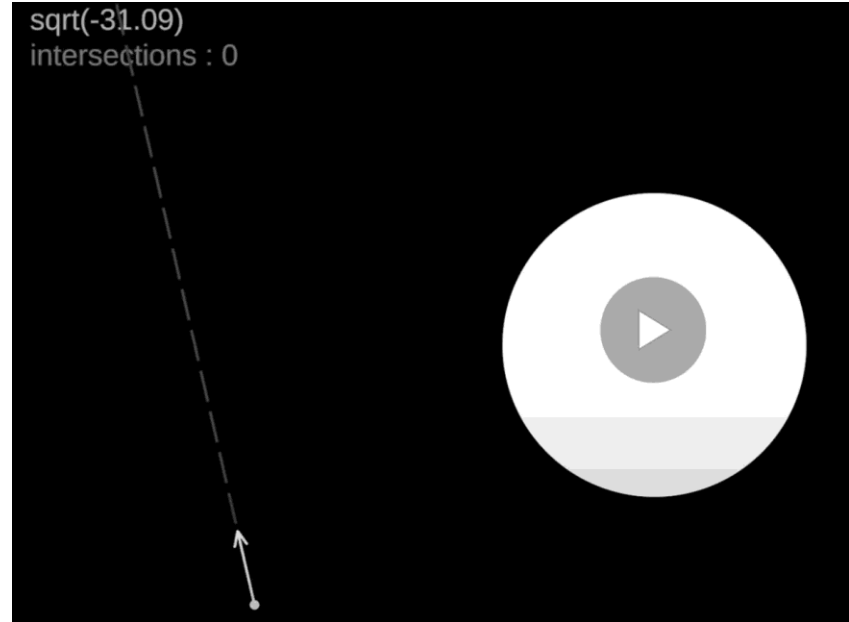
$$\underbrace{t^2 \mathbf{b} \cdot \mathbf{b}}_a + \underbrace{2t \mathbf{b}(\mathbf{A} - \mathbf{C})}_b + \underbrace{(\mathbf{A} - \mathbf{C}) \cdot (\mathbf{A} - \mathbf{C}) - r^2}_c = 0$$

Se o delta:

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



# Desenhando a esfera

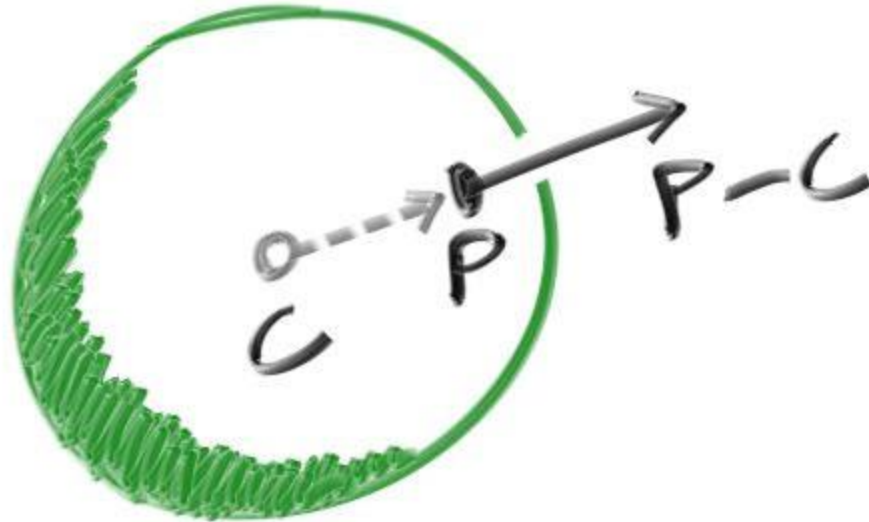


Fonte animação: [link](#)

