

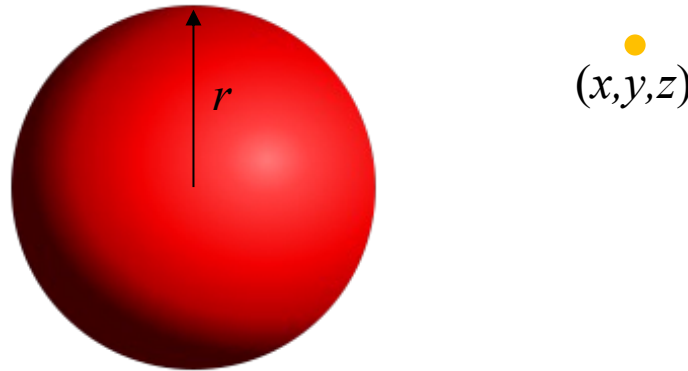
# Computação Gráfica

Aula 23: Ray Marching

# 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.

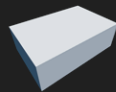
# Distance Functions for Basic Primitives

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



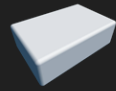
**Sphere - exact** (<https://www.shadertoy.com/view/Xds3zN>)

```
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**

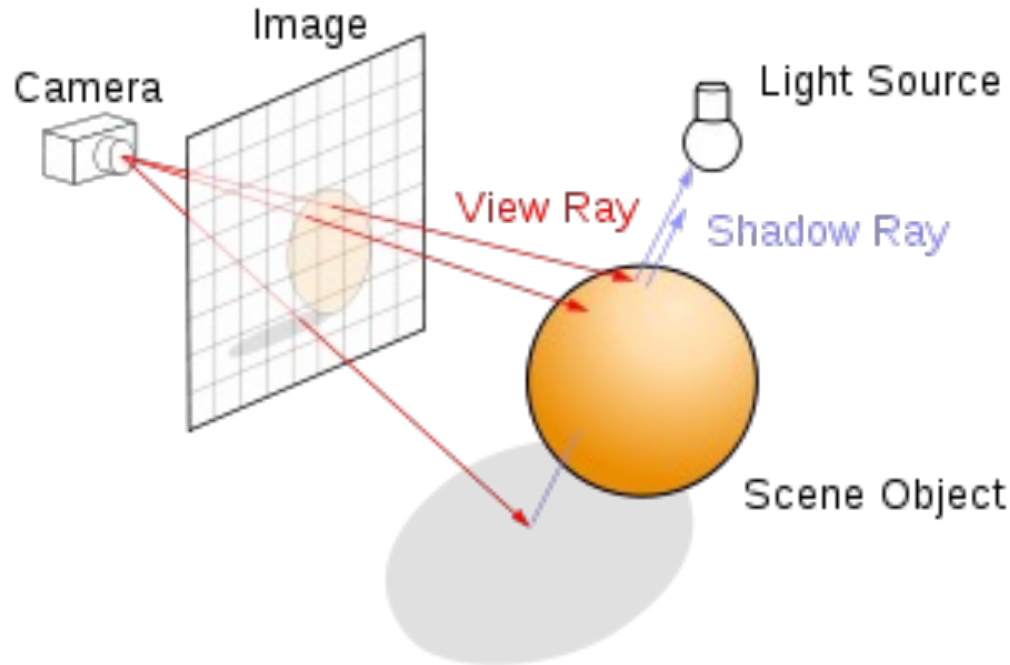
```
float sdRoundBox( vec3 p, vec3 b, float r )
{
    vec3 q = abs(p) - b;
    return length(max(q,0.0)) + min(max(q.x,max(q.y,q.z)),0.0) - r;
}
```



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

# Lançamento de Raios

Uma das propostas de renderização 3D é lançar um raio. Procurar a primeira superfície de intersecção desse raio. Agregar informações como a direção das fontes de luz ou sombras.



# Origem e Direção dos Raios

O raio é definido por uma origem e um destino. Tanto a origem como a direção do vetor podem ser representados como um `vec3` (ou `vec2` se for em 2D).

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

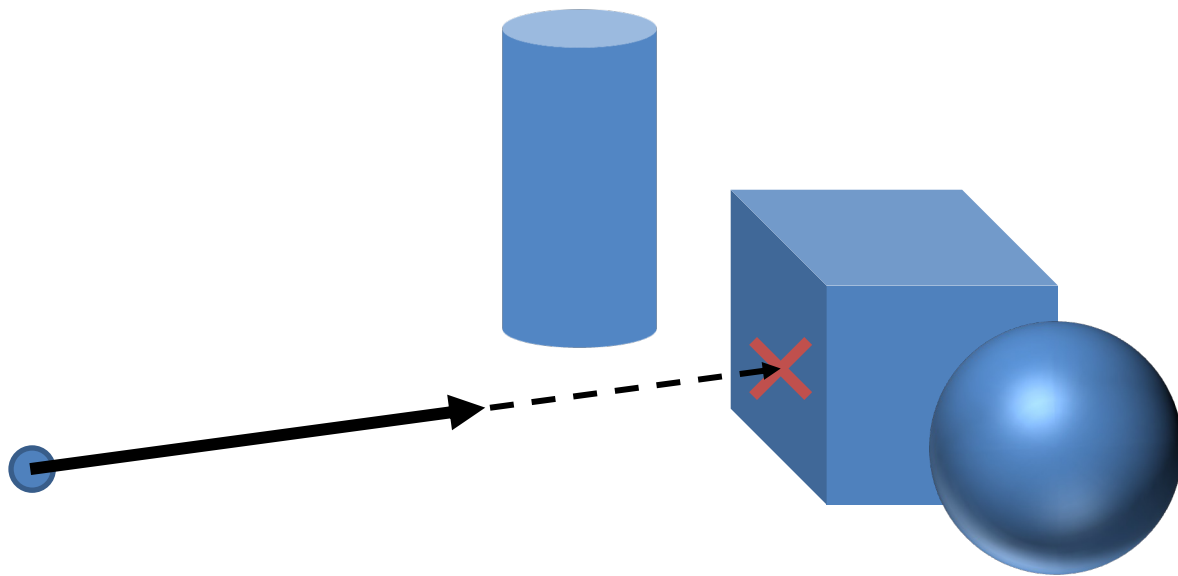
```
vec3 origin = vec3(1.0, 2.1, 1.5);  
vec3 direction = vec3(3.0, 2.0, 4.0);  
  
