Insper

# Computação Gráfica

Aula 20: SDF (Signed Distance Function)

# Ray Casting, Ray Marching, Ray Tracing

- **Ray Casting**: Uma forma simples de lançamento de raios. Essa técnica define o um único raio que detecta interseções com objetos. Essa técnica é usada muitas vezes para interações de raios com objetos.
- **Ray Marching**: Um método de projeção de raios que usa SDFs (Signed Distance Fields) e rastreia objetos marchando em direção a eles.
- **Ray Tracing**: uma versão mais sofisticada de ray casting que dispara vários raios, calcula interseções e e cria recursivamente novos raios a cada reflexão.
    - **Path Tracing**: um tipo de algoritmo de rastreamento de raios que dispara diversos raios por pixel em vez de apenas um. Os raios são disparados em direções aleatórias usando o método Monte Carlo.

# Shaders

Linguagens para shaders mais populares são:

- OpenGL Shading Language (GLSL)
- High-Level Shading Language (HLSL)

# Introdução a Shaders (GLSL)

Um típico Shader tem a seguinte estrutura:

```
#version numero_da_versão
in type nome_da_variável_de_entrada;
in type nome_da_variável_de_entrada;
out type nome_da_variável_de_saída;

uniform type nome_do_uniform;

void main() { // processa as entrada(s) e faz algo gráfico
    ...
    // pega as coisas processadas e coloca em variáveis de saída
    out_variable_name = weird_stuff_we_processed;
}
```



Referência: https://learnopengl.com/

# Tipos de Dados (GLSL)

Os tipos básicos de variáveis são:

`int, float, double, uint e bool`

Vetores podem ter 2, 3 ou 4 compomentes:

`vecn: o vetor padrão com n floats.`

`bvecn: um vetor com n booleanos.`

`ivecn: um vetor com n inteiros.`

`uvecn: um vetor com n inteiros sem sinal.`

`dvecn: um vetor com n doubles.`

Você consegue acessar os valores dos vetores com as extensões: .x .y .z .w, ou também com rgba, ou stqp



Referência: https://learnopengl.com/

# Exemplo com Vetores

Um recurso interessante é o **swizzling**, onde você pode combinar e misturar os valores do vetor, por exemplo:

```
vec2 someVec;
vec4 differentVec = someVec.xyxx;
vec3 anotherVec = differentVec.zyw;
vec4 otherVec = someVec.xxxx + anotherVec.yxzy;
```

Na construção de vetores, várias formas de combinação também são viáveis, por exemplo:

```
vec2 vect = vec2(0.5, 0.7);
vec4 result = vec4(vect, 0.0, 0.0);
vec4 otherResult = vec4(result.xyz, 1.0);
```



Referência: https://learnopengl.com/

# Uniforms

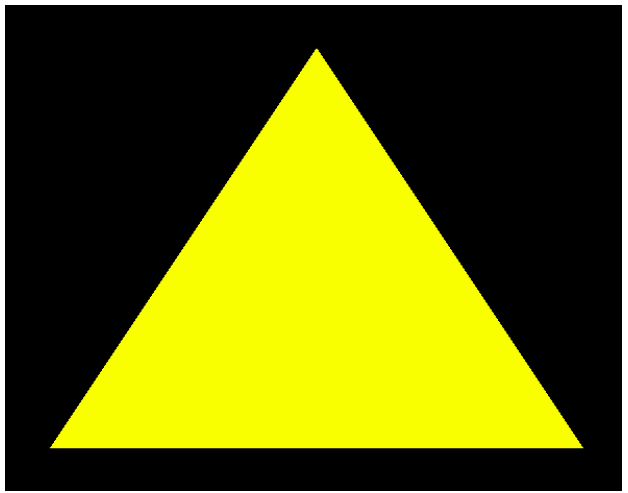
Uniforms são valores globais que podem ser acessados em qualquer shader do pipeline gráfico. Contudo você tem de declarar ele antes de usar.

```
#version 330 core
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 normal;

out vec3 bColor;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

out vec3 bNormal;

void main() {
    // Invertendo a transformação para normal
    bNormal = mat3(transpose(inverse(model))) * normal;

    // Aplicando trasnsformações em cada vértice
    gl_Position = projection * view * model * vec4(position, 1.0);
}
```

# Uniforms

```
...
# Código OpenGL
glUniform1f(uniforms[”color"], [1.0, 1.0, 0.0])
```

```
#version 330 core
uniform vec3 color;
out vec4 FragColor;
void main() {
    FragColor = vec4(color, 1.0);
}
```



Qual seria o resultado?