# Normais na Superfície

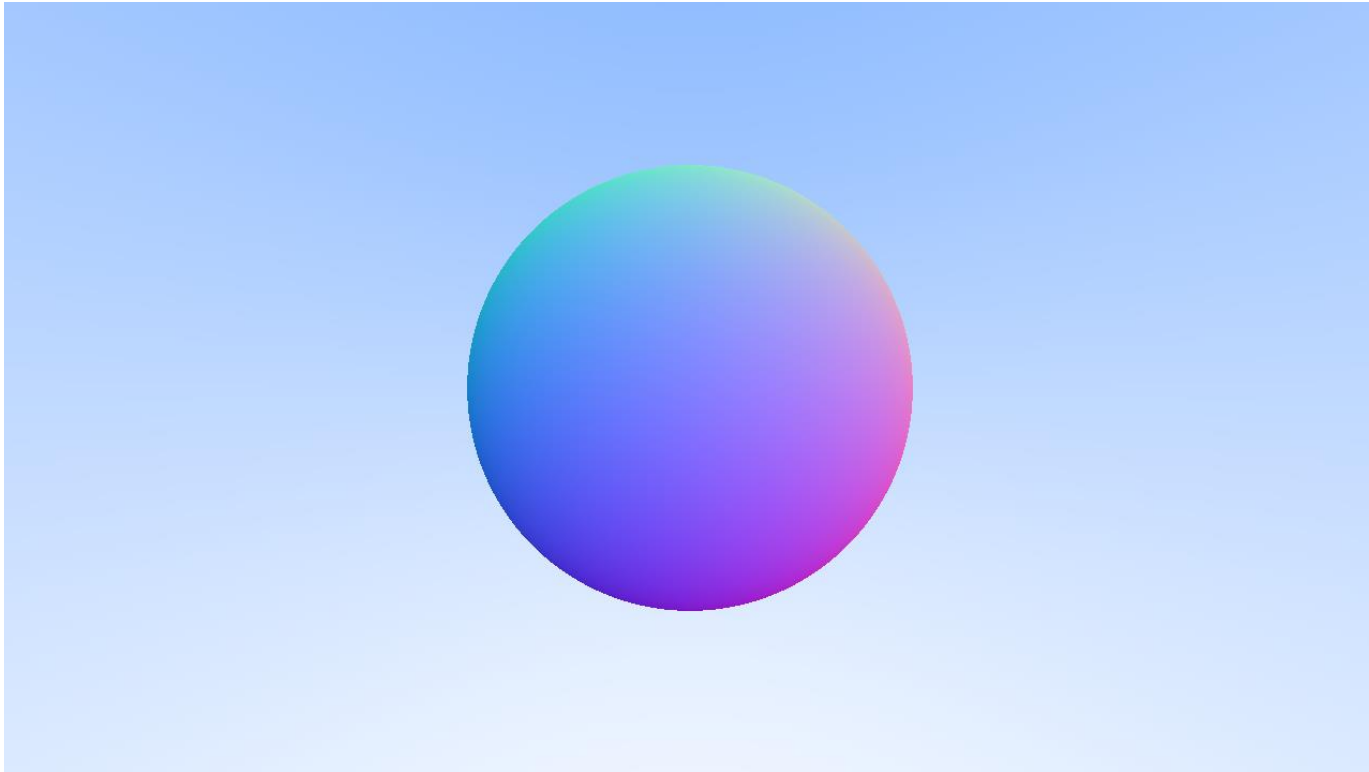
Para descobrir o vetor normal em um ponto na superfície de uma esfera, basta subtrair esse ponto do centro da esfera:

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



# Esfera exibindo valores das normais

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





# Simplificando o código de intersecção de raios

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

- Primeiro: o produto escalar de um vetor por ele mesmo é igual ao quadrado do módulo daquele vetor.

$$(A - C) \cdot (A - C) = \|A - C\|^2$$

# Simplificando o código de intersecção de raios

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

- Segundo: que o valor do coeficiente linear da equação do segundo grau (b) é uma multiplicação por 2, assim podemos colocar em evidência esse fator 2 e simplificar a equação.

$$b = 2\mathbf{b}(\mathbf{A}-\mathbf{C})$$

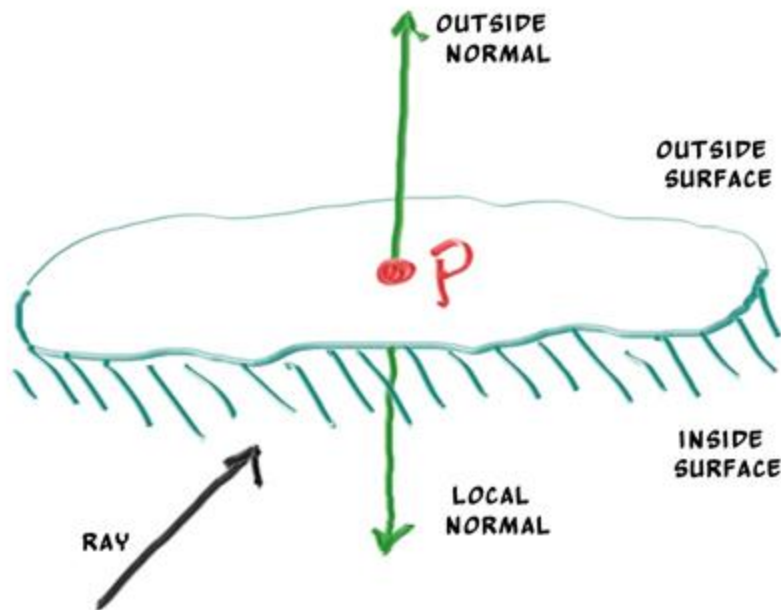
$$h = \mathbf{b}(\mathbf{A}-\mathbf{C})$$