vec3 direction = normalize(direction);
```



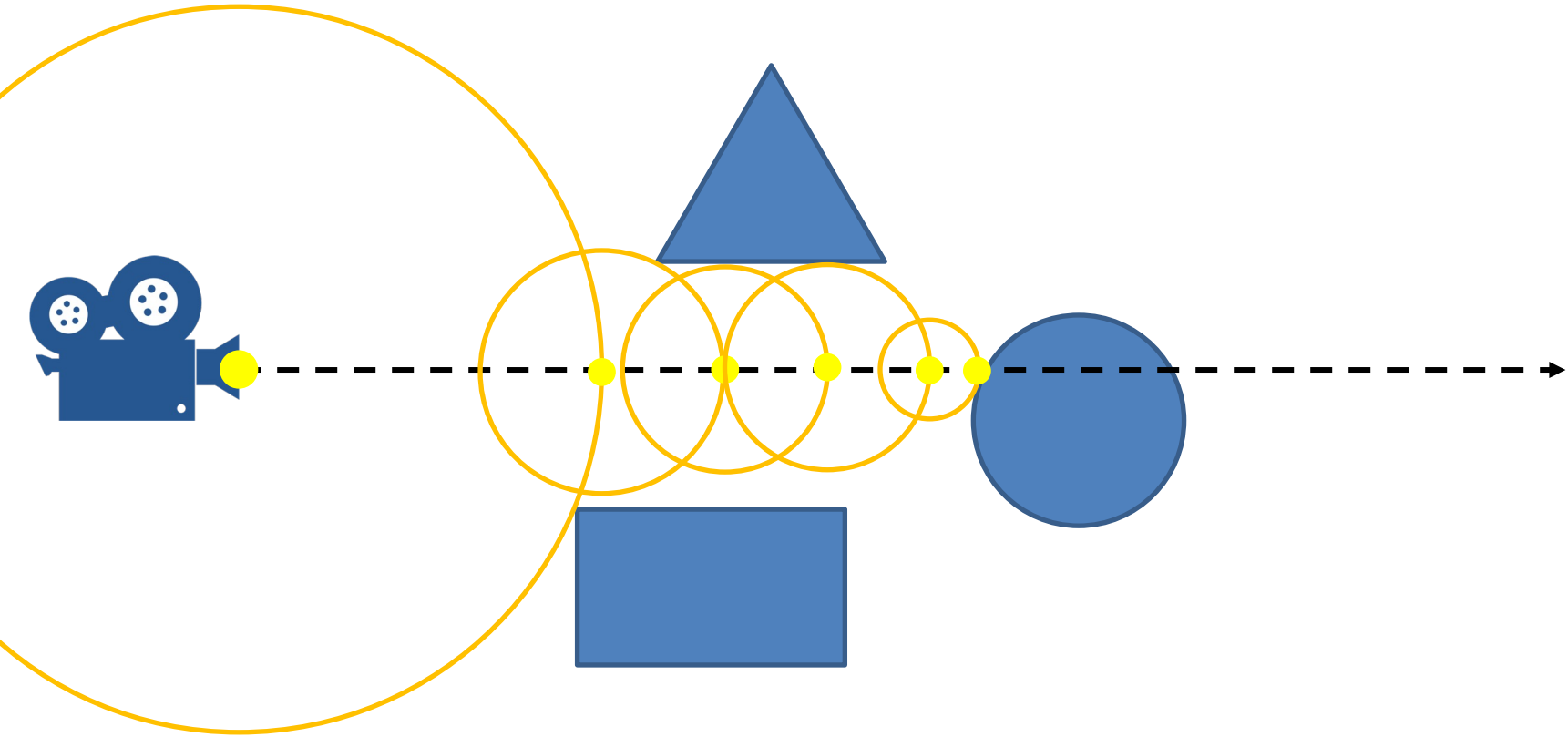
# Raios

Agora temos de verificar se o raio atinge alguma geometria na cena.

Uma das técnicas conhecida é o Ray Tracing que calcula essa intersecção, mas aqui queremos usar SDFs.

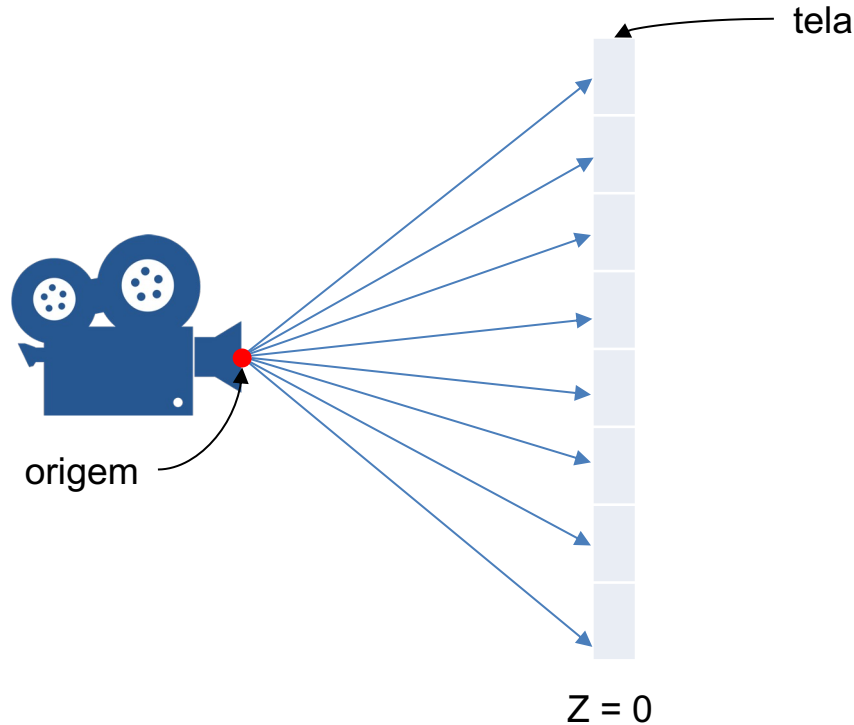


# Como sabemos se há um objeto?



# Origem dos raios

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





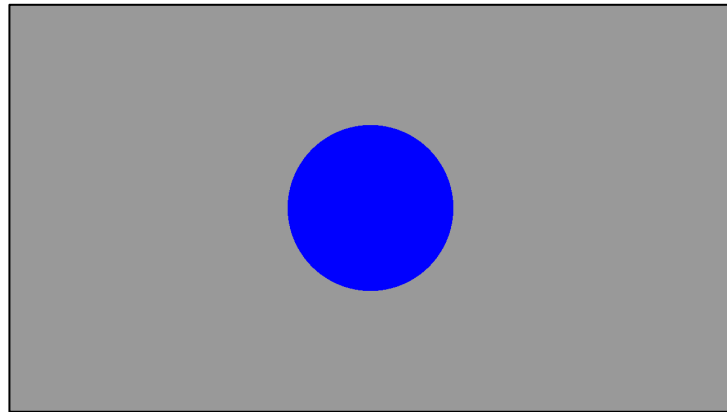
# Setup inicial para Ray Marching

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

```
void mainImage( out vec4 fragColor, in vec2 fragCoord ) {  
  
    vec2 uv = (fragCoord - .5 * iResolution.xy) / iResolution.y;  
  
    vec3 ro = vec3(0.0, 0.0, 10.0);  
    vec3 rd = normalize(vec3(uv, -1));  
  
    vec3 col = vec3(smoothstep(0.49, 0.5, abs(uv)), 0.0);  
    fragColor = vec4(col, 1.0);  
  
