Computação Gráfica

Ray Marching 1

Ray Marching: Introdução

- Semelhança com **Ray Tracing**: raios são lançados de uma câmera virtual em objetos que não são definidos por uma malha poligonal, são calculados matematicamente

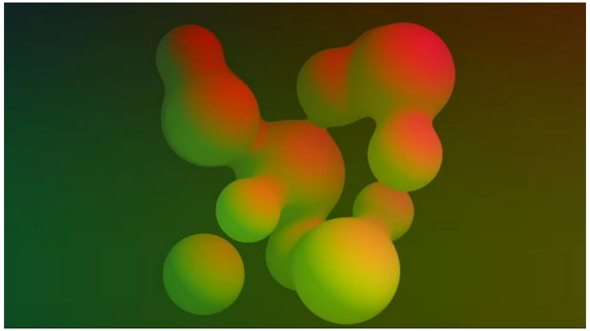
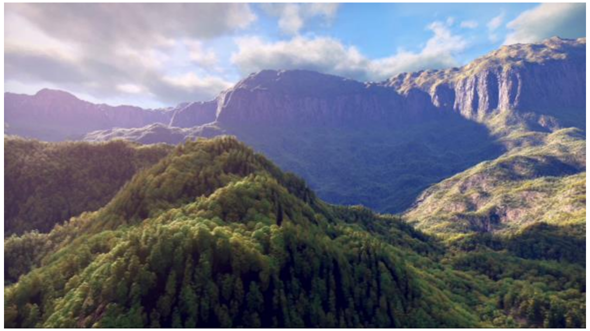
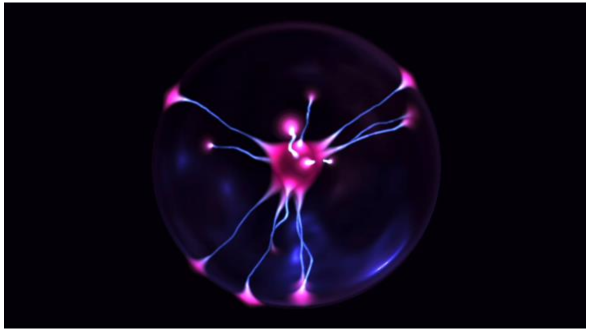
- Diferença com **Ray Tracing**: não calcula interseções diretas

- **O objetivo é verificar se um raio que lançarmos intersecta* um objeto e o ponto de interseção*.**

- Permite criar cenas com objetos difíceis de modelar, como fractais, formas suaves e volumes (nuvem por exemplo)

Ray Marching: Exemplos

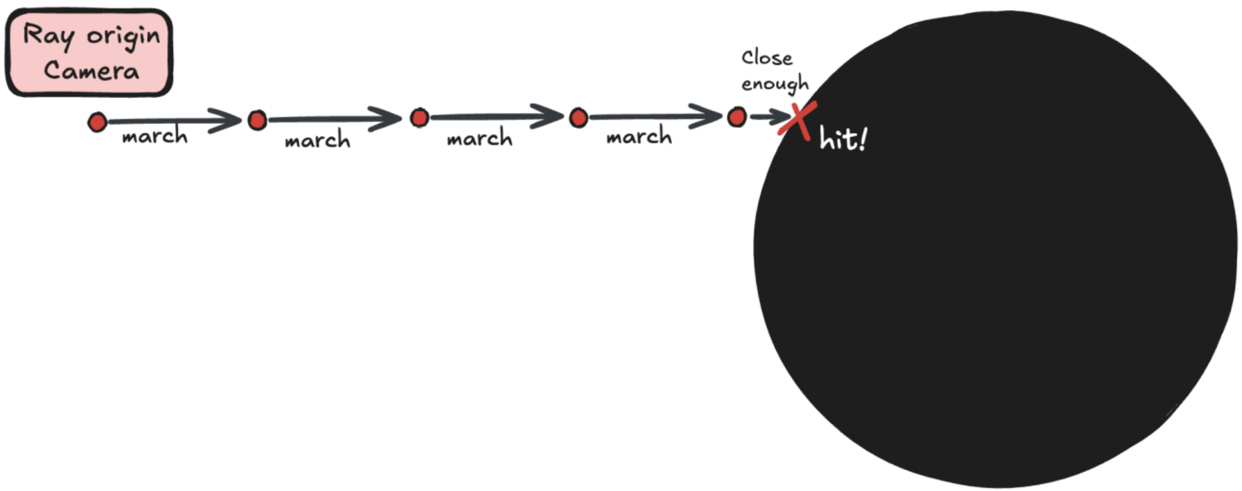


Fonte: Shadertoy

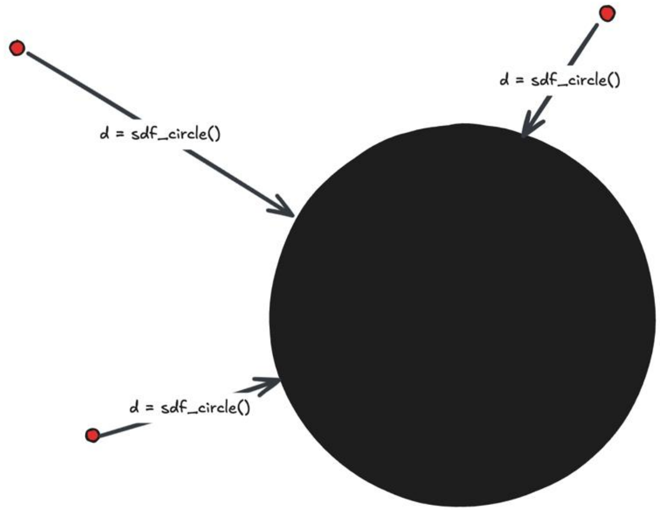
Ray Marching: Introdução

- Vamos lançar raios que saem da tela
- Vamos "marchar" (andar na direção do raio)
- Se houver algum objeto perto o bastante do raio (valor pequeno), consideramos que batemos nele



Como saber a distância até o objeto?

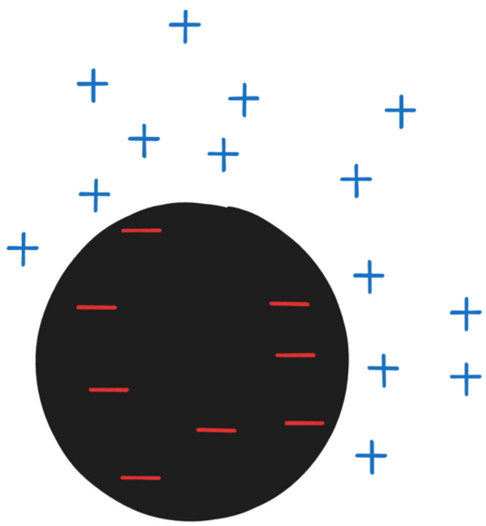
- **Signed Distance Function (SDF)**
- A menor distância entre um ponto (raio) e o objeto (geometria) é chamada de **SDF.**
- Cada geometria pode ter sua **SDF (função de distância).**
- Se usarmos a **SDF** do objeto, saberemos quão longe ele está da posição de um ponto do raio nessa "marcha"

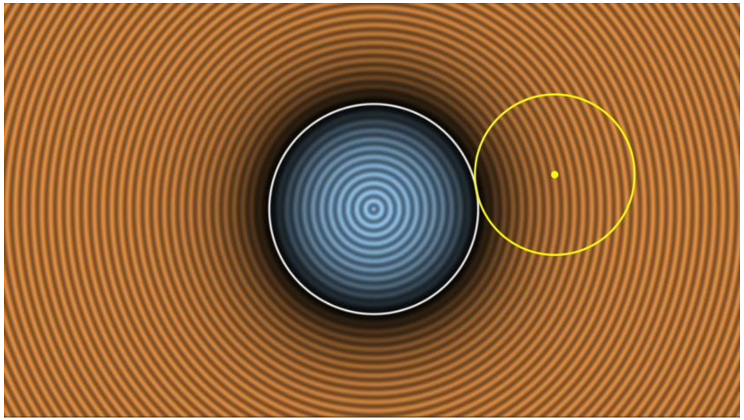


Fonte: wikipedia

SDFs

- **SDF (Signed Distance Function)**: função que recebe coordenadas de um ponto no espaço
- Retorna a menor distância entre o ponto e uma superfície
- O sinal do valor retornado indica a posição do ponto:
  - **Positivo:** ponto está fora da superfície
  - **Negativo:** ponto está dentro da superfície
  - **Zero:** ponto está tangenciando a superfície