$$b=2h$$

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

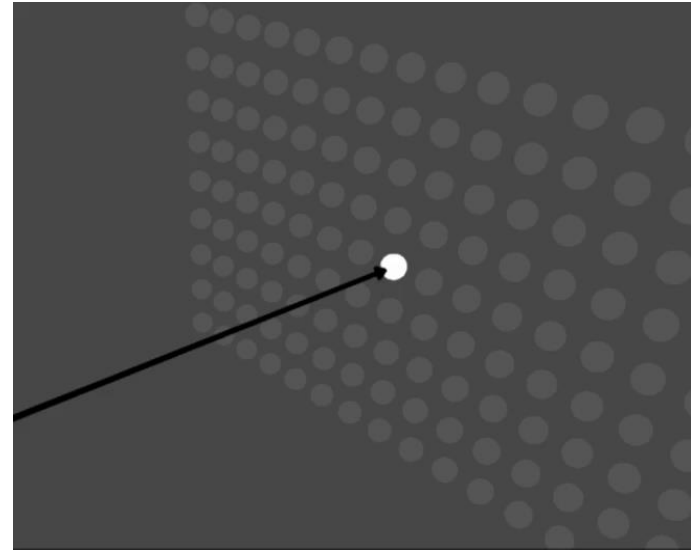
# Face frontal e traseira

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.



# Antialiasing & Luz

Artefatos podem surgir nas bordas dos objetos na imagem. Vamos fazer um antialiasing para reduzir esse problema.



Fonte animação: [link](#)

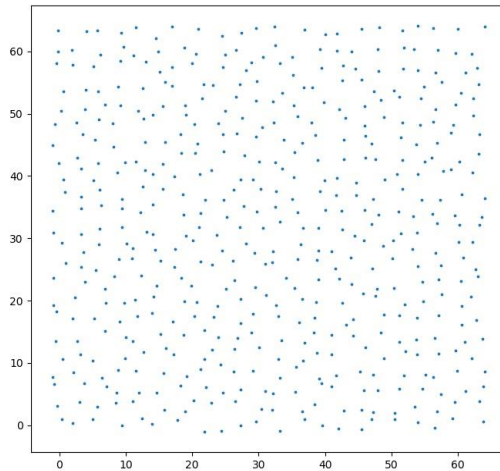
Vamos lançar vários raios por pixel?

# Gerador de Números Aleatórios

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

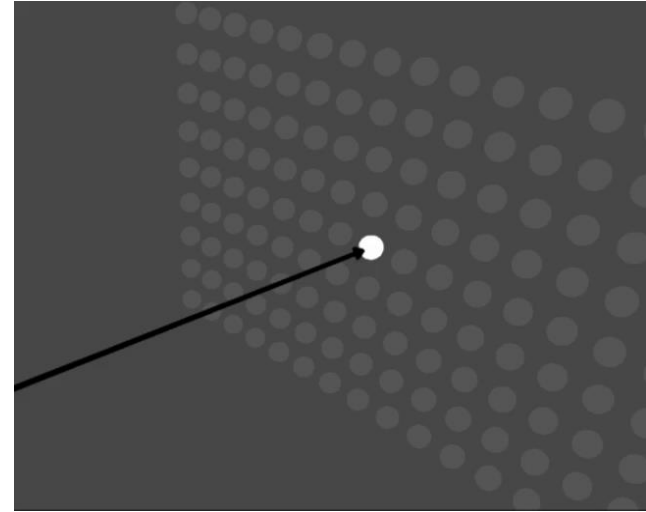
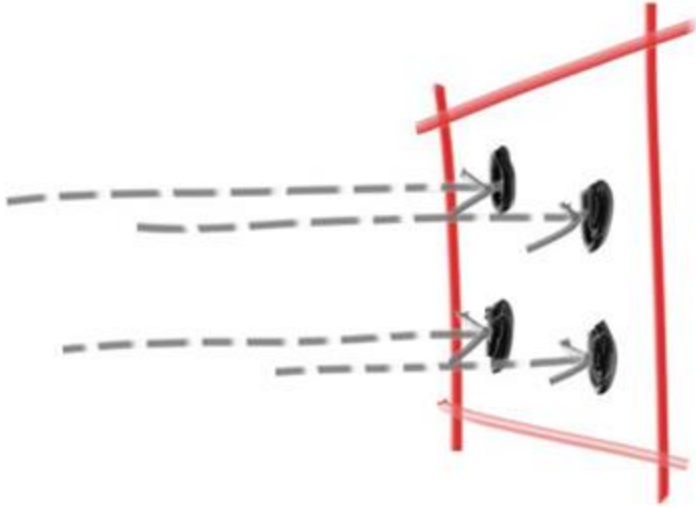
Além disso, precisamos de um gerador de números que gere um ponto aleatório dentro de um retângulo 2D.