}
```

# Buscando uma esfera

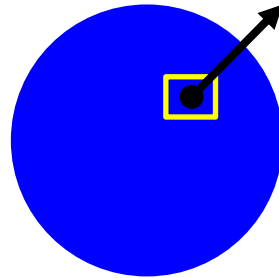
```
float sdSphere(vec3 p, float r) {  
    return length(p) - r;  
}  
  
float rayMarch(vec3 ro, vec3 rd, float start, float end) {  
    float depth = start;  
    for (int i = 0; i < 255; i++) {  
        vec3 p = ro + depth * rd;  
        float d = sdSphere(p, 1.0);  
        depth += d;  
        if (d < 0.001 || depth > end) break;  
    }  
    return depth;  
}  
  
void mainImage( out vec4 fragColor, in vec2 fragCoord ) {  
    vec2 uv = (fragCoord - .5*iResolution.xy)/iResolution.y;  
  
    vec3 col = vec3(0);  
    vec3 ro = vec3(0, 0, 5);  
    vec3 rd = normalize(vec3(uv, -1));  
  
    float d = rayMarch(ro, rd, 0.01, 100.0);  
  
    if (d > 100.0) col = vec3(0.6);  
    else col = vec3(0, 0, 1);  
  
    fragColor = vec4(col, 1.0);  
}
```



# Cálculo de Iluminação

A esfera parece um puro círculo, vamos incluir um cálculo de iluminação para fazer o objeto de fato parecer com uma esfera.

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



**Precisamos das normais da superfície.**

# Cálculo da Normal por Gradiente

Para descobrir a Normal vamos usar uma técnica de gradiente (muitas vezes representado com o símbolo  $\nabla$ ).

O gradiente é onde temos o maior valor de derivada. O que para nós significa um vetor perpendicular da curva ou superfície.

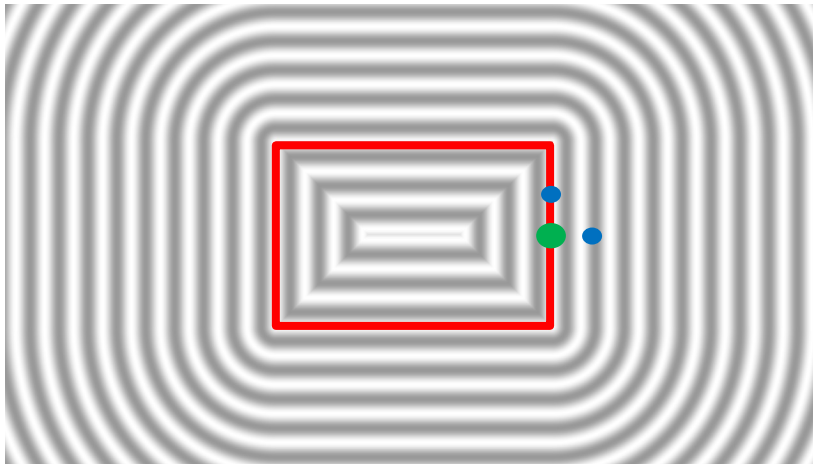
Conceitualmente, o gradiente de uma função  $f$  no ponto  $(x,y,z)$  diz a você em que direção se mover  $(x,y,z)$  para aumentar mais rapidamente o valor de  $f$ . Esta será a nossa superfície normal.

$$\nabla f = \left( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right)$$

# Calculando a Normal na Superfície

Como estamos trabalhando com SDFs, podemos testar agora o que acontece com o valor de distância se nos locomovermos 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 da função. Já se testarmos outro ponto em y (vertical) o valor de distância é o mesmo.



cinza

# 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 direção 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  $\varepsilon$  (épsilon) pode ser um valor bem pequeno mesmo.

# Calculando a Normal na Superfície

Agora é só normalizamos para ter 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.

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

O  $\varepsilon$  (épsilon) pode ser um valor bem pequeno mesmo.

# Calculando a Normal na Superfície em GLSL

Para calcular as normais na esfera, podemos usar:

```
vec3 calcNormal(vec3 p) {
    float e = 0.0005; // epsilon
    float r = 1.0; // raio da esfera
    return 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)
    ));
}
```

Usando alguns truques de programação podemos simplificar para:

```
vec3 calcNormal(vec3 p) {
    vec2 e = vec2(1.0, -1.0) * 0.0005; // epsilon
    float r = 1.0; // raio da esfera
    return normalize(
        e.xyy * sdSphere(p + e.xyy, r) + e.yyx * sdSphere(p + e.yyx, r) +
        e.yxy * sdSphere(p + e.yxy, r) + e.xxx * sdSphere(p + e.xxx, r));
}
```



# Verificando o cálculo

```
float sdSphere(vec3 p, float r) { ... }

float rayMarch(vec3 ro, vec3 rd, float start, float end) {
    ...
}

vec3 calcNormal(vec3 p) {
    vec2 e = vec2(1.0, -1.0) * 0.0005; // epsilon
    float r = 1.; // raio da esfera
    return normalize(
        e.xyy * sdSphere(p + e.xyy, r) +
        e.yyx * sdSphere(p + e.yyx, r) +
        e.yxy * sdSphere(p + e.yxy, r) +
        e.xxx * sdSphere(p + e.xxx, r));
}

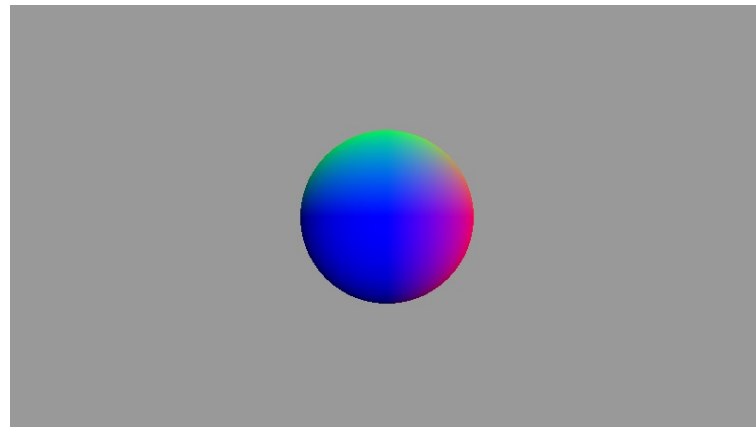
void mainImage( out vec4 fragColor, in vec2 fragCoord ) {
    ...
    float d = rayMarch(ro, rd, 0., 100.);

    if (d > 100.0) col = vec3(0.6);
    else {
        vec3 p = ro + rd * d;
        col = calcNormal(p);
    }

    fragColor = vec4(col, 1.0);
}
```

Uma boa prática é sempre ir verificando o que se consegue.

Como será a imagem?



# Calculando Iluminação

Vamos criar agora uma fonte de luz. Por exemplo:

```
vec3 lightPosition = vec3(-2, 2, 4);
```

Agora vamos criar um vetor no ponto sendo renderizado da superfície que aponte para essa fonte de luz:

```
vec3 lightDirection = normalize(lightPosition - p);
```

Finalmente vamos fazer o produto escalar e calcular a cor

```
col = vec3(clamp(dot(normal, lightDirection), 0., 1.));
```

# Iluminando a esfera

```
float sdSphere(vec3 p, float r) { ... }

float rayMarch(vec3 ro, vec3 rd, float start, float end) {
    ...
}

vec3 calcNormal(vec3 p) {
    ...
}

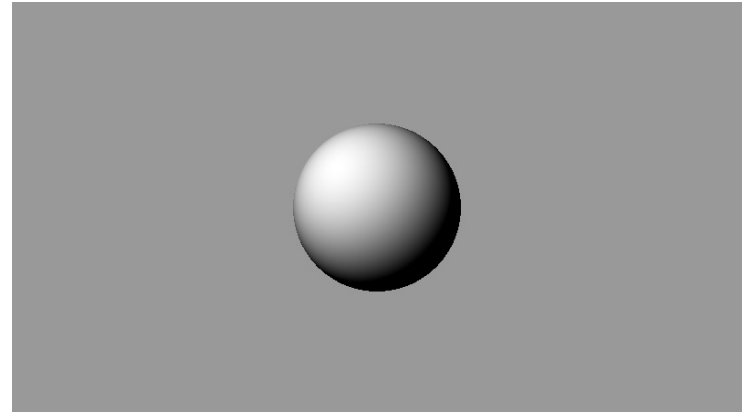
void mainImage( out vec4 fragColor, in vec2 fragCoord ) {
    ...
    float d = rayMarch(ro, rd, 0., 100.);

    if (d > 100.0) col = vec3(0.6);
    else {
        vec3 p = ro + rd * d;
        vec3 normal = calcNormal(p);
        vec3 lightPos = vec3(-2, 2, 4);
        vec3 lightDir = normalize(lightPos - p);

        col = vec3(clamp(dot(normal, lightDir), 0., 1.));
    }

    fragColor = vec4(col, 1.0);
}
```

Como será a imagem?



# Material e Luz Ambiente

```
float sdSphere(vec3 p, float r) { ... }

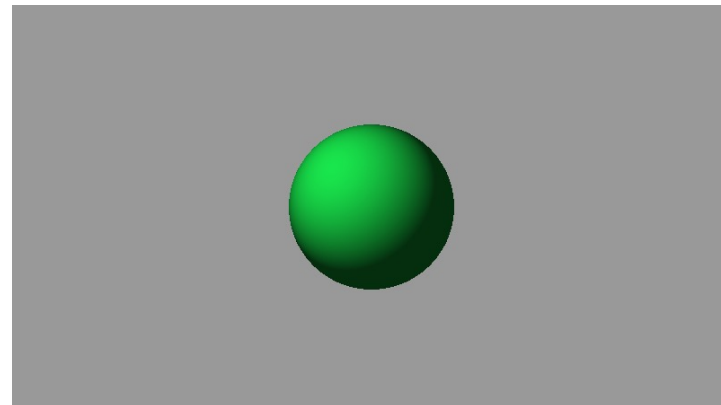
float rayMarch(vec3 ro, vec3 rd, float start, float end) {
    ...
}

vec3 calcNormal(vec3 p) {
    ...
}

void mainImage( out vec4 fragColor, in vec2 fragCoord ) {
    ...
    float d = rayMarch(ro, rd, 0., 100.);

    if (d > 100.0) col = vec3(0.6);
    else {
        vec3 p = ro + rd * d;
        vec3 normal = calcNormal(p);
        vec3 lightPos = vec3(-2, 2, 4);
        vec3 lightDir = normalize(lightPos - p);
        float ambient = 0.2;
        float difuse = clamp(dot(normal, lightDir), ambient, 1.);
        col = difuse * vec3(0.1, 0.9, 0.3);
    }

    fragColor = vec4(col, 1.0);
}
```



# Múltiplos objetos

Existem diversas estratégias de gerenciar múltiplos objetos em Ray Marching. Uma das propostas é gerenciar melhor a cena. Vamos colocar um melhor controle de objetos, criando uma função 'sdScene' que vai tratar todos os objetos da cena.

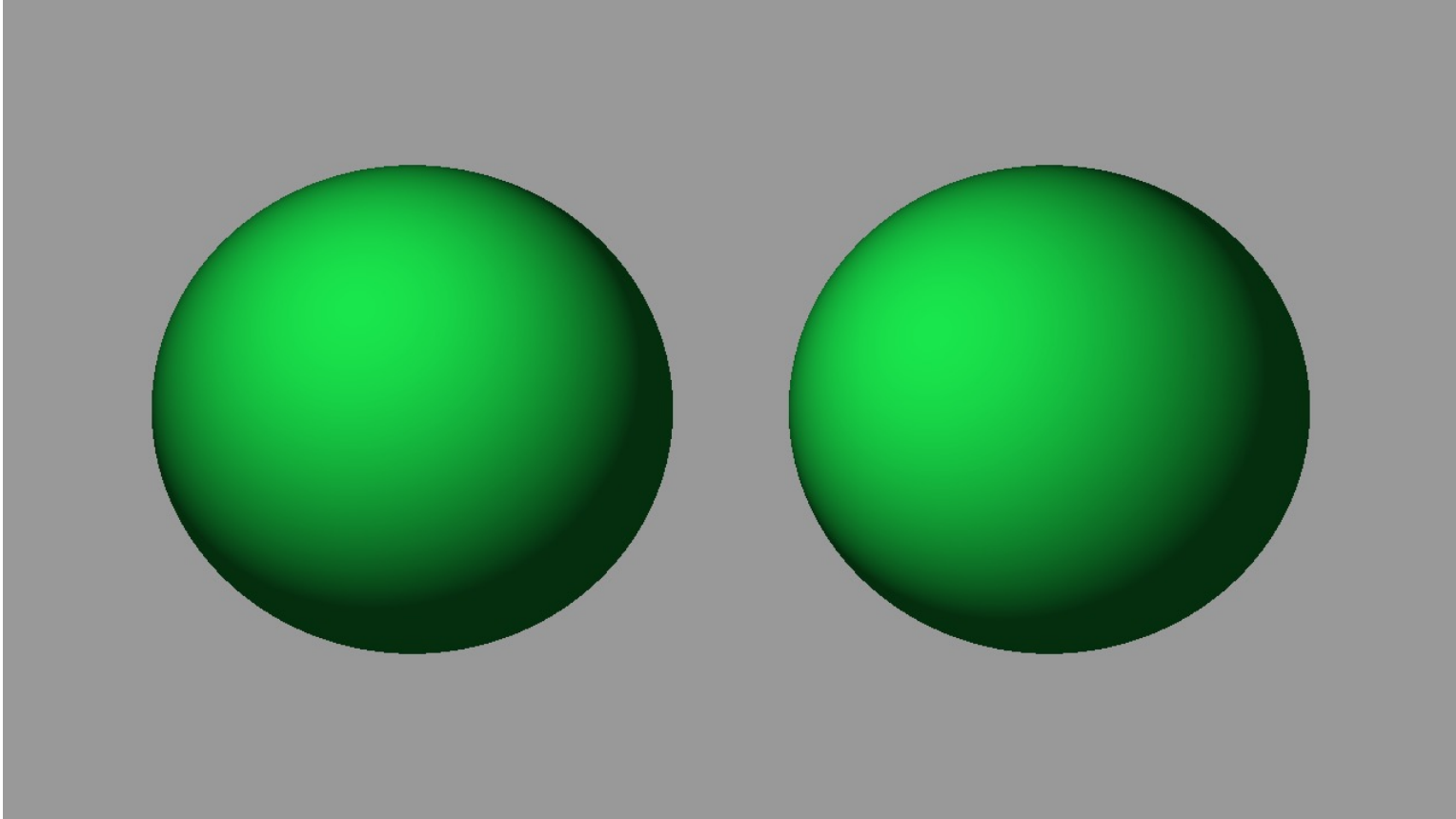
# Múltiplos objetos