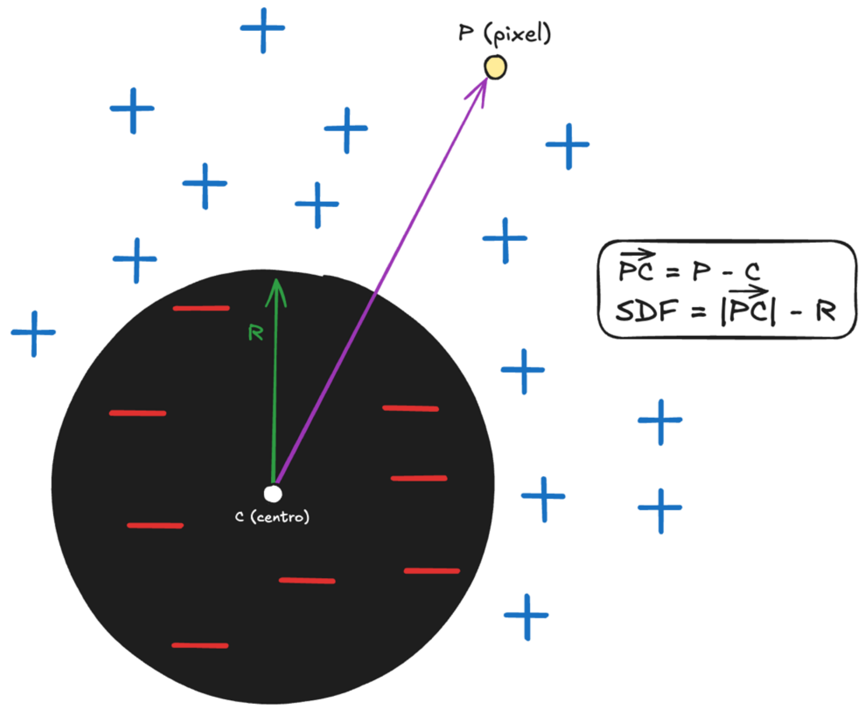




Fonte: wikipedia

SDFs – Exemplo círculo

6.6	5.9	5.3	4.7	4.3	3.9	3.6	3.5	3.5	3.6	3.9	4.3	4.7	5.3	5.9	6.6
5.9	5.2	4.5	3.9	3.4	3.0	2.7	2.5	2.5	2.7	3.0	3.4	3.9	4.5	5.2	5.9
5.3	4.5	3.8	3.1	2.5	2.0	1.7	1.5	1.5	1.7	2.0	2.5	3.1	3.8	4.5	5.3
4.7	3.9	3.1	2.4	1.7	1.1	0.7	0.5	0.5	0.7	1.1	1.7	2.4	3.1	3.9	4.7
4.3	3.4	2.5	1.7	0.9	0.3	-0.2	-0.5	-0.5	-0.2	0.3	0.9	1.7	2.5	3.4	4.3
3.9	3.0	2.0	1.1	0.3	-0.5	-1.1	-1.5	-1.5	-1.1	-0.5	0.3	1.1	2.0	3.0	3.9
3.6	2.7	1.7	0.7	-0.2	-1.1	-1.9	-2.4	-2.4	-1.9	-1.1	-0.2	0.7	1.7	2.7	3.6
3.5	2.5	1.5	0.5	-0.5	-1.5	-2.4	-3.3	-3.3	-2.4	-1.5	-0.5	0.5	1.5	2.5	3.5
3.5	2.5	1.5	0.5	-0.5	-1.5	-2.4	-3.3	-3.3	-2.4	-1.5	-0.5	0.5	1.5	2.5	3.5
3.6	2.7	1.7	0.7	-0.2	-1.1	-1.9	-2.4	-2.4	-1.9	-1.1	-0.2	0.7	1.7	2.7	3.6
3.9	3.0	2.0	1.1	0.3	-0.5	-1.1	-1.5	-1.5	-1.1	-0.5	0.3	1.1	2.0	3.0	3.9
4.3	3.4	2.5	1.7	0.9	0.3	-0.2	-0.5	-0.5	-0.2	0.3	0.9	1.7	2.5	3.4	4.3
4.7	3.9	3.1	2.4	1.7	1.1	0.7	0.5	0.5	0.7	1.1	1.7	2.4	3.1	3.9	4.7
5.3	4.5	3.8	3.1	2.5	2.0	1.7	1.5	1.5	1.7	2.0	2.5	3.1	3.8	4.5	5.3
5.9	5.2	4.5	3.9	3.4	3.0	2.7	2.5	2.5	2.7	3.0	3.4	3.9	4.5	5.2	5.9
6.6	5.9	5.3	4.7	4.3	3.9	3.6	3.5	3.5	3.6	3.9	4.3	4.7	5.3	5.9	6.6



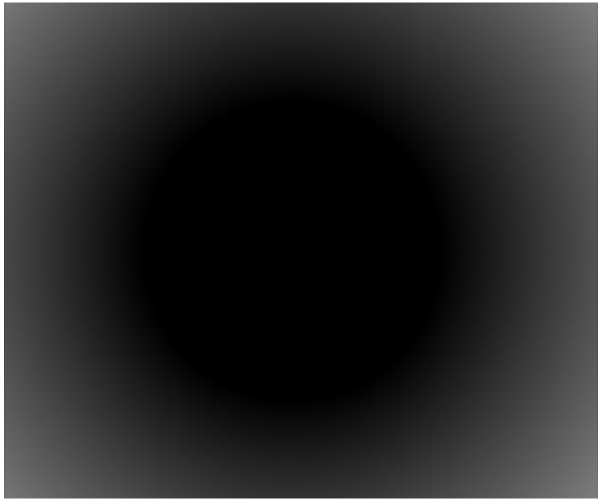
Insper

SDFs – Exemplo Círculo

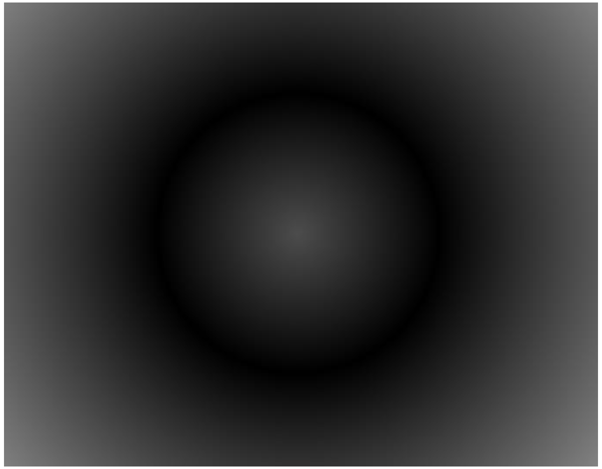
No shadertoy:

(Qualquer semelhança com a aula de shaders não é mera coincidência)