O WGSL (nenhuma linguagem de GPU) não possui uma função de números aleatórios de forma nativa. Assim temos de fazer uma.



# Várias amostras por pixel

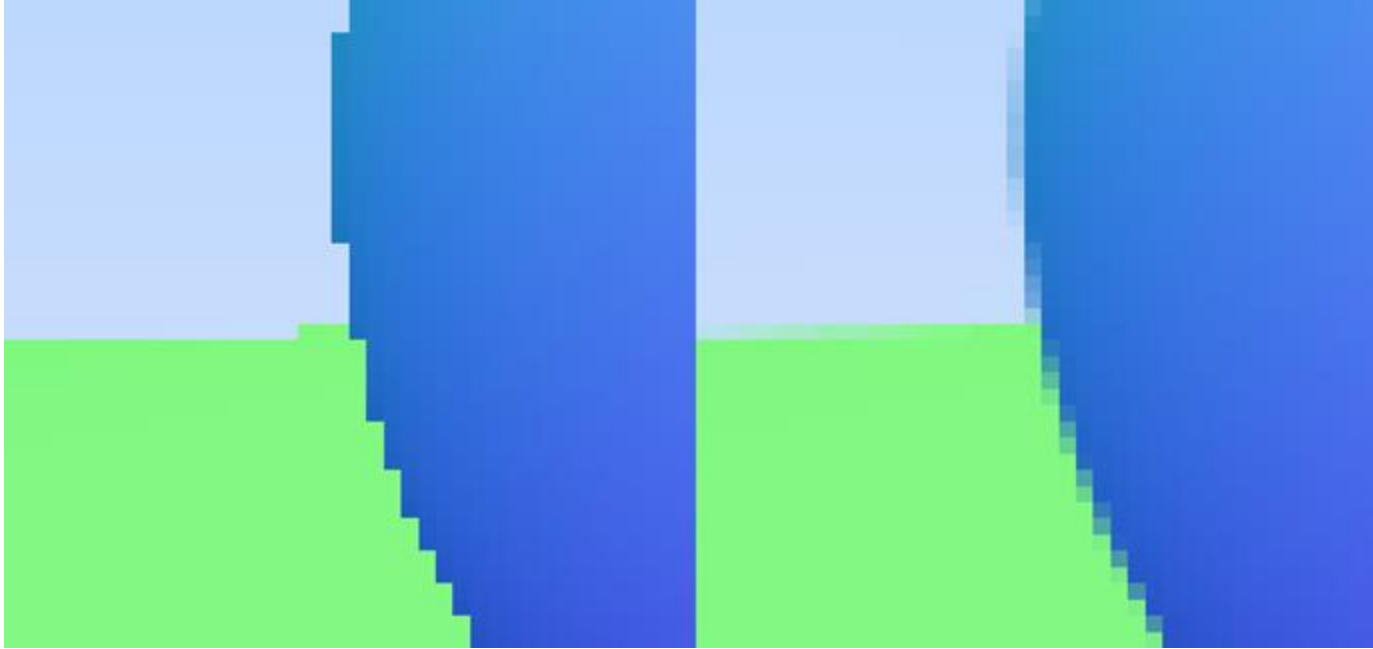
Vamos lançar vários raios por pixel e depois fazer uma média.



Fonte animação: [link](#)

```
fn sample_square(rngState: ptr<function, u32>) -> vec2f
{
    return vec2f(rng_next_float(rngState), rng_next_float(rngState));
}
var uv = (fragCoord + sample_square(&rng_state)) / vec2(rez);
```