```
...
float sdScene(vec3 p) {
    float sphereLeft = sdSphere(p - vec3(-2.5, 0, -2), 2.0);
    float sphereRight = sdSphere(p - vec3(2.5, 0, -2), 2.0);
    return min(sphereLeft, sphereRight);
}

float rayMarch(vec3 ro, vec3 rd, float start, float end) {
    float depth = start;
    for (int i = 0; i < 255; i++) {
        vec3 p = ro + depth * rd;
        float d = sdScene(p);
        depth += d;
        if (d < 0.001 || depth > end) break;
    }
    return depth;
}

vec3 calcNormal(vec3 p) {
    vec2 e = vec2(1.0, -1.0) * 0.0005; // epsilon
    return normalize(
        e.xyy * sdScene(p + e.xyy) +
        e.yyx * sdScene(p + e.yyx) +
        e.yxy * sdScene(p + e.yxy) +
        e.xxx * sdScene(p + e.xxx));
}
...
```

# Resultado do gerenciamento de objetos



# Selecionando cores

Não queremos que todos os objetos fiquem com as mesmas cores. Assim vamos criar uma forma de armazenar as cores.

Para essa implementação vamos armazenar a distância no quarto valor de um `vec4`, deixando os 3 primeiros para as cores.

Existem muitas outras formas de realizar esse processo, por exemplo com estruturas de dados tradicionais.



# Cores (Parte 1)

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

vec4 minWithColor(vec4 obj1, vec4 obj2) {
    if (obj2.a < obj1.a) return obj2;
    return obj1;
}

vec4 sdScene(vec3 p) {
    vec4 sphereLeft = vec4(vec3(0.1, 0.9, 0.3), sdSphere(p - vec3(-2.5, 0, -2), 2.0));
    vec4 sphereRight = vec4(vec3(0.1, 0.3, 0.9), sdSphere(p - vec3(2.5, 0, -2), 2.0));
    vec4 co = minWithColor(sphereLeft, sphereRight);
    return co;
}

vec4 rayMarch(vec3 ro, vec3 rd, float start, float end) {
    float depth = start;
    vec4 co;
    for (int i = 0; i < 255; i++) {
        vec3 p = ro + depth * rd;
        co = sdScene(p);
        depth += co.a;
        if (co.a < 0.001 || depth > end) break;
    }
    return vec4(co.rgb, depth);
}
```