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 uv = (fragCoord - iResolution.xy / 2.0) / iResolution.y;
    float circ = length(uv) - 0.3;
    fragColor = vec4(circ);
}
```



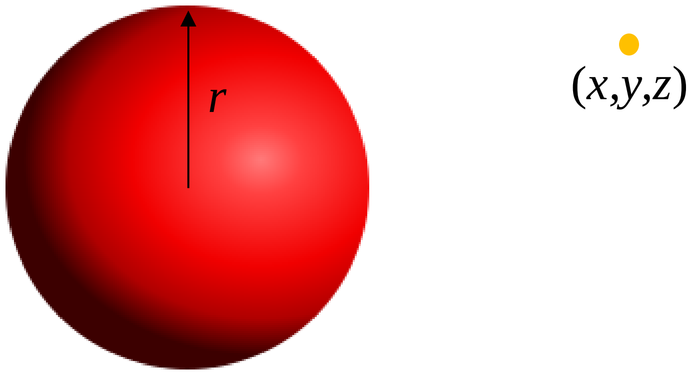
Com abs():



SDF (Signed Distance Function) em 3D

- Da mesma forma que as funções 2D, as funções 3D retornam a menor distância de um ponto no espaço a uma superfície.
- Mas agora temos uma coordenada a mais, assim por exemplo para uma esfera de raio ‘r’ a função ficaria:

$$f\_dist(x, y, z, r) = \sqrt{x^2 + y^2 + z^2} - r$$

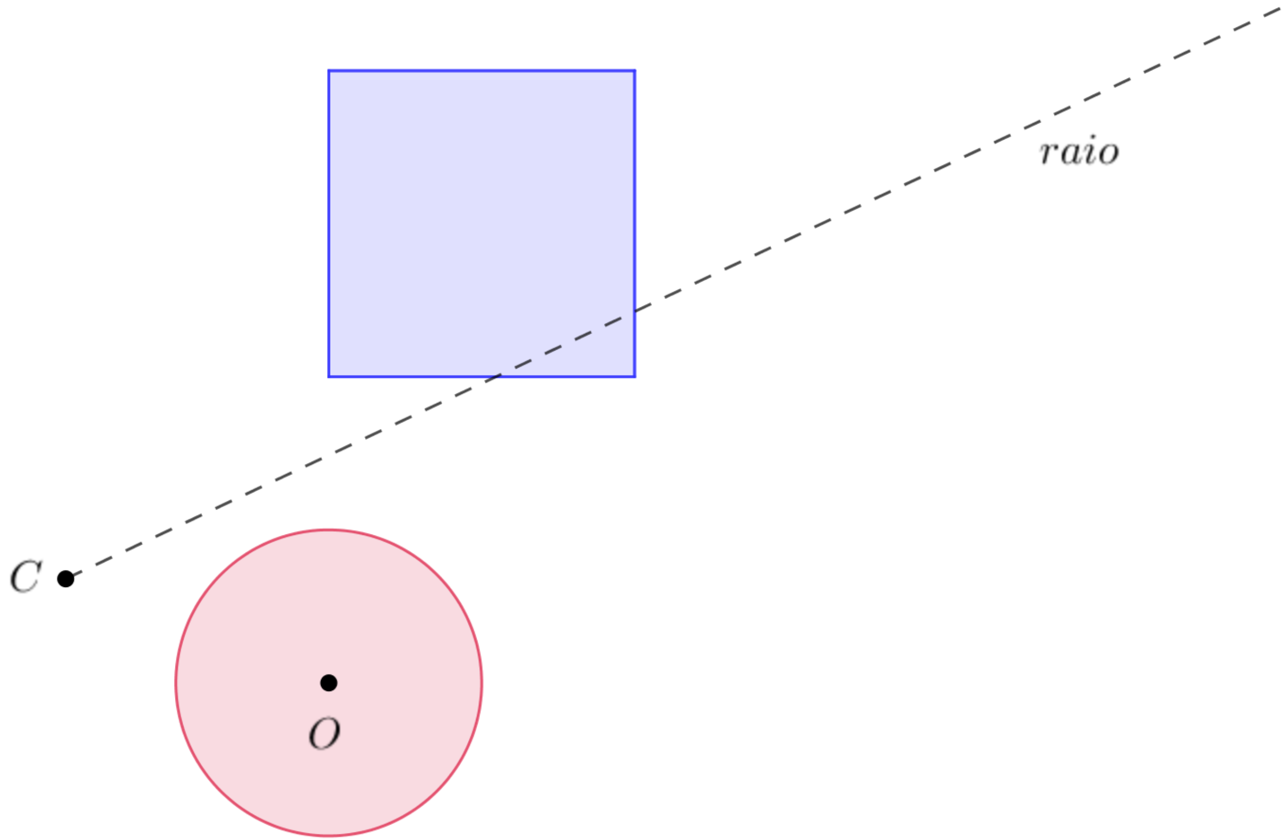


Normalmente desenhamos a esfera no zero da equação, ou seja, na curva de nível zero.