# Antes e depois do Antialiasing

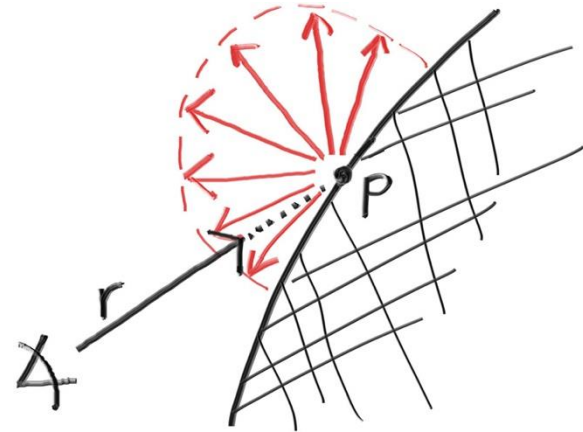
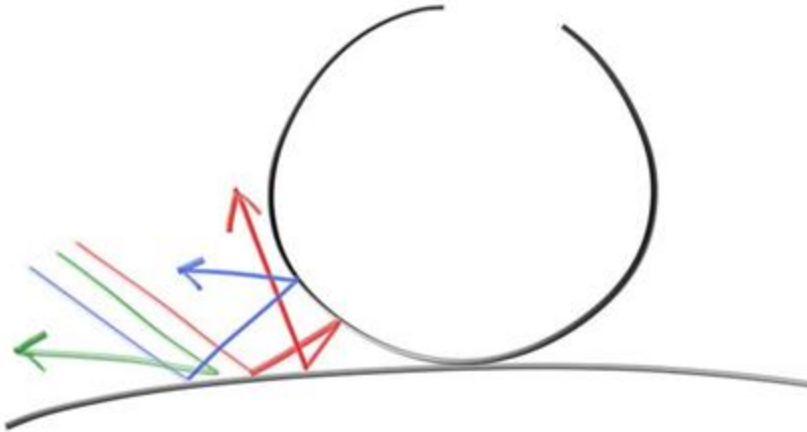


Somente anti-aliasing?

Gabarito do projeto: <https://gubebra.itch.io/raytracing>

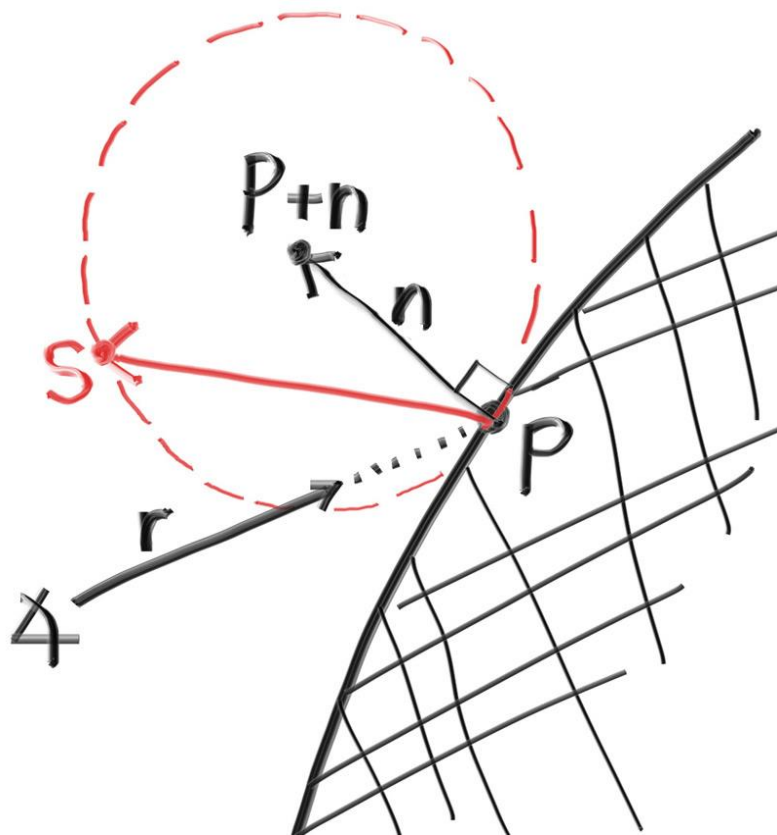
# Materiais difusos

Os raios de luz que refletem em uma superfície difusa tem uma distribuição de sua direção definida com uma certa aleatoriedade.