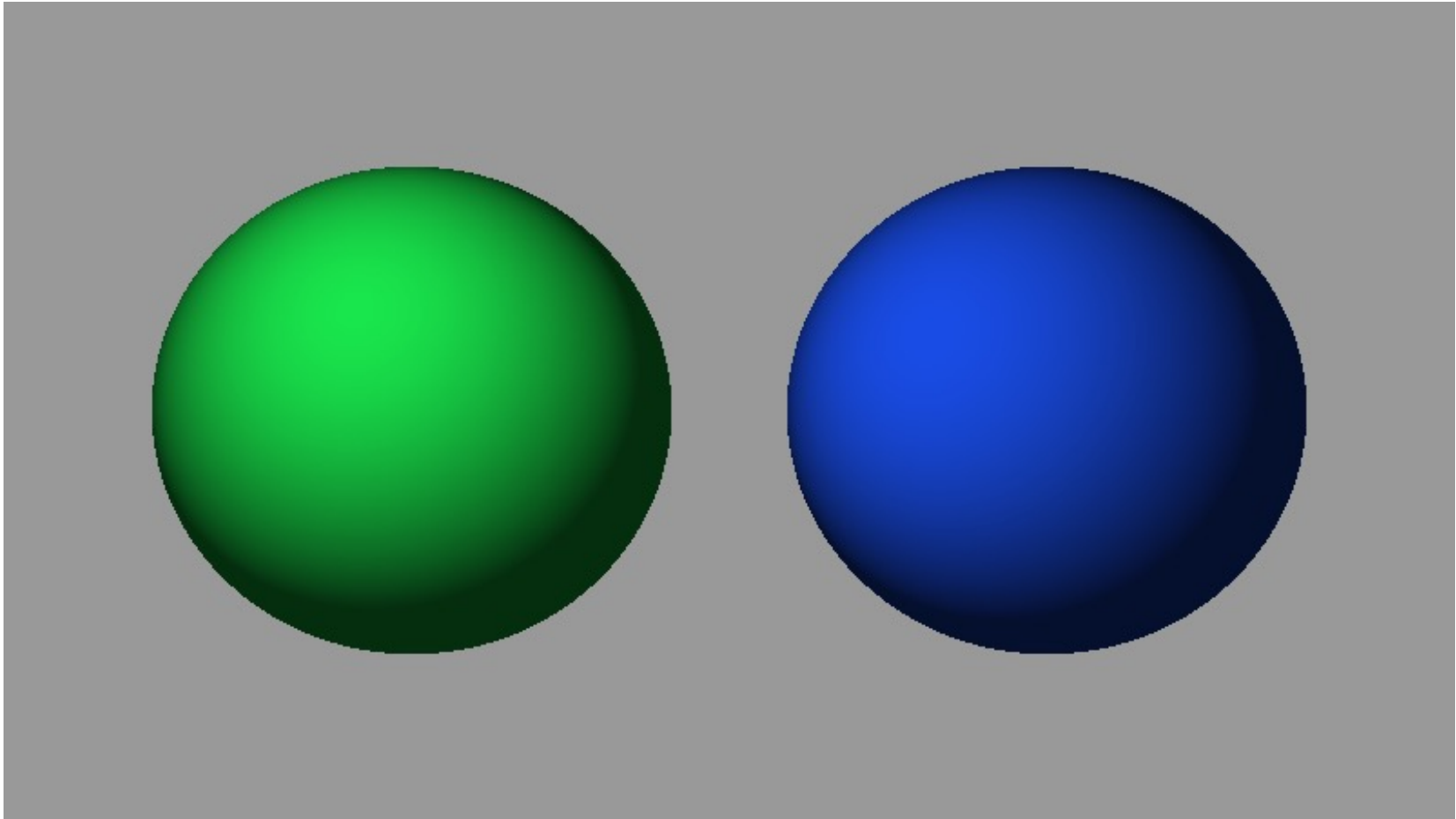
# Cores (Parte 2)

```
vec3 calcNormal(vec3 p) {
    vec2 e = vec2(1.0, -1.0) * 0.0005; // epsilon
    return normalize(
        e.xyy * sdScene(p + e.xyy).a +
        e.yyx * sdScene(p + e.yyx).a +
        e.yxy * sdScene(p + e.yxy).a +
        e.xxx * sdScene(p + e.xxx).a);
}

void mainImage( out vec4 fragColor, in vec2 fragCoord ) {
    vec2 uv = (fragCoord-.5*iResolution.xy)/iResolution.y;
    vec3 col = vec3(0);
    vec3 ro = vec3(0, 0, 5);
    vec3 rd = normalize(vec3(uv, -1));
    vec4 co = rayMarch(ro, rd, 0.01, 100.0);

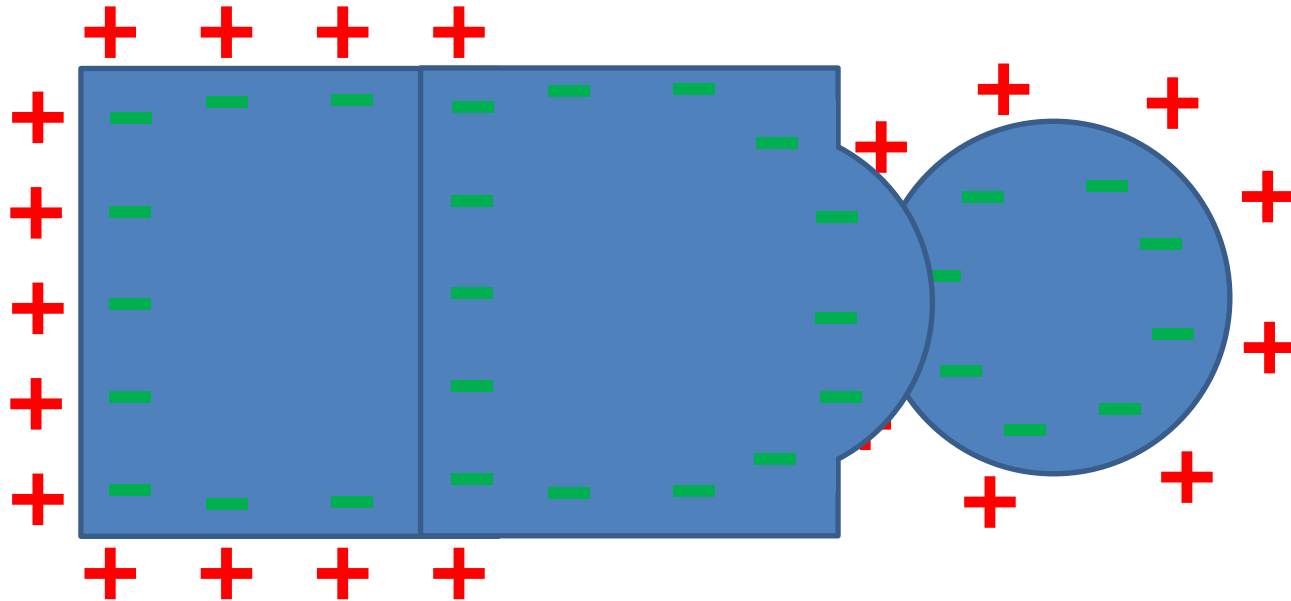
    if (co.a > 100.0) col = vec3(0.6);
    else {
        vec3 p = ro + rd * co.a;
        vec3 normal = calcNormal(p);
        vec3 lightPos = vec3(-2, 2, 4);
        vec3 lightDir = normalize(lightPos - p);
        float ambient = 0.2;
        float difuse = clamp(dot(normal, lightDir), ambient, 1.);
        col = difuse * co.rgb;
    }
    fragColor = vec4(col, 1.0);
}
```

# Resultado das Cores



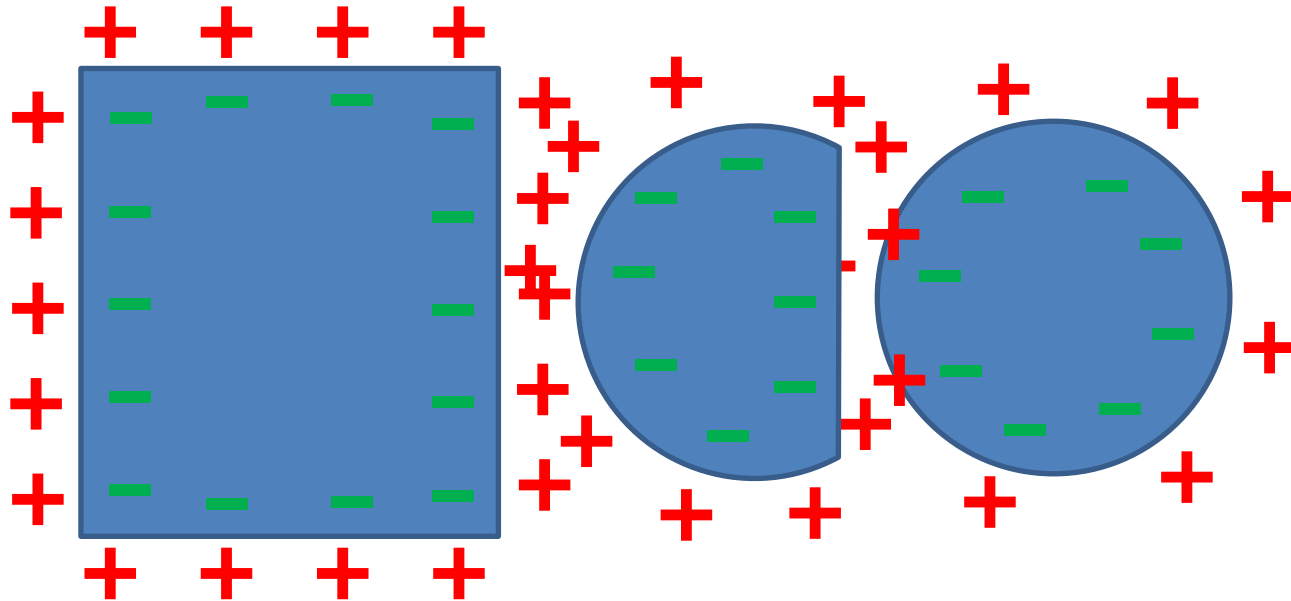
# Operações de União, Intersecção e Diferença

A operação mais simples que temos é a **união**. Para isso imagine que se uma das funções de distância retorna um número negativo, vamos ficar com ele, assim uma das estratégias é usar a função `min()`.