Ray Marching: Algoritmo

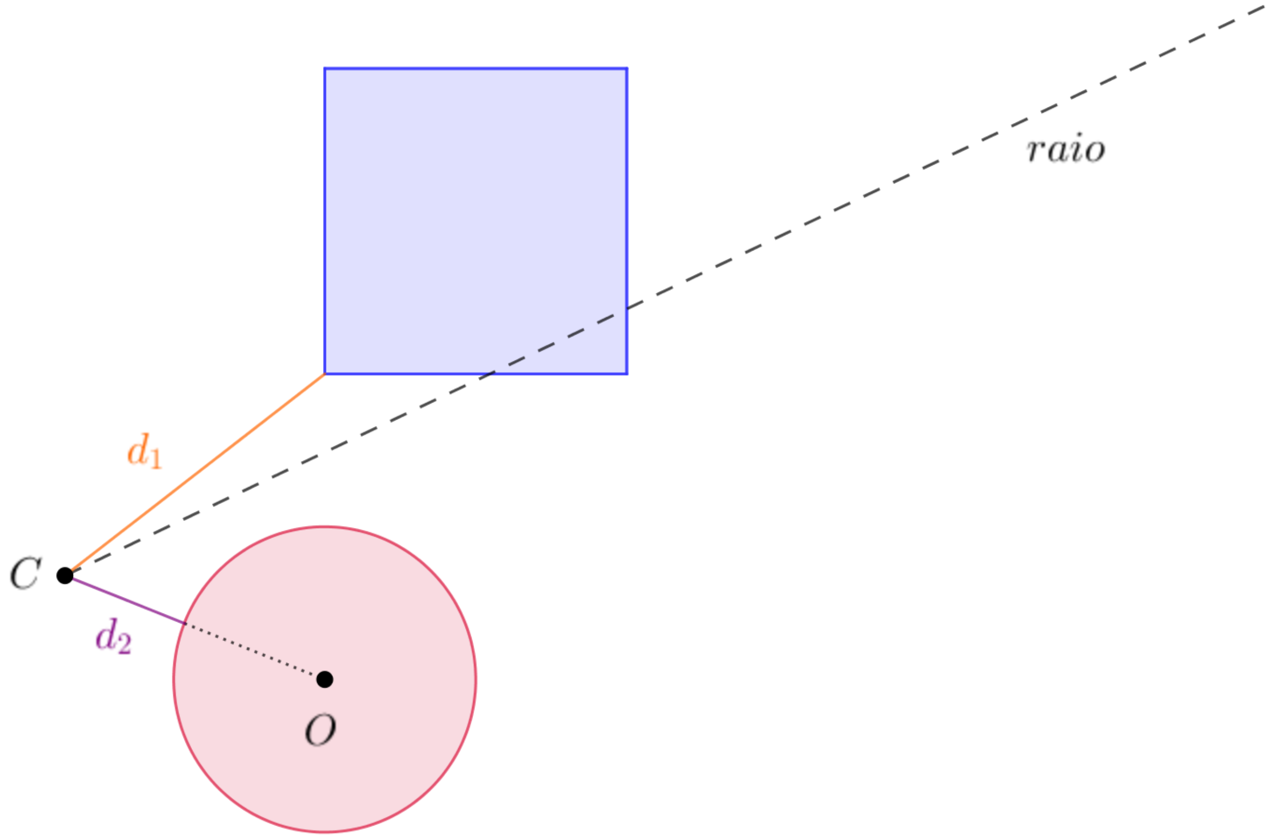
- Regra de como avançamos sobre raio:
    - Calculamos a **SDF** de cada objeto**,** que é a distância mínima de um ponto a cena (conjunto de objetos)
    - Avançamos sobre o raio o valor encontrado (a distância máxima segura sem atravessar a cena)
- Se houver algum objeto perto o bastante do raio (valor pequeno), consideramos que batemos nele
- O algoritmo de Ray Marching vai se aproximando da colisão, ao contrário do Ray Tracing, onde é possível determinar com exatidão se o raio intersectou um objeto.
- Essa imprecisão do Ray Marching acaba permitindo criar efeitos visuais únicos, difíceis de alcançar com outras técnicas.