# Reflexão Lambertiana



*Lambertian distribution:*

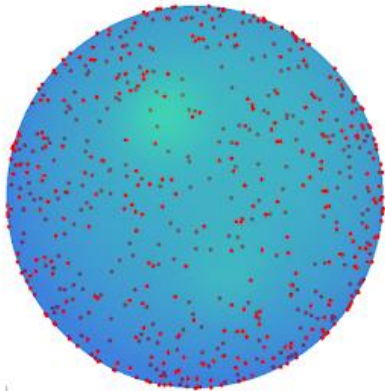
- A distribuição dispersa os raios refletidos de maneira proporcional a  $\cos(\phi)$
- $\phi$  é o ângulo entre o raio refletido e a normal da superfície
- Isso significa que um raio refletido tem maior probabilidade de dispersar em uma direção próxima à normal da superfície
- Há menor probabilidade de dispersar em direções afastadas da normal

`Normalize(Normal + RandomInSphere())`

# Reflexão Lambertiana

## Função random: Ponto em uma esfera

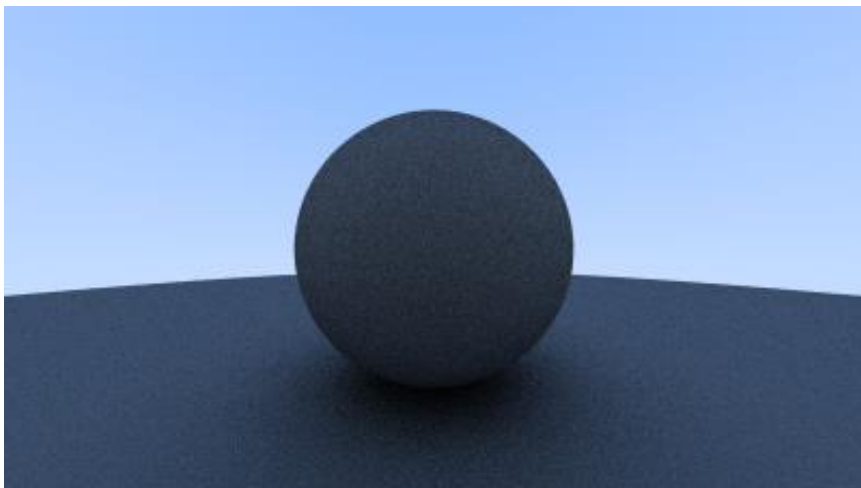
```
fn rng_next_vec3_in_unit_sphere(state: ptr<function, u32>) -> vec3<f32>
{
    var z = 2.0 * rng_next_float(state) - 1.0;
    var a = 2.0 * PI * rng_next_float(state);
    var r = sqrt(1.0 - z * z);
    var x = r * cos(a);
    var y = r * sin(a);
    return vec3(x, y, z);
}
```



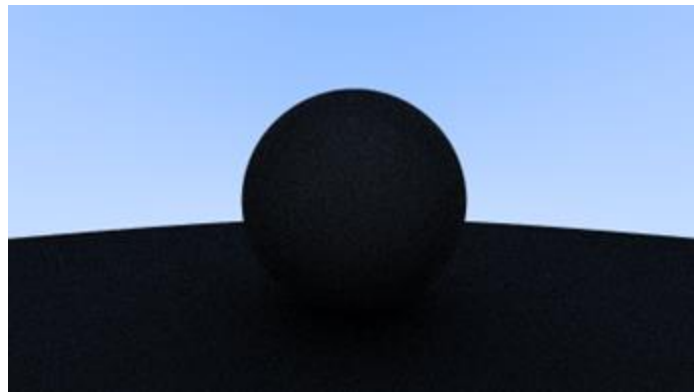
# Resultado

Parece que tem algo estranho.