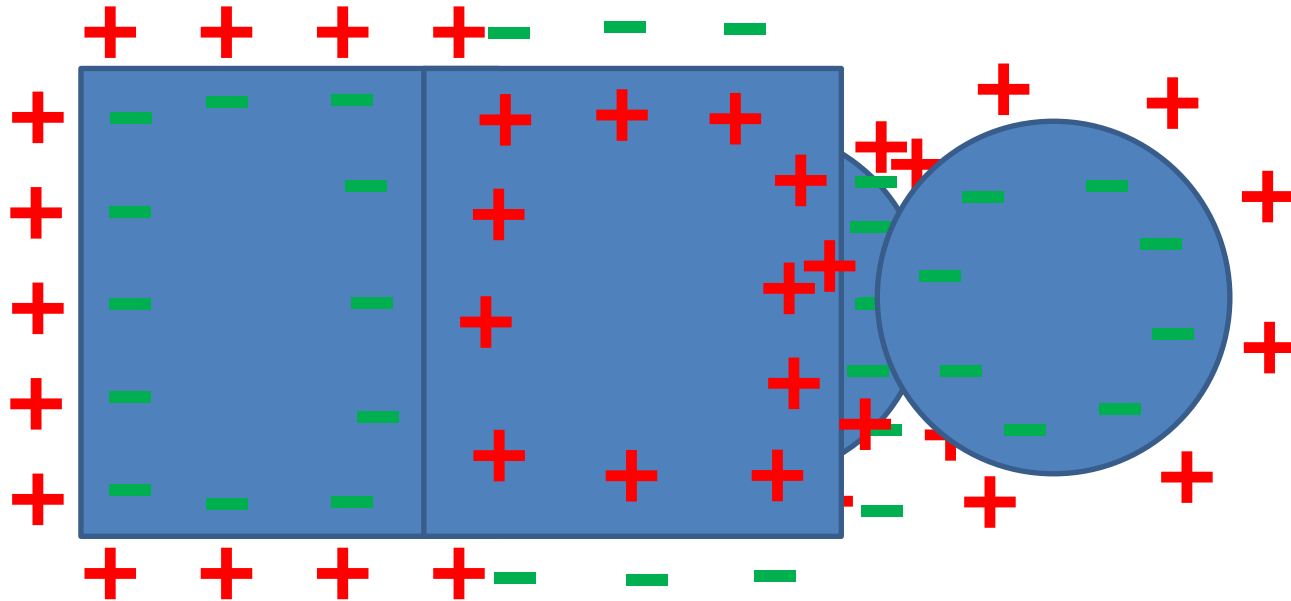
# Operações de União, Intersecção e Diferença

Outra operação que temos é a intersecção. Para isso imagine que só queremos se ambas as funções retornarem negativos. Só nessa região estaremos dentro das duas regiões. Para isso usamos uma função `max()`.



# Operações de União, Intersecção e Diferença

Para a operação de intersecção, imagine que estamos invertendo uma geometria (o que esta dentro fica fora e vice versa) e depois fazemos a intersecção. Para isso negamos o resultado de uma função e depois fazemos a intersecção com o `max()`.



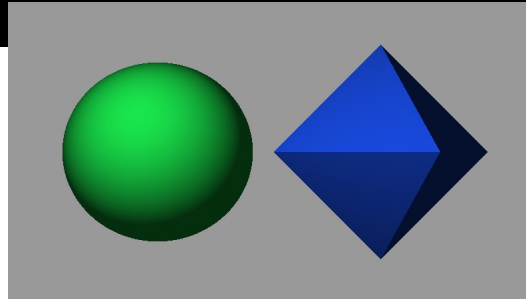
# Operações de União, Intersecção e Diferença

Da mesma forma que fazíamos operações entre duas geometrias em 2D, podemos fazer em 3D.

Vamos criar um octaedro para fazer operações com a esfera.

```
float sdOctahedron( vec3 p, float s){
    p = abs(p);
    return (p.x+p.y+p.z-s)*0.57735027;
}

vec4 sdScene(vec3 p) {
    vec4 sphere = vec4(vec3(0.1, 0.9, 0.3), sdSphere(p - vec3(-2.5, 0, -2), 2.0));
    vec4 octahedron = vec4(vec3(0.1, 0.3, 0.9), sdOctahedron(p - vec3(2.5, 0, -2), 2.5));
    vec4 co = minWithColor(sphere, octahedron);
    return co;
}
...
```



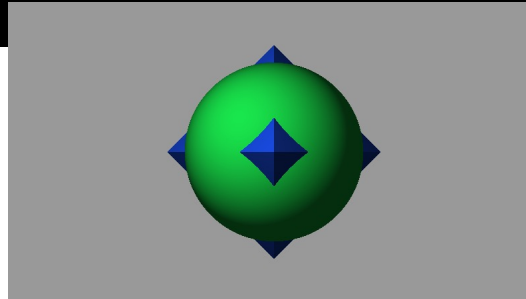
# União

Já estávamos fazendo o mínimo. Operação simples.

“Existem operações de mistura (blend) interessantes, fica de lição de casa verificar essas operações.”

```
float sdOctahedron( vec3 p, float s){
    p = abs(p);
    return (p.x+p.y+p.z-s)*0.57735027;
}

vec4 sdScene(vec3 p) {
    vec4 sphere = vec4(vec3(0.1, 0.9, 0.3), sdSphere(p - vec3(0, 0, -2), 2.0));
    vec4 octahedron = vec4(vec3(0.1, 0.3, 0.9), sdOctahedron(p - vec3(0, 0, -2), 2.5));
    vec4 co = minWithColor(sphere, octahedron);
    return co;
}
...
```

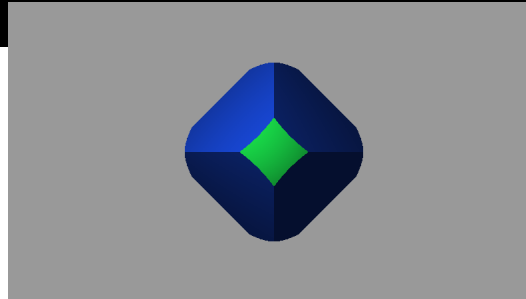




# Intersecção

Vamos agora usar uma função max().

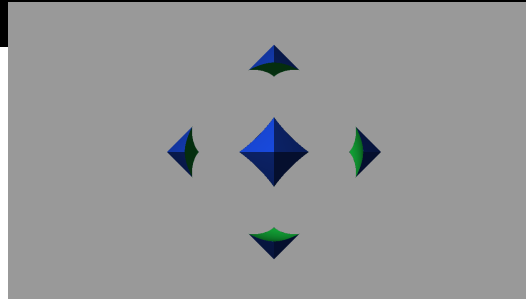
```
vec4 maxWithColor(vec4 obj1, vec4 obj2) {  
    if (obj2.a > obj1.a) return obj2;  
    return obj1;  
}  
  
vec4 sdScene(vec3 p) {  
    vec4 sphere = vec4(vec3(0.1, 0.9, 0.3), sdSphere(p - vec3(0, 0, -2), 2.0));  
    vec4 octahedron = vec4(vec3(0.1, 0.3, 0.9), sdOctahedron(p - vec3(0, 0, -2), 2.5));  
    vec4 co = maxWithColor(sphere, octahedron);  
    return co;  
}  
...
```



# Diferença

Vamos agora inverter uma função e usar uma função max().