Ray Marching: Algoritmo

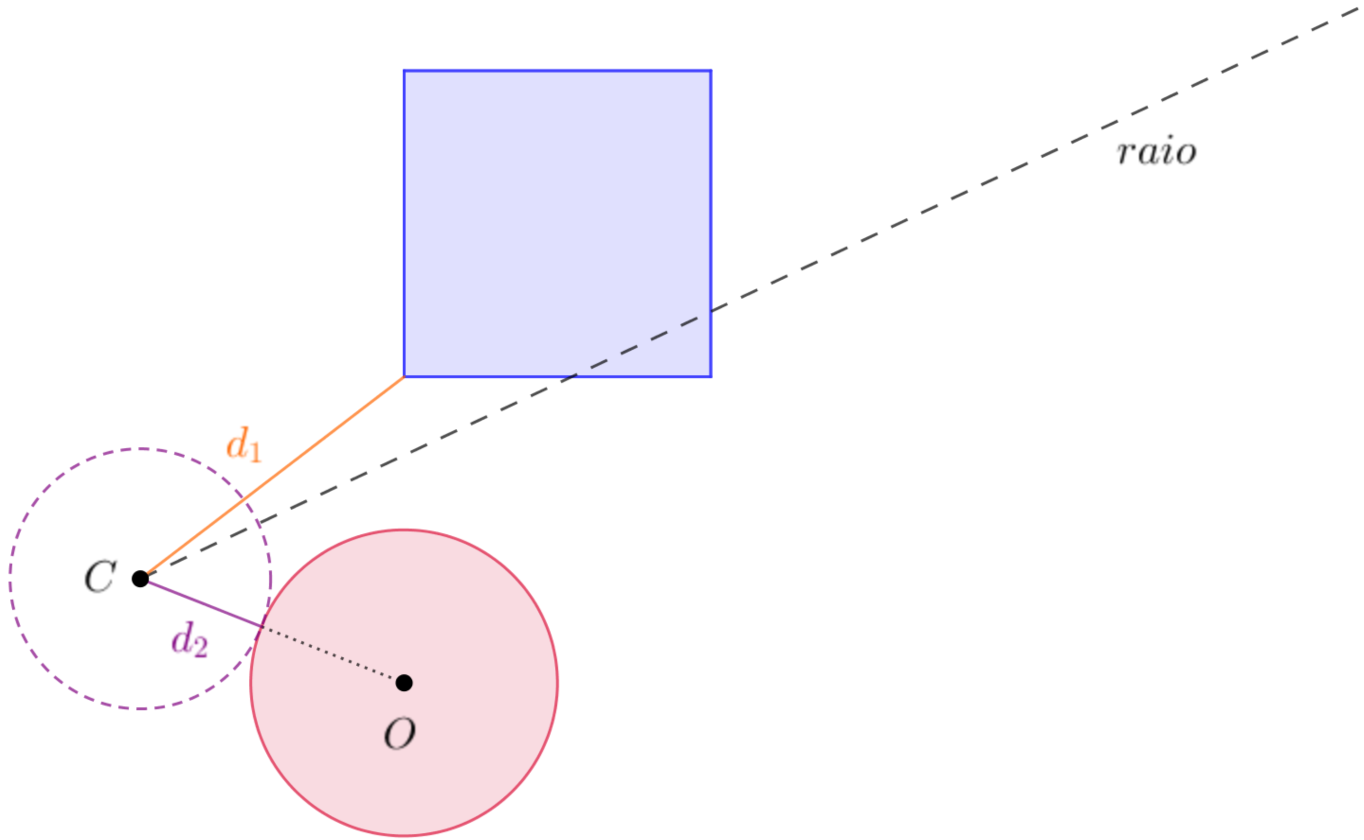


Insper

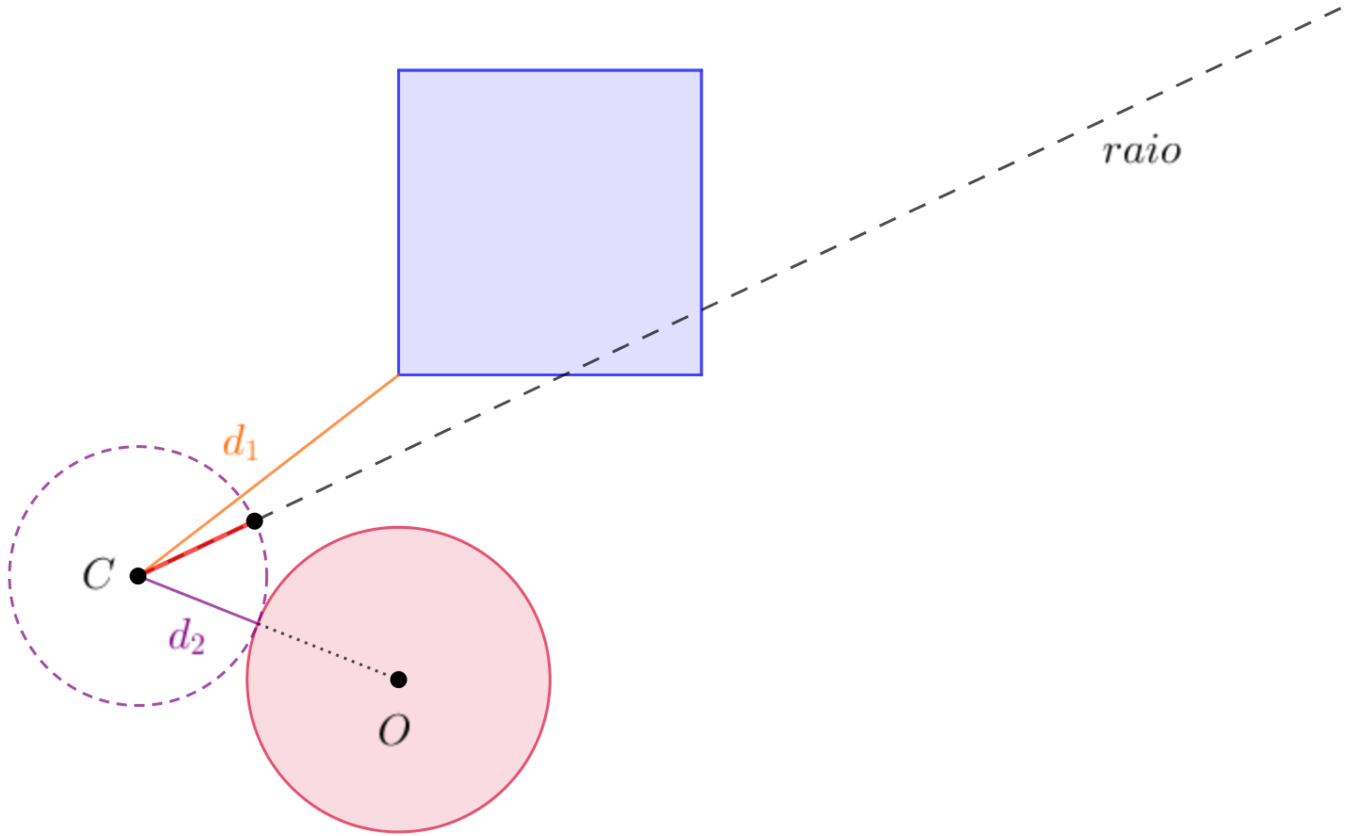
Ray Marching: Algoritmo



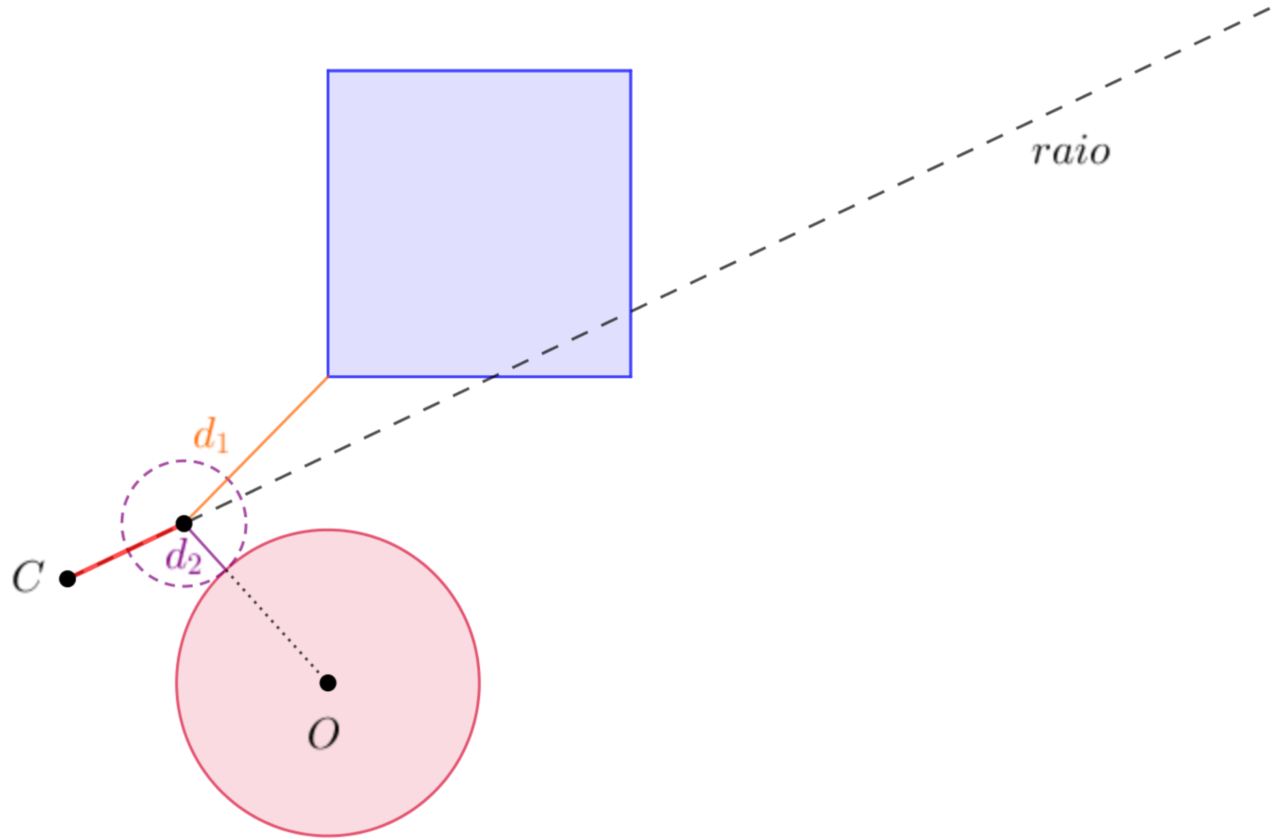
Ray Marching: Algoritmo



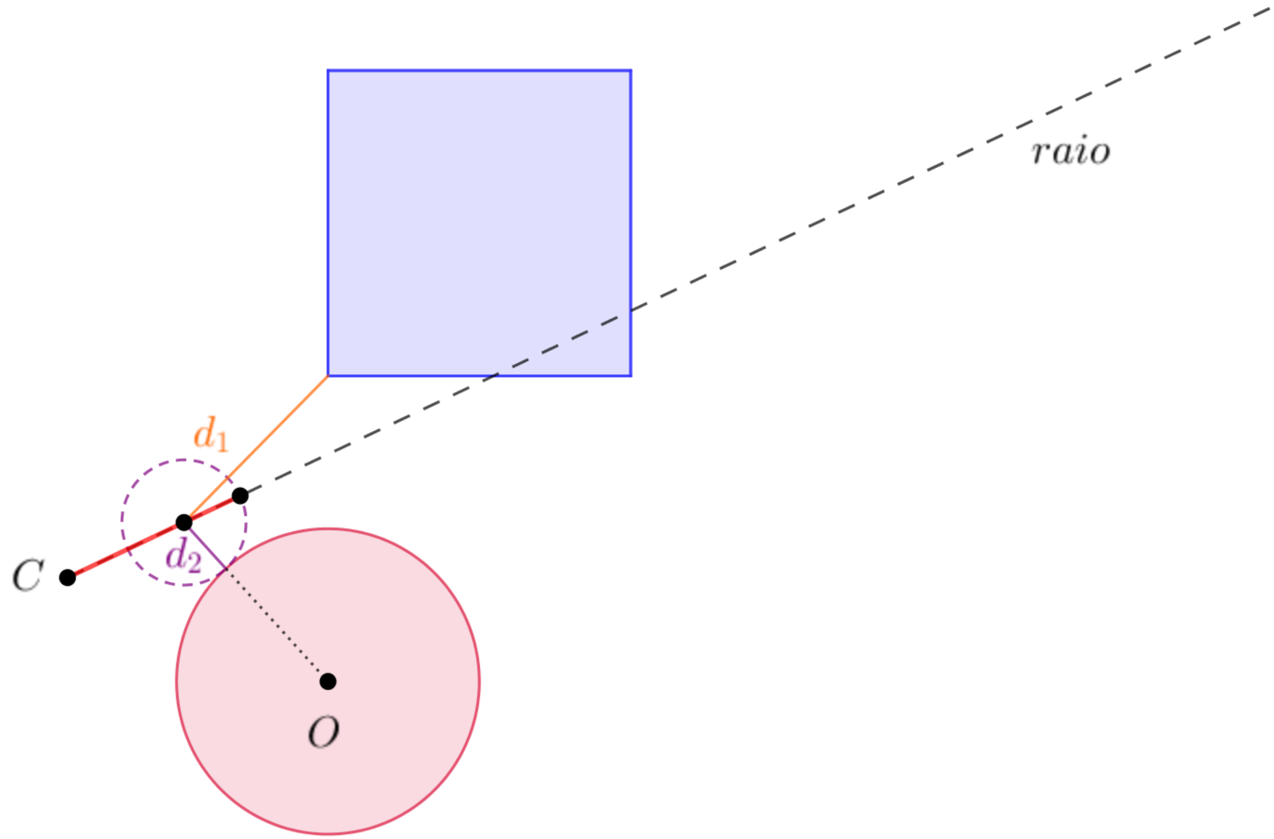
Ray Marching: Algoritmo



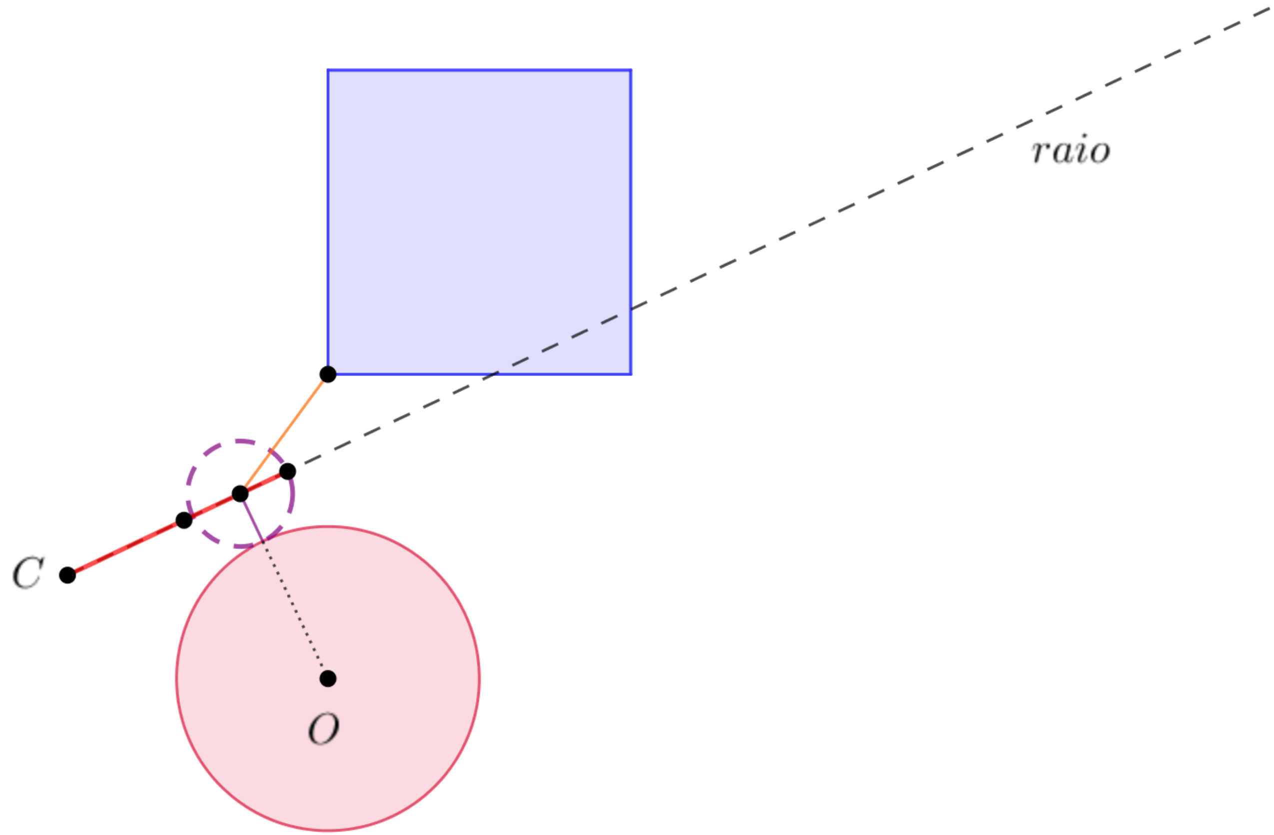
Ray Marching: Algoritmo



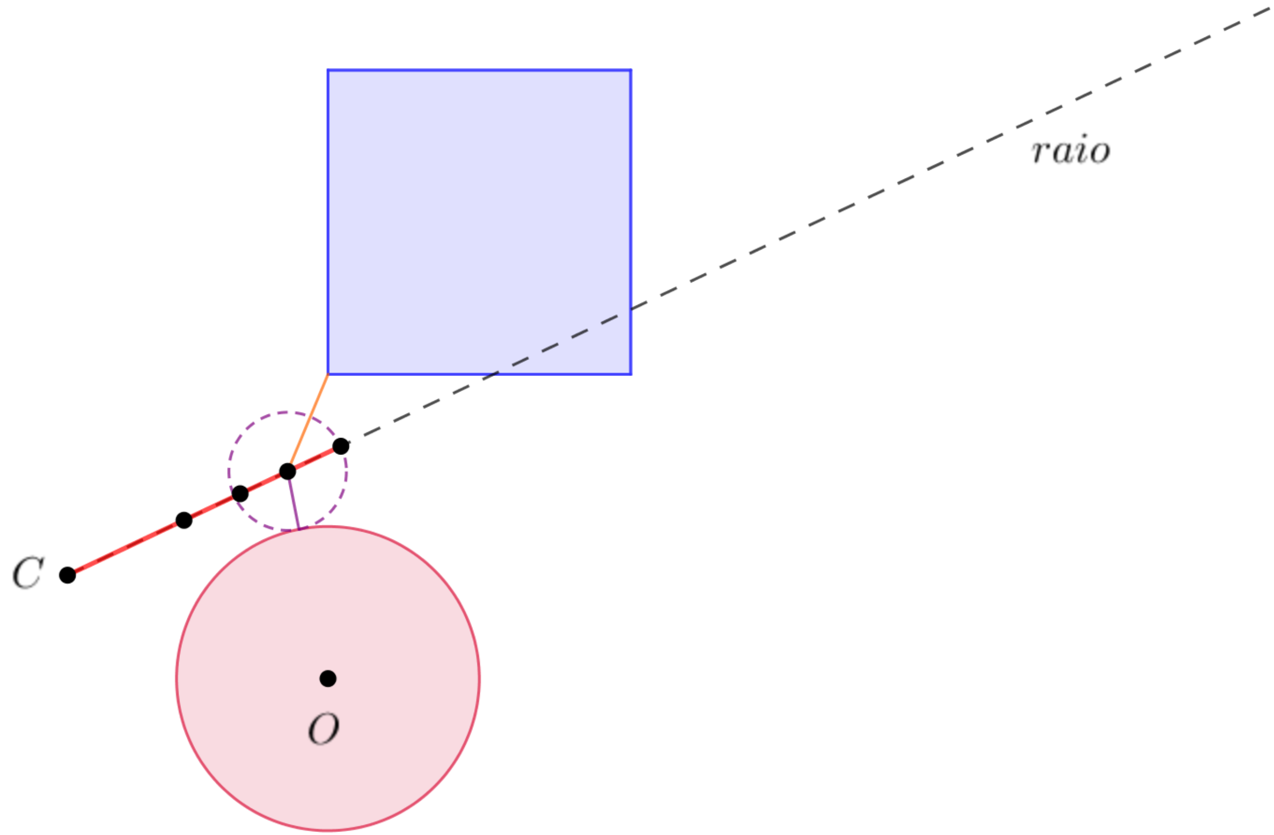
Ray Marching: Algoritmo



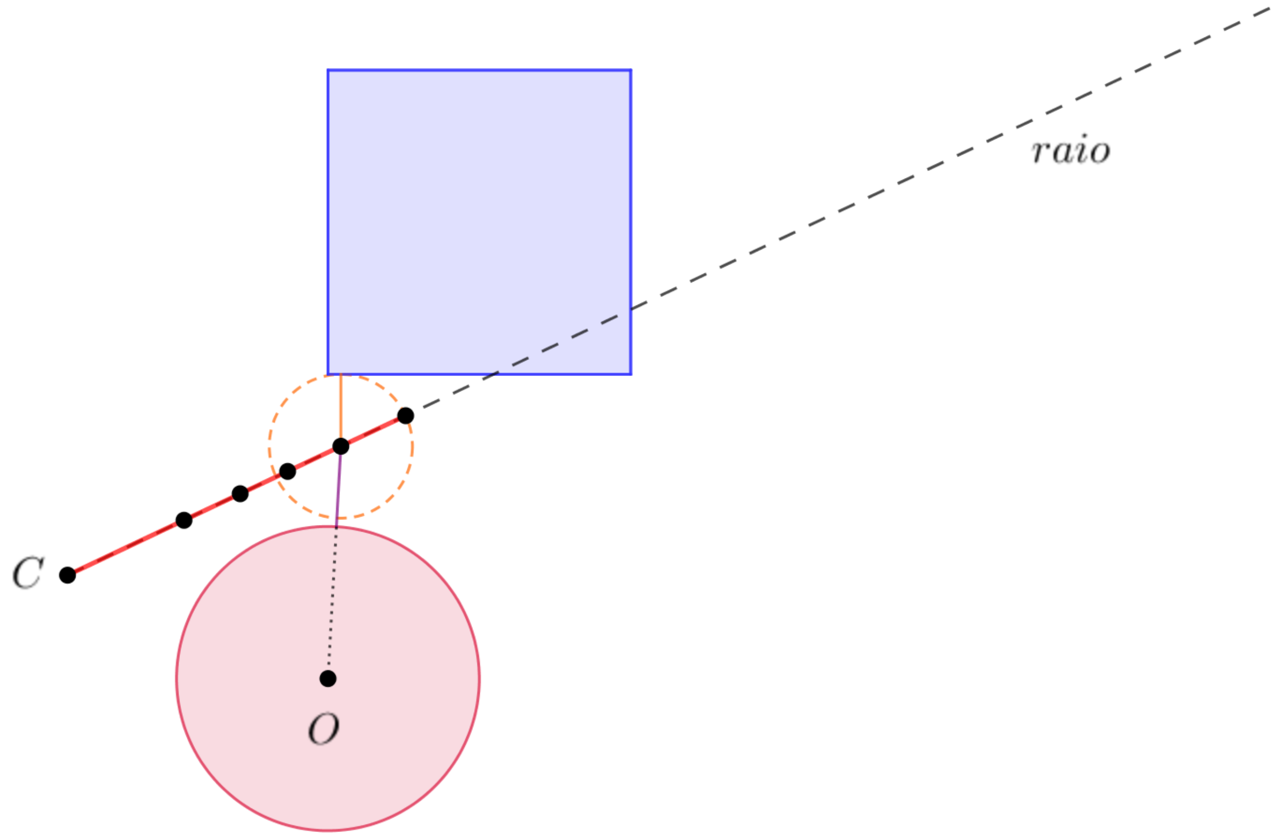
Ray Marching: Algoritmo



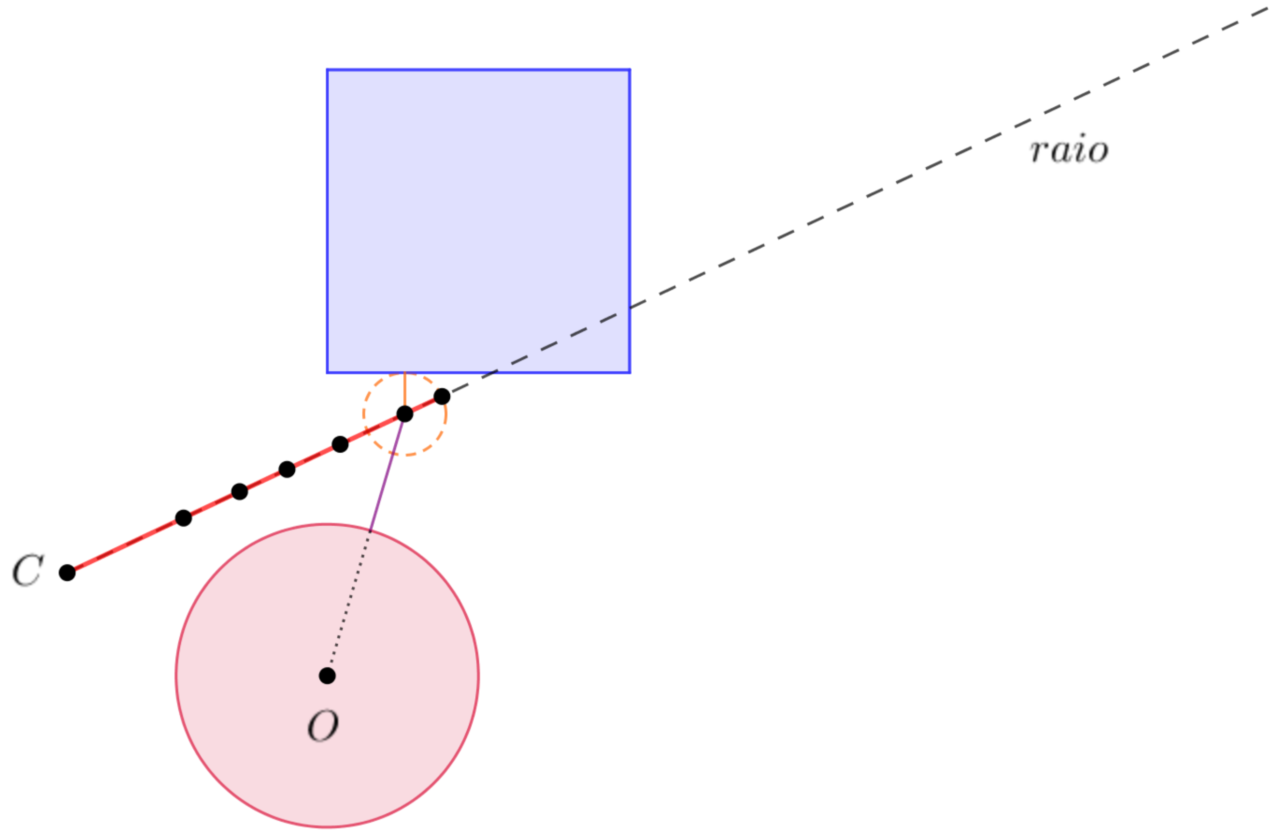
Ray Marching: Algoritmo



Ray Marching: Algoritmo



Ray Marching: Algoritmo



Ray Marching: Algoritmo

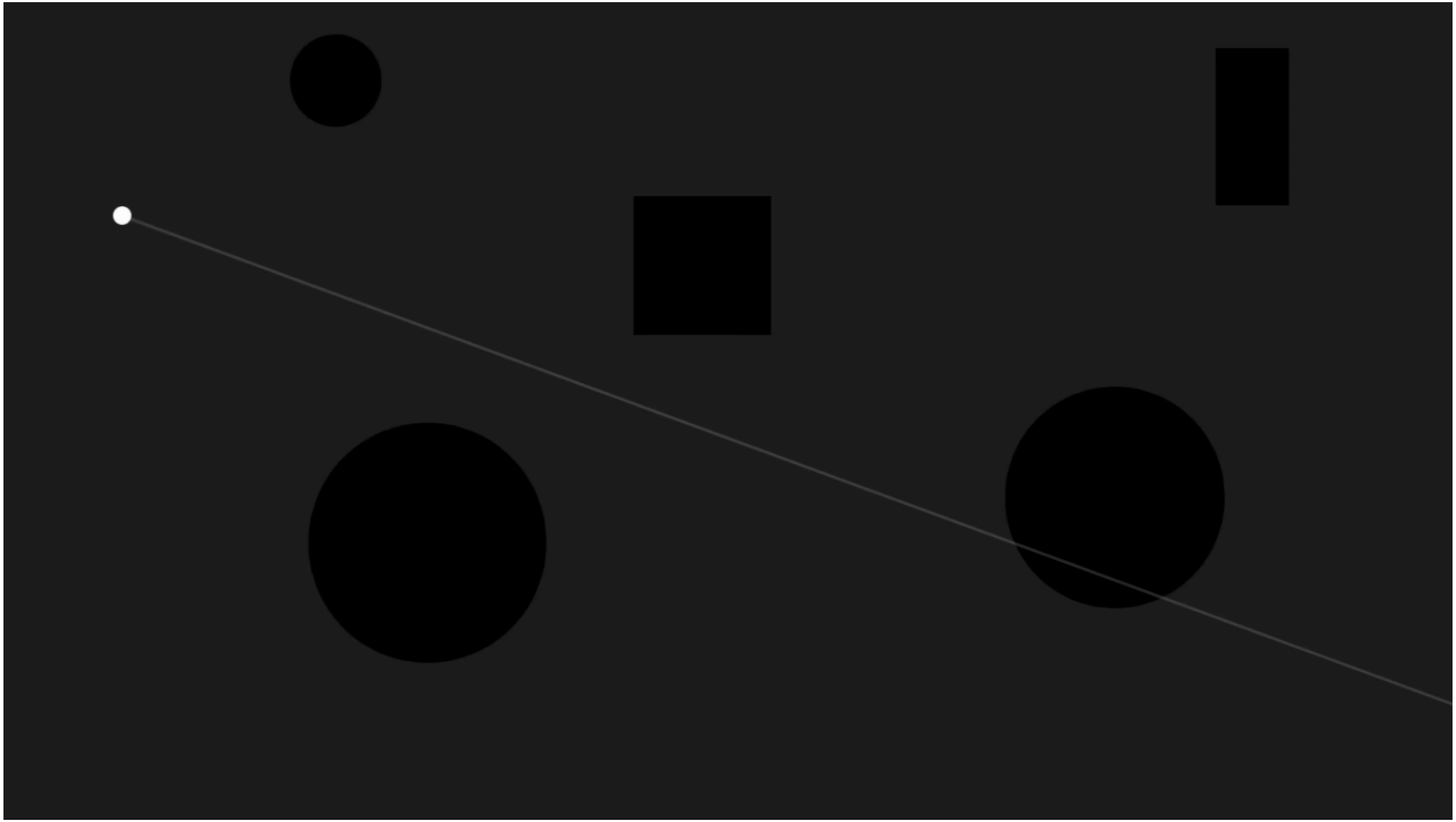
```
Para cada raio lançado:

    Enquanto não batermos em nada ou não ultrapassarmos o limite de marchas:

        Para cada objeto:

            Calcule a SDF do objeto para ter a menor distância entre o
            ponto e ele;

            Se a distância foi a menor achada até agora, guarde ela;

        Se a distância foi menor que um valor mínimo definido, batemos em
        algo, retorne da função

        Senão, ande com o raio a distância encontrada e repita o processo

    Cálculo de iluminação, etc...
```

- O que seria "marchar", ou andar sobre o raio?

```
var new_ray_origin = ray_origin + ray_direction * max_safe_distance;
```


Insper

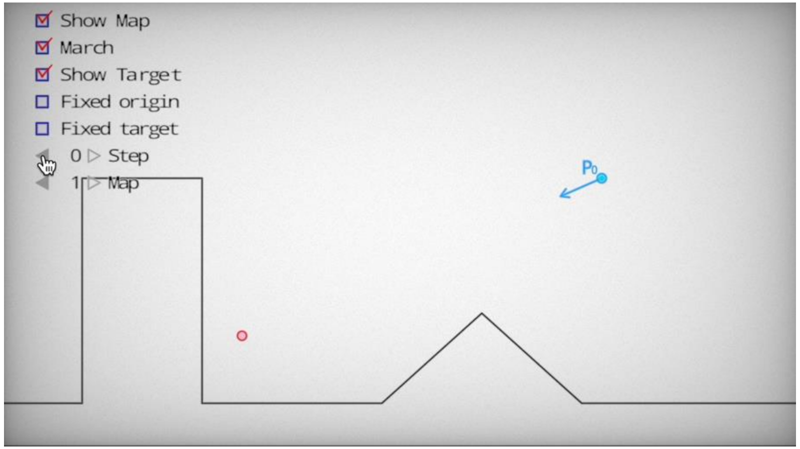
Ray Marching: Visualização

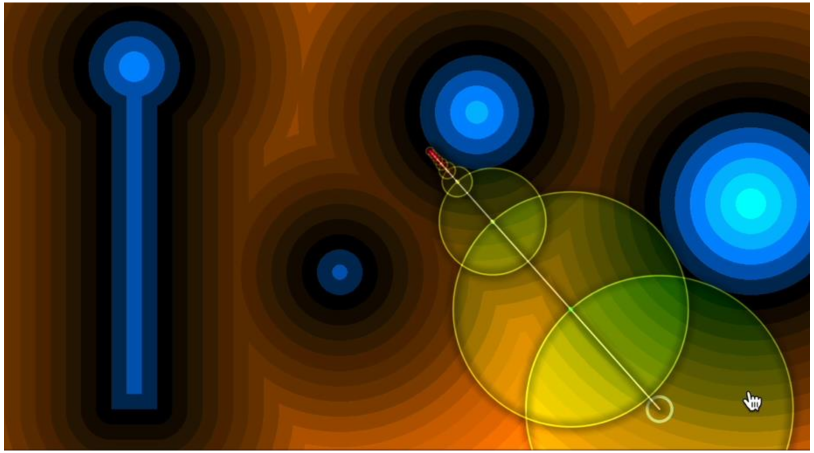


https://youtu.be/Cp5WWtMoeKg

Insper

Ray Marching: Exemplos visualização interativos

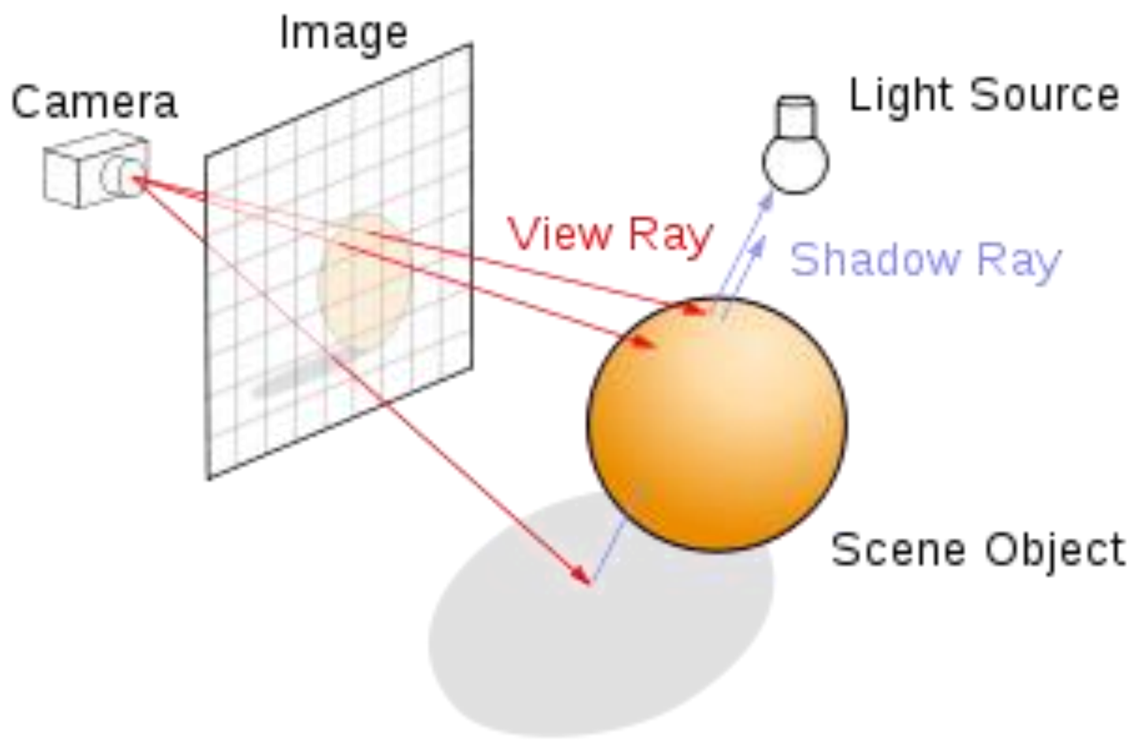




https://www.shadertoy.com/view/4dKyRz
https://www.shadertoy.com/view/4lyBDV

Lançamento de Raios

A origem do lançamento dos raios é a câmera, que podemos dizer que fica atrás da nossa tela.

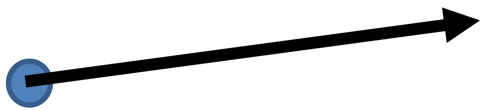


Lançamento de Raios

O raio é definido por uma origem e uma direção. Tanto a origem como a direção do vetor podem ser representados como um vec3f (ou vec2f se for em 2D).

Idealmente trabalhamos com vetores normalizados, ou seja, de magnitude 1.

```
var origin = vec3f(1.0, 2.1, 1.5);
var direction = vec3f(3.0, 2.0, 4.0);

var direction = normalize(direction);
```



Lançamento de Raios - Exemplo

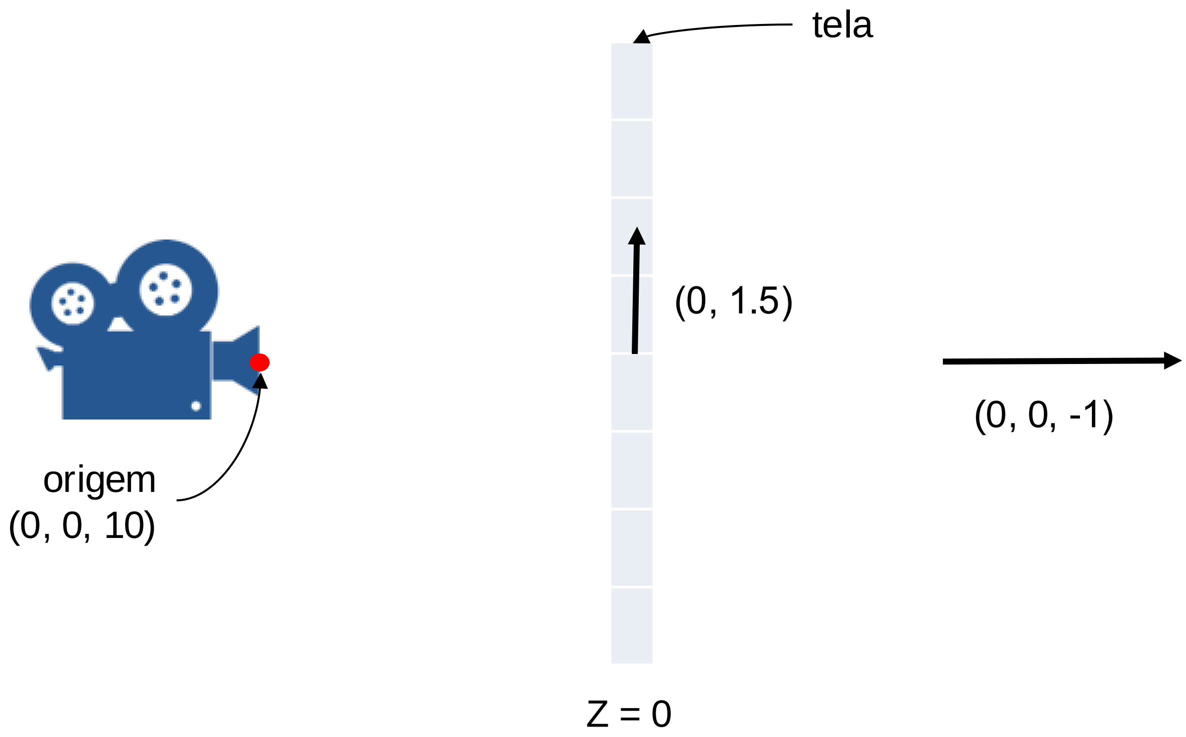
Criando uma cena com a câmera posicionada atrás da tela, apontando para dentro da tela.