# In e Outs

Podemos especificar se as variáveis são para a entrada de dados, ou saída de dados. Isso é importante para pode exemplo passar o valor de um shader no pipeline para outro.

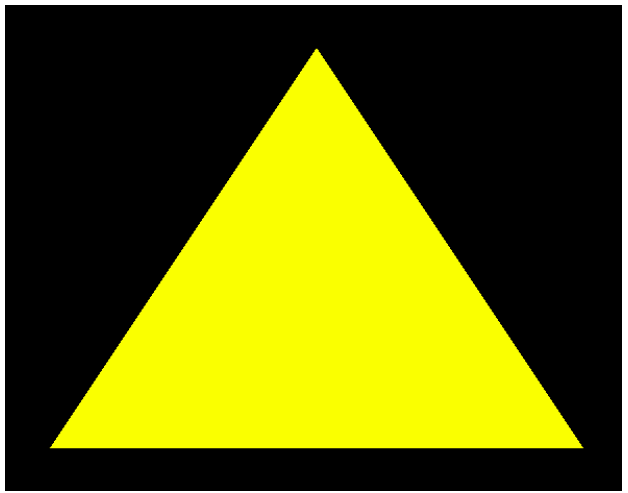
Pode ser usado para receber os dados dos vértices (vertex shader) ou para definir o valor da cor final do pixel (fragmente shader)

```
#version 330 core
in vec4 vertexColor;
out vec4 FragColor;
void main() {
    FragColor = vertexColor;
}
```

# In e Outs

```
#version 330 core
layout (location = 0) in vec3 position;
out vec3 bColor;
void main() {
    gl_Position = vec4(position, 1.0);
    bColor = vec3( 1.0, 1.0, 0.0);
}
```

```
#version 330 core
in vec4 vertexColor;
out vec4 FragColor;
void main() {
    FragColor = vertexColor;
}
```

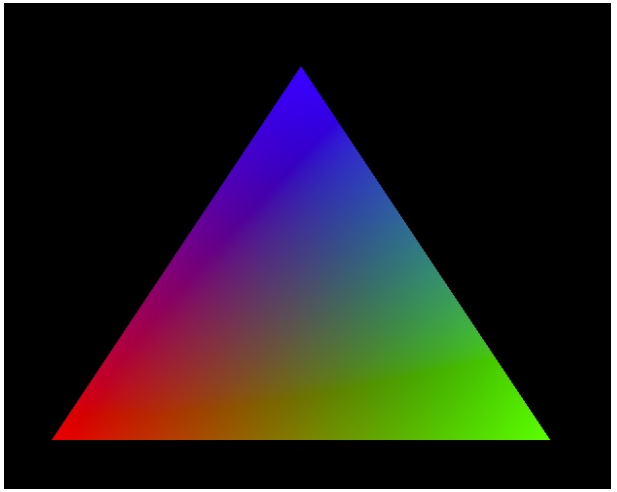


Qual seria o resultado?

# In e Outs

```
#version 330 core
layout (location = 0) in vec3 position;
out vec3 bColor;
void main() {
    gl_Position = vec4(position, 1.0);
    if(gl_VertexID == 0) bColor = vec3(1.0, 0.0, 0.0);
    if(gl_VertexID == 1) bColor = vec3(0.0, 1.0, 0.0);
    if(gl_VertexID == 2) bColor = vec3(0.0, 0.0, 1.0);
}
```

```
#version 330 core
in vec4 vertexColor;
out vec4 FragColor;
void main() {
    FragColor = vertexColor;
}
```



Qual seria o resultado?

# Variáveis Built-in

**Vertex Shader:**

Saída:

- gl_Position (vec4)
- gl_Position (int)

**Fragment Shader:**

Entrada:

- gl_FragCoord (vec4)
- gl_FrontFacing (bool)
- gl_PointCoord (vec2)

Saída:

- gl_FragColor (vec4)

# Funções Built-in

O GLSL possui vária funções nativas, a maioria delas funciona para um valor escalar ou um vetor.

Em geral as funções não funcionam para inteiros ou booleanos, assim pode ser necessário converter os valor.

Por exemplo, a função seno funciona com vários tipos:

**Name**

sin — return the sine of the parameter

**Declaration**

```
genType sin( genType angle);
```

**Parameters**

*angle*

Specify the quantity, in radians, of which to return the sine.