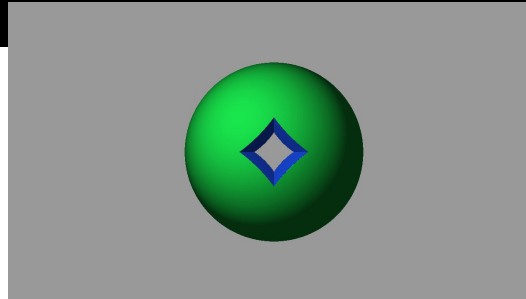
```
vec4 maxWithColor(vec4 obj1, vec4 obj2) {  
    if (obj2.a > obj1.a) return obj2;  
    return obj1;  
}  
  
vec4 sdScene(vec3 p) {  
    vec4 sphere = vec4(vec3(0.1, 0.9, 0.3), sdSphere(p - vec3(0, 0, -2), 2.0));  
    vec4 octahedron = vec4(vec3(0.1, 0.3, 0.9), sdOctahedron(p - vec3(0, 0, -2), 2.5));  
    vec4 co = maxWithColor(sphere*vec3(1,1,1,-1), octahedron);  
    return co;  
}  
...
```



# Diferença

Ou ainda

```
vec4 maxWithColor(vec4 obj1, vec4 obj2) {  
    if (obj2.a > obj1.a) return obj2;  
    return obj1;  
}  
  
vec4 sdScene(vec3 p) {  
    vec4 sphere = vec4(vec3(0.1, 0.9, 0.3), sdSphere(p - vec3(0, 0, -2), 2.0));  
    vec4 octahedron = vec4(vec3(0.1, 0.3, 0.9), sdOctahedron(p - vec3(0, 0, -2), 2.5));  
    vec4 co = maxWithColor(sphere, octahedron*vec3(1,1,1,-1));  
    return co;  
}  
...
```



# Transformações

Vamos agora trabalhar mais nas transformações.

A proposta é alterar o objeto com a transformação inversa do que desejamos.

# 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:

```
sdSphere(p + vec3(-2.0, 0.0, 0.0))
```

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

```
sdSphere(p - vec3(2.0, 0.0, 0.0))
```

# Rotação

Para a rotação podemos multiplicar uma matriz de rotação no ponto. Aplique o inverso da rotação desejada

```
vec4 sdScene(vec3 p) {  
    float theta = 0.2;  
    mat3 rot = mat3(  
        cos(theta) , sin(theta), 0.0, // primeira coluna  
        -sin(theta), cos(theta), 0.0, // segunda coluna  
        0.0          , 0.0          , 1.0 // terceira coluna  
    );  
    vec4 co = vec4(vec3(0.1, 0.3, 0.9), sdOctahedron(rot * p, 2.0));  
    return co;  
}
```

# Coordenadas Homogêneas

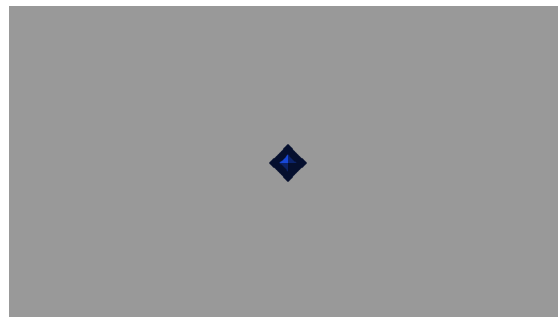
Sim, você pode fazer essas mesmas operações com as coordenadas homogêneas. Isso pode simplificar a lógica do programa.

$$\begin{bmatrix} x'_h \\ y'_h \\ z'_h \\ h' \end{bmatrix} = \begin{bmatrix} M_{11} & M_{12} & M_{13} & M_{14} \\ M_{21} & M_{22} & M_{23} & M_{24} \\ M_{31} & M_{32} & M_{33} & M_{34} \\ M_{41} & M_{42} & M_{43} & M_{44} \end{bmatrix} \cdot \begin{bmatrix} x_h \\ y_h \\ z_h \\ h \end{bmatrix}$$

# Escala

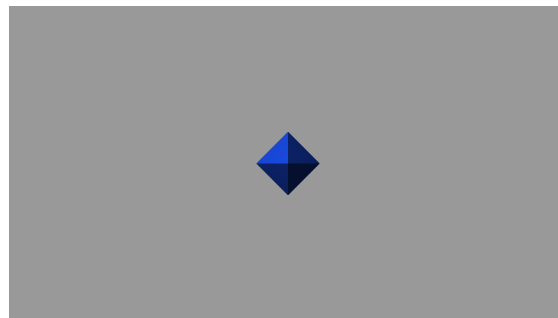
Escala é um problema. A lógica diz para multiplicar pelo inverso da escala. Contudo não vai funcionar direito, pois a escala altera a diferença da função de distância.

```
vec4 sdScene(vec3 p) {  
    return vec4(vec3(0.1, 0.3, 0.9),  
        sdOctahedron(2.0 * p, 1.0)  
    );  
}
```



O truque é depois dividir o resultado pela escala.

```
vec4 sdScene(vec3 p) {  
    return vec4(vec3(0.1, 0.3, 0.9),  
        sdOctahedron(2.0 * p, 1.0)/2.0  
    );  
}
```





# Câmera

Finalmente podemos manipular nossa câmera, reposicionando e rotacionando ela. Para isso vamos usar a matriz de Look At.

```
mat4 look_at(vec3 eye, vec3 at, vec3 up) {
    vec3 w = normalize(at - eye);
    vec3 u = normalize(cross(w, up));
    vec3 v = cross(u, w);
    return mat4(
        vec4(u, 0.0),
        vec4(v, 0.0),
        vec4(-w, 0.0),
        vec4(vec3(0.0), 1.0)
    );
}

void mainImage( out vec4 fragColor, in vec2 fragCoord ) {
    vec2 uv = (fragCoord-.5*iResolution.xy)/iResolution.y;
    vec3 col = vec3(0);
    vec3 ro = vec3(0.0, 0.0, 5.0);
    mat4 view = look_at(ro, vec3(0.0, 0.0, 0.0), vec3(0.0, 1.0, 0.0));
    vec3 rd = normalize(vec3(uv, -1));
    rd = normalize((view * vec4(rd, 1.0)).xyz);

    vec4 co = rayMarch(ro, rd, 0.01, 100.0);
    ...
}
```

# 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>