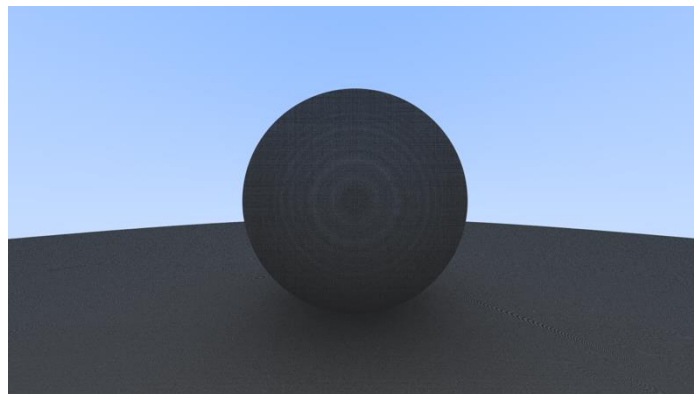
Deveria Acontecer



Mas Acontece



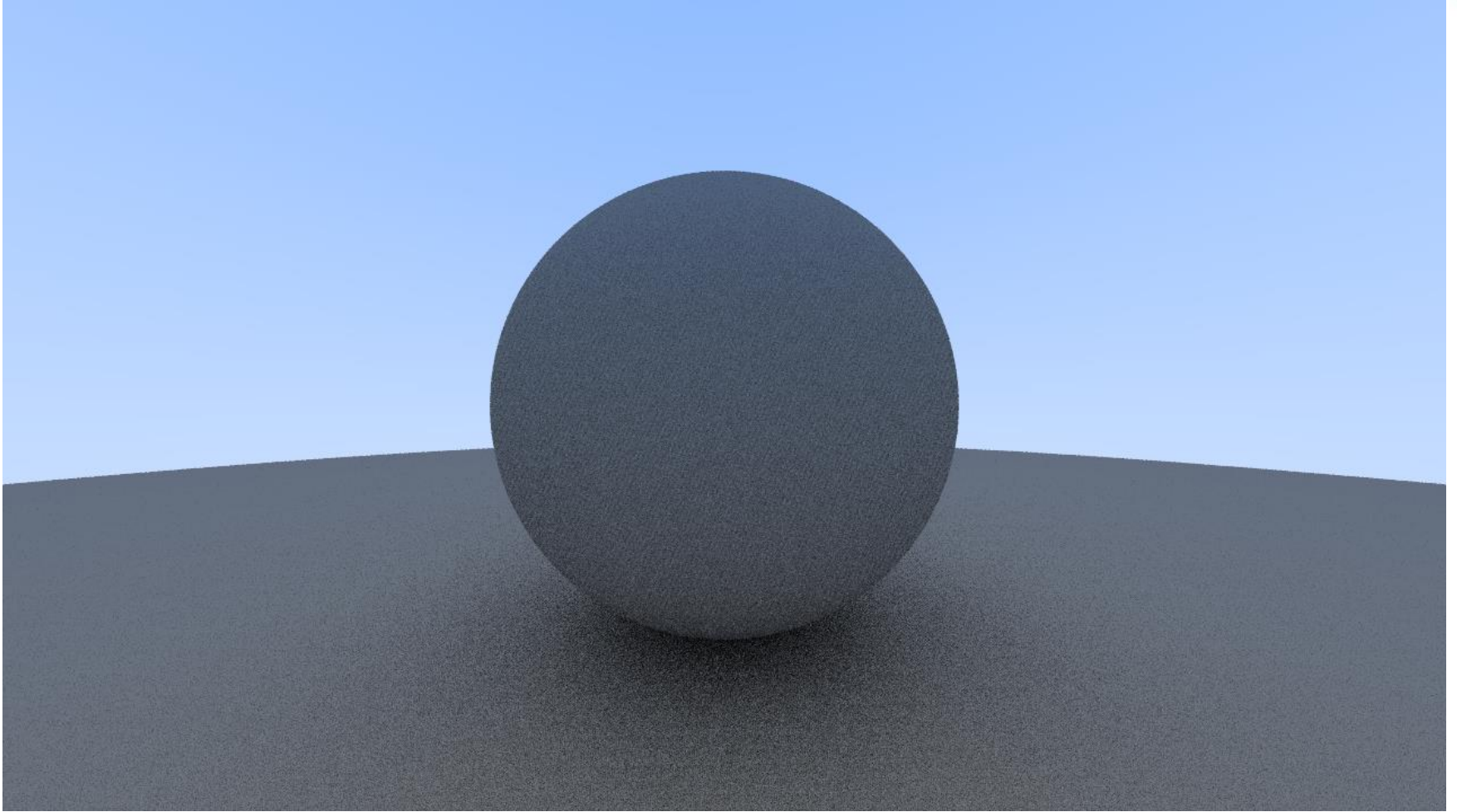
ou



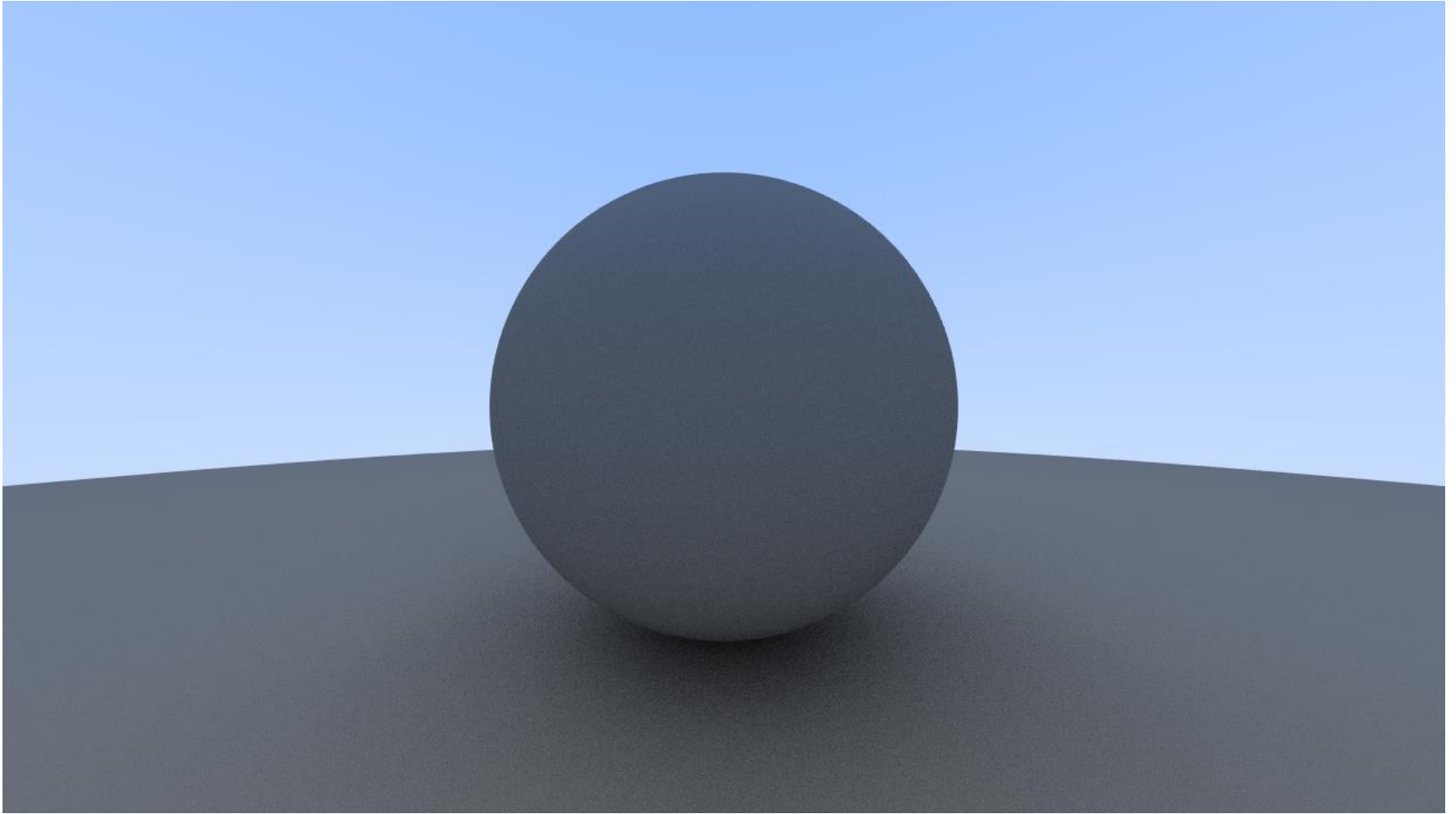
**Shadow Acne - Os raios intersectam diretamente com a própria geometria.**

**ou seja, um  $t=0.0001$  de inicio de raio resolve!**

# Exemplo (10 amostras)



# Exemplo (100 amostras)



# Projeto Raytracer

Rubrica/Dicas/Código

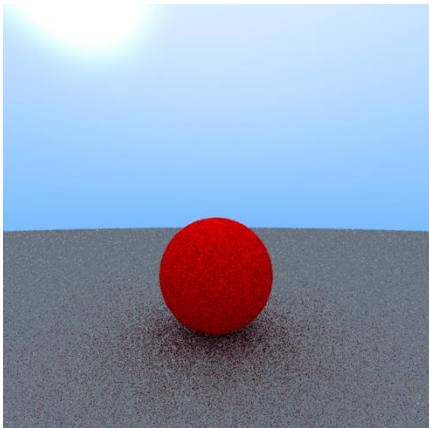
<https://github.com/Gustavobb/raytracing-wgsl-template> - Fazer um fork

# Projeto Raytracer

Se você fizer:

```
para cada raio (samples_per_pixel):  
    crie um raio  
    trace do raio:  
  
    para cada bounce:  
        para cada esfera (nesse caso):  
            veja se a colisão foi encontrada (colisão mais perto) e pegue cor/material do objeto  
  
        se colisão foi encontrada:  
            calcule a nova direção do acordo com o material  
            calcule a cor do raio de acordo com o material  
            calcule a nova origem do raio de acordo com a colisão  
  
        se colisão não foi encontrada:  
            retorne a cor de fundo e cor do objeto  
  
faça a média das cores dos raios
```

Com a aula de hoje, você chega (cena "Basic"):





# Computação Gráfica

Luciano Soares  
<lpsoares@insper.edu.br>

Fabio Orfali  
<fabioo1@insper.edu.br>

Gustavo Braga  
<gustavobb1@insper.edu.br>