**Description**

**sin** returns the trigonometric sine of *angle*.

https://registry.khronos.org/OpenGL-Refpages/gl4/html/sin.xhtml

**genType** indica que o tipo de dados pode ser float, vec2, vec3 ou vec4.

# Algumas das principais funções

| Função | Descrição |
|---|---|
| genType abs(genType α) | Retorna valor absoluto de α, ou seja, -α se α < 0; |
| genType sign(genType α) | Retorna:<br>-1 para α < 0<br>0 para α = 0<br>1 para α > 0 |
| genType floor(genType α) | Retorna um inteiro menor ou igual a α |
| genType ceil(genType α) | Retorna um inteiro maior ou igual a α |
| genType mod(genType α, float β)<br>genType mod(genType α, genType β) | Retorna o resto da divisão α por β |
| genType min(genType α, float β)<br>genType min(genType α, genType β) | Retorna α quando α < β<br>Retorna β quando β < α |
| genType max(genType α, float β)<br>genType max(genType α, genType β) | Retorna α quando α > β<br>Retorna β quando β > α |
| genType clamp(genType α, float β, float δ)<br>genType clamp(genType α, genType β, genType δ) | Retorna:<br>α quando β < α < δ<br>β quando α > β<br>δ quando α > δ |
| genType mix(genType α, float β, float δ)<br>genType mix(genType α, genType β, genType δ) | Retorna a interpolação linear de α e β, ou seja, α + δ(β - α) |
| genType step(float limit, genType α)<br>genType step(genType limit, genType α) | Retorna 0 quando α < limit<br>Retorna 1 quando α >= limit |
| genType smoothstep(float α0, float α1, genType β)<br>genType smoothstep(genType α0, genType α1, genType β) | Retorna 0 quando β < α0<br>Retorna 1 quando β > α1;<br>Retorna a interpolação de Hermite quando α0 < β < α1 |

https://relativity.net.au/gaming/glsl/Built-inCommonFunctions.html

# Algumas das principais funções

| Função | Descrição |
|---|---|
| genType pow(genType x, genType y) | Retorna valor de x elevado a y |
| float dot(genType x, genType y) | Retorna o produto escalar dos vetores x e y |
| vec3 cross(vec3 x, vec3 y) | Retorna o produto vetorial dos vetores x e y |
| float length(genType x) | Retorna o comprimento (magnitude) do vetor x |
| genType normalize(genType v); | Retorna o vetor v normalizado |

# Shadertoy

O Shadertoy é uma ferramenta da internet que permite escrever Fragment Shaders direto no navegador.

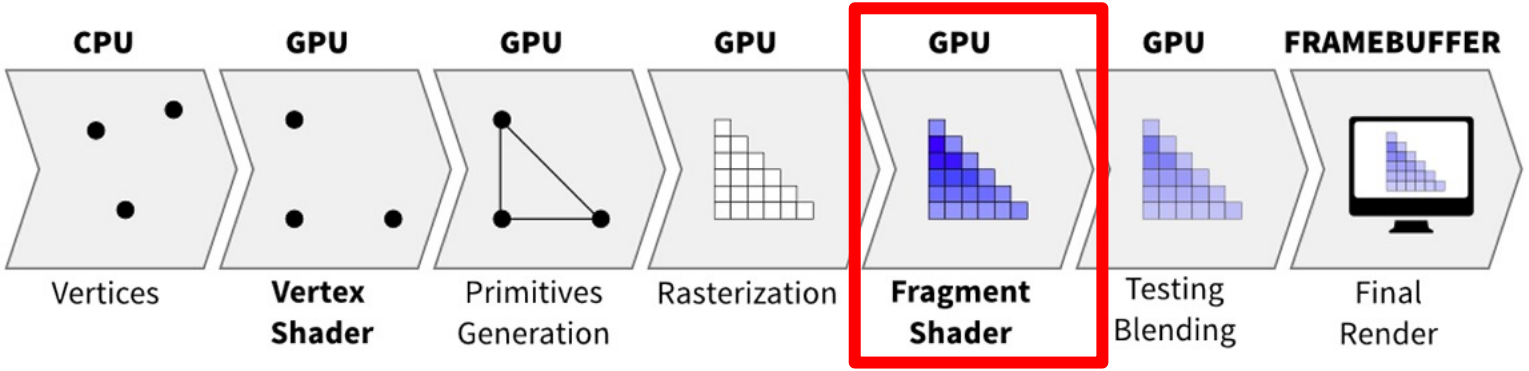
Alguns Uniforms já são automaticamente fornecidos, e todo o processo de compilação é basicamente instantâneo.

O Shadertoy usa alguns padrões para passar os dados, como no caso a chamada do main(), que é mainImage().