```
fn foo()
{
  var uv = fragment_coordinate / resolution.xy;

  var ro = vec3f(0.0, 0.0, 10.0);
  var rd = normalize(vec3f(uv, -1));

  ...
}
```

https://inspirnathan.com/posts/52-shadertoy-tutorial-part-6

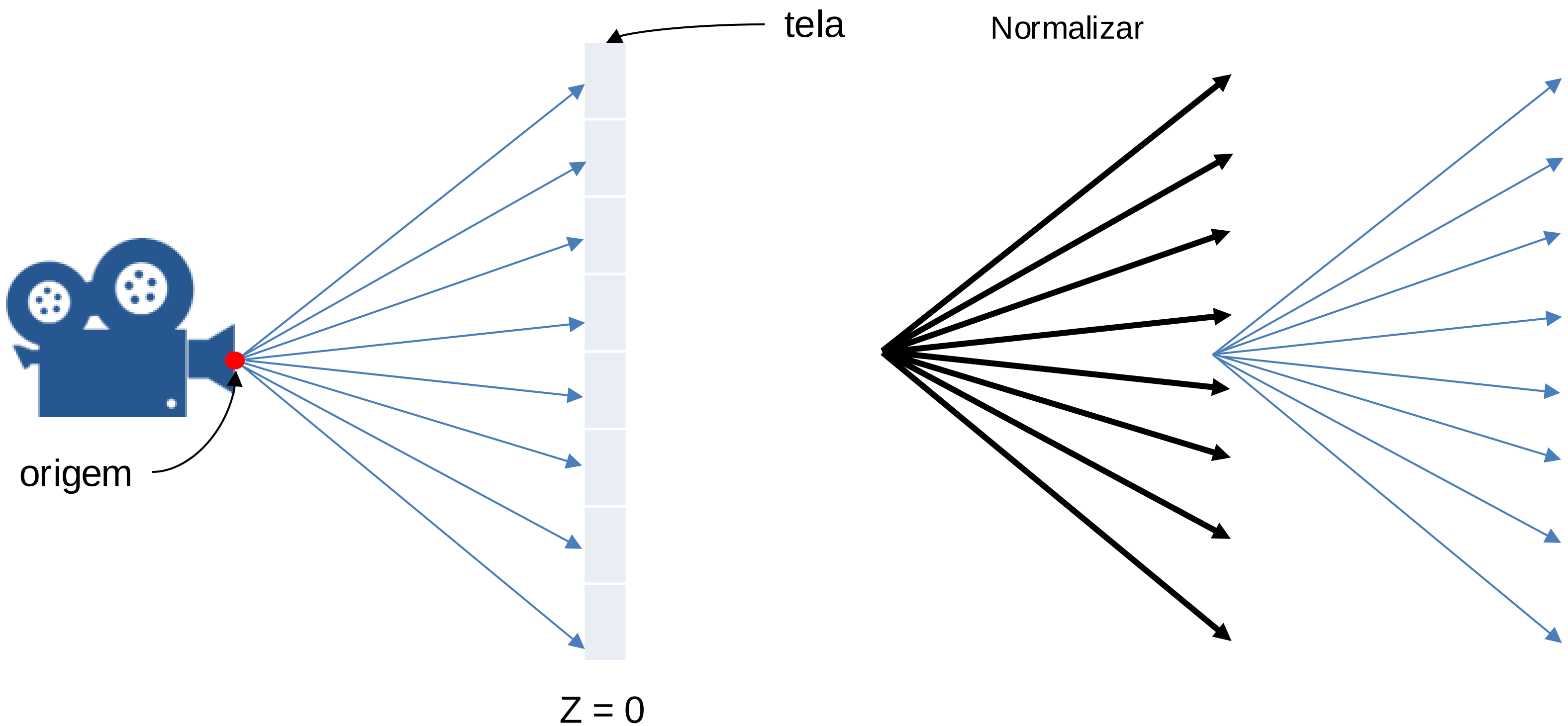
Origem dos raios

A origem do lançamento dos raios é a câmera, que podemos dizer que fica atrás da nossa tela.



Origem dos raios

A origem do lançamento dos raios é a câmera, que podemos dizer que fica atrás da nossa tela.



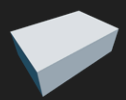
Insper

Distance Functions for Basic Primitives

https://iquilezles.org/articles/distfunctions/

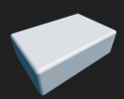
Sphere - exact (https://www.shadertoy.com/view/Xds3zN)

```
float sdSphere( vec3 p, float s )
{
  return length(p)-s;
}
```



Box - exact (Youtube Tutorial with derivation: https://www.youtube.com/watch?v=62-pRVZuS5c)

```
float sdBox( vec3 p, vec3 b )
{
  vec3 q = abs(p) - b;
  return length(max(q,0.0)) + min(max(q.x,max(q.y,q.z)),0.0);
}
```



Round Box - exact

```
float sdRoundBox( vec3 p, vec3 b, float r )
{
  vec3 q = abs(p) - b;
  return length(max(q,0.0)) + min(max(q.x,max(q.y,q.z)),0.0) - r;
}
```

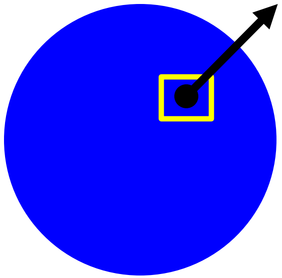
Box Frame - exact (https://www.shadertoy.com/view/3ljcRh)

Insper

Cálculo de Iluminação

Precisamos realizar um cálculo de iluminação para fazer o objeto de fato parecer uma esfera.

O que precisamos saber da superfície para fazer o cálculo de Iluminação?



Precisamos das normais da superfície.

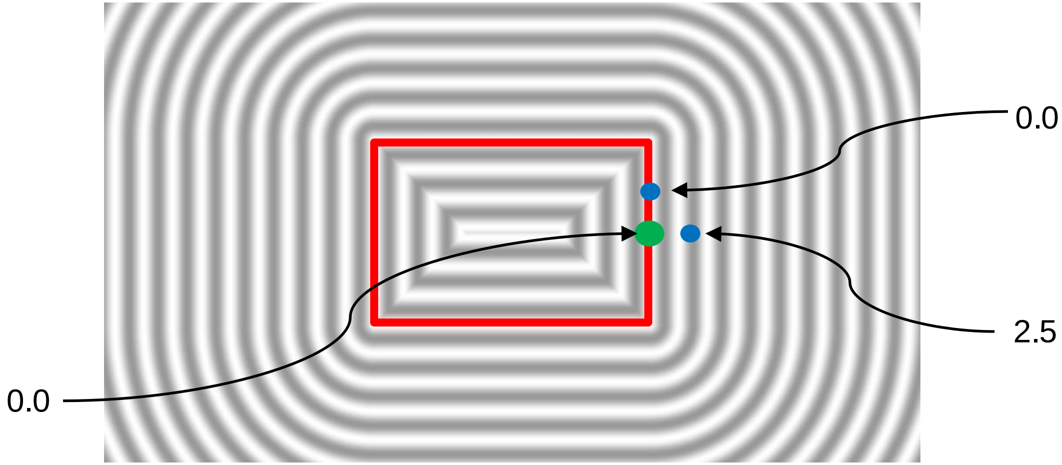
Calculando a Normal pelo Gradiente

Como estamos trabalhando com SDFs, podemos testar agora o que acontece com o valor de distância se nos deslocarmos um pouco para fora do ponto testado.

Veja no exemplo 2D para o ponto verde.

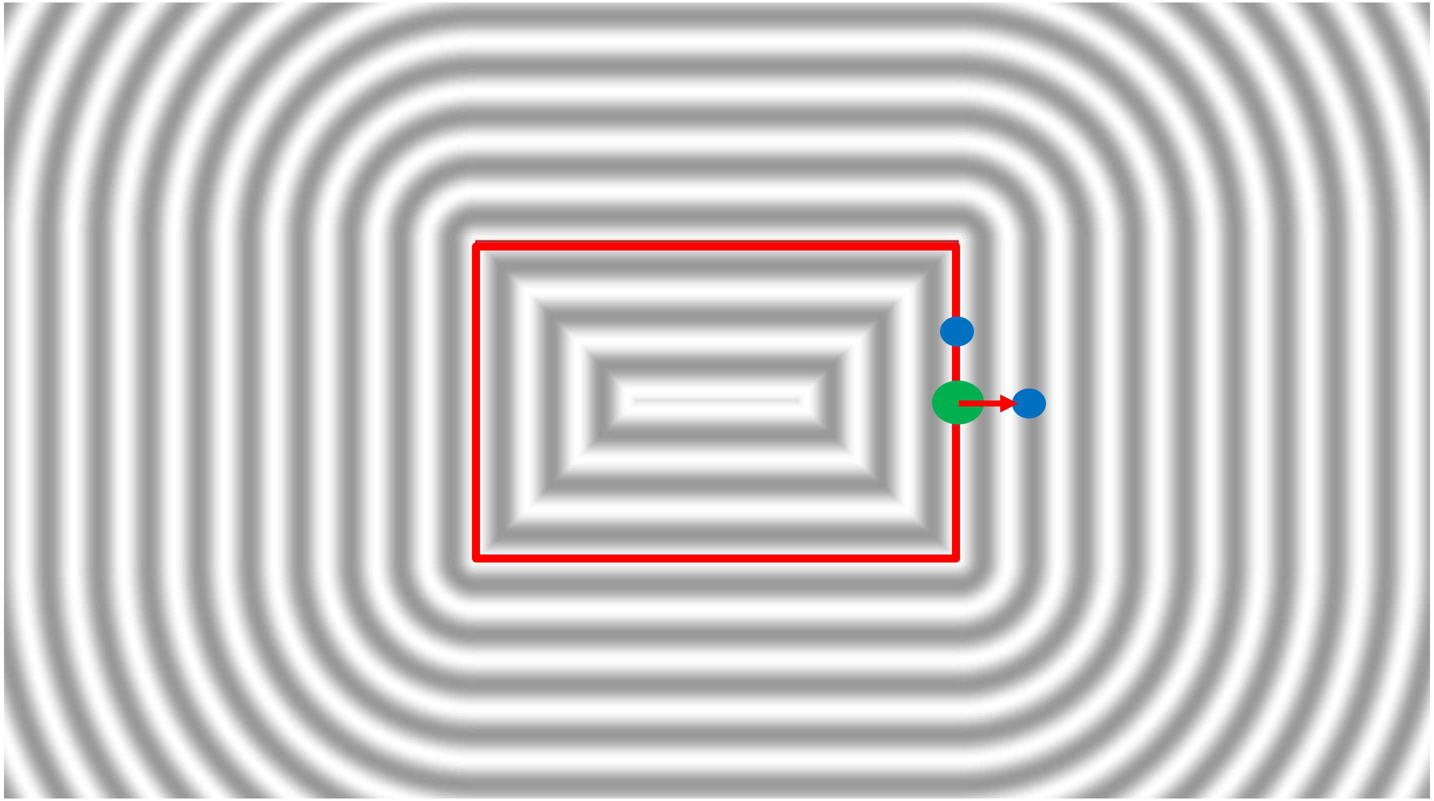
Se testarmos um outro ponto ligeiramente perto do eixo x (horizontal) teremos uma mudança no valor retornado pela função.

Já se testarmos outro ponto em y (vertical) o valor de distância é o mesmo.



Calculando a Normal pelo Gradiente

Baseado nos valores do gradiente, podemos calcular a normal



Gradiente na Superfície

O truque então é testar pontos próximos e ver como o valor da função reage. Usando as variações em cada eixo teremos a valor do gradiente.

$$\nabla f(x,y,z) = \begin{bmatrix} (f(x+\varepsilon,y,z) - f(x-\varepsilon,y,z))/2\varepsilon \\ (f(x,y+\varepsilon,z) - f(x,y-\varepsilon,z))/2\varepsilon \\ (f(x,y,z+\varepsilon) - f(x,y,z-\varepsilon))/2\varepsilon \end{bmatrix}$$

O ε (épsilon) pode ser um valor bem pequeno mesmo.

Calculando a Normal na Superfície

Agora é só normalizar para termos um vetor unitário.

$$\vec{n}(x,y,z) = \frac{\nabla f(x,y,z)}{\|\nabla f(x,y,z)\|}$$

Podemos simplificar a equação e então usar a normal identificada.

$$\vec{n}(x,y,z) = norm\left(\begin{bmatrix} f(x+\varepsilon,y,z) - f(x-\varepsilon,y,z) \\ f(x,y+\varepsilon,z) - f(x,y-\varepsilon,z) \\ f(x,y,z+\varepsilon) - f(x,y,z-\varepsilon) \end{bmatrix}\right)$$

Calculando a Normal na Superfície

$$\nabla f(x,y,z) = \begin{bmatrix} (f(x+\varepsilon,y,z) - f(x-\varepsilon,y,z))/2\varepsilon \\ (f(x,y+\varepsilon,z) - f(x,y-\varepsilon,z))/2\varepsilon \\ (f(x,y,z+\varepsilon) - f(x,y,z-\varepsilon))/2\varepsilon \end{bmatrix}$$

Para o exemplo da esfera:

```
normalize(vec3(
  sdSphere(vec3(p.x + e, p.y, p.z), r) - sdSphere(vec3(p.x - e, p.y, p.z), r),
  sdSphere(vec3(p.x, p.y + e, p.z), r) - sdSphere(vec3(p.x, p.y - e, p.z), r),
  sdSphere(vec3(p.x, p.y, p.z + e), r) - sdSphere(vec3(p.x, p.y, p.z - e), r)
));
```


Insper

Calculando Iluminação

Vamos criar agora uma fonte de luz. Por exemplo:

```
var lightPosition = vec3f(-2, 2, 4);
```

Na sequência criaremos um vetor do ponto da superfície do objeto (p) para essa fonte de luz:

```
var lightDirection = normalize(lightPosition - p);
```

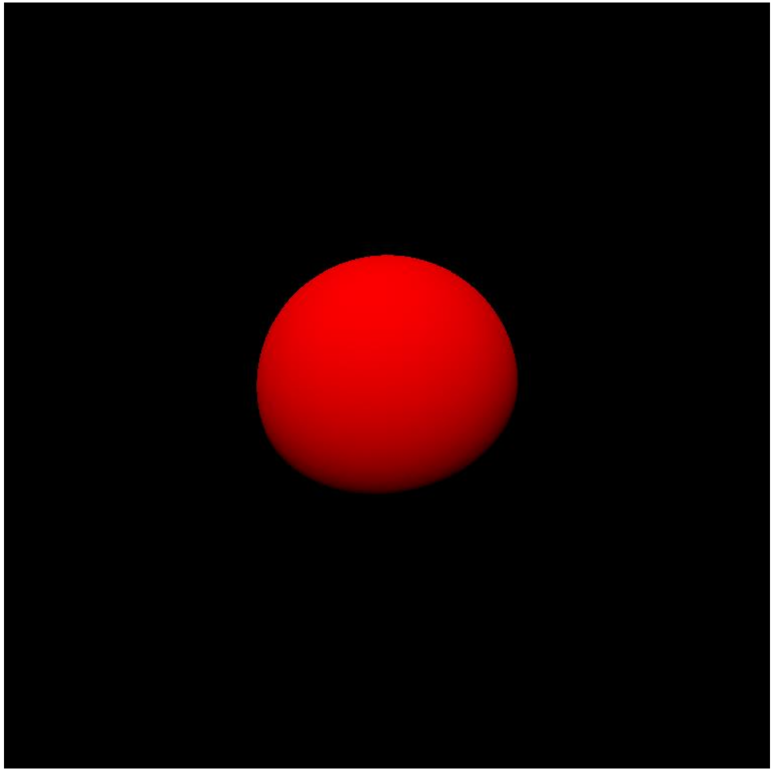
Finalmente vamos fazer o produto escalar (se lembre do cálculo de iluminação) e calcular a cor

```
col = saturate(dot(normal, lightDirection)) * vec3f(1.0, 0.0, 0.0);
```

Obs: saturate() é uma função que limita (clamp) entre 0.0 e 1.0

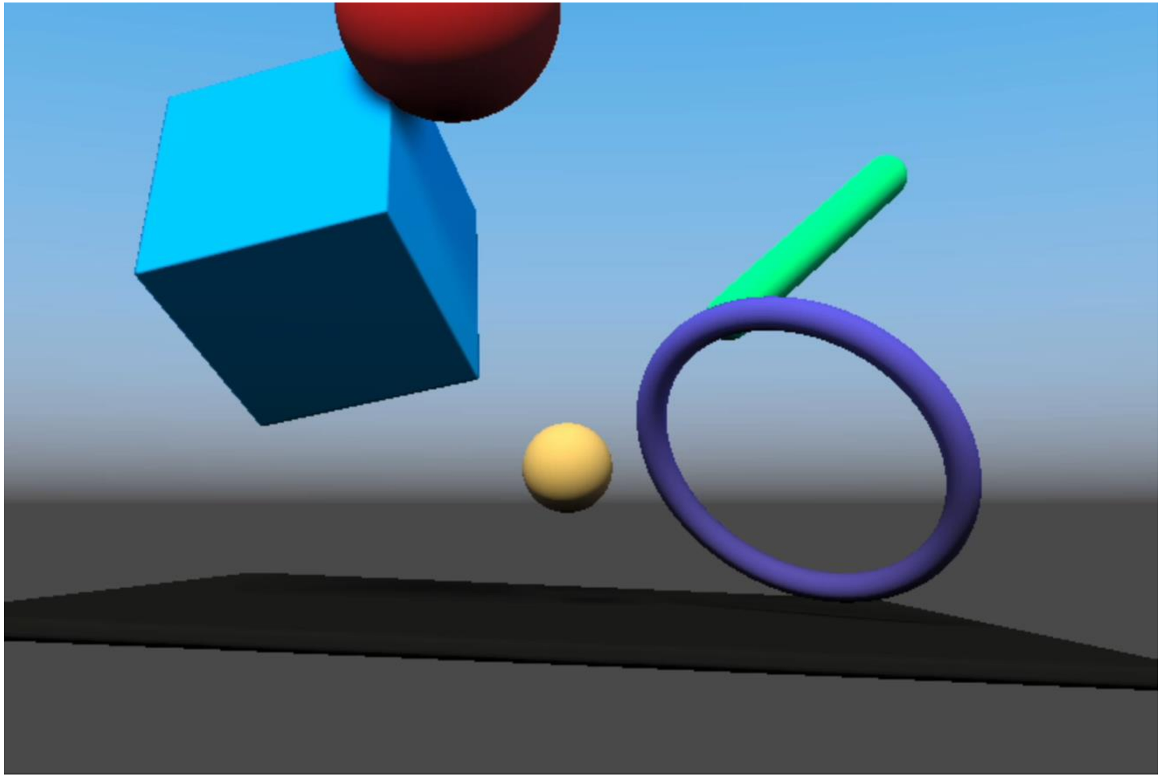
Calculando Iluminação

Até aqui no projeto (cena "Sphere"):



Transformações

Vamos agora trabalhar mais nas transformações.



Translação

Para a translação basta aplicar o inverso do deslocamento do que deseja no objeto.

Por exemplo, se deseja deslocar o objeto +2.0 no X. Você deve alterar o valor de X em -2.0:

```
sdf_sphere(p + vec3f(-2.0, 0.0, 0.0)
```

Porém para ficar mais simples, podemos inverter todo o deslocamento de uma vez:

```
sdf_sphere(p - vec3f(2.0, 0.0, 0.0)
```

Rotação

Para a rotação podemos multiplicar o ponto pelo quatérnio da rotação em Euler.

```
var quat = quaternion_from_euler(sphere.rotation.xyz);

...

fn sdf_sphere(p: vec3f, r: vec4f, quat: vec4f) -> f32
{
    var p_new = rotate_vector(p, quat);
        ...
}
```

Dica: No projeto existe uma "biblioteca" de quatérnios, que você pode usá-la

Insper

Escala

Escala é um problema. A lógica diz para multiplicar pelo inverso da escala. Contudo não vai funcionar direito, pois a escala altera a diferença da função de distância.

```
sdOctahedron(2.0 * p, 1.0);
```

O truque é depois dividir o resultado pela escala.

```
sdOctahedron(2.0 * p, 1.0)/2.0);
```

Projeto

Rúbrica e projeto:
https://github.com/Gustavobb/raymarching-wgsl-template

Gabarito:
https://gubebra.itch.io/raymarching-webgpu

Avaliação Disciplina

Blackboard

Referências

https://inspirnathan.com/posts/52-shadertoy-tutorial-part-6

https://jamie-wong.com/2016/07/15/ray-marching-signed-distance-functions/

https://iquilezles.org/articles/raymarchingdf/

http://bentonian.com/Lectures/FGraphics1819/1.%20Ray%20Marching%20and%20Signed%20Distance%20Fields.pdf

https://www.shadertoy.com/view/ltyXD3

Computação Gráfica

Luciano Soares
<lpsoares@insper.edu.br>

Fabio Orfali
<fabioo1@insper.edu.br>

Gustavo Braga
<gustavobb1@insper.edu.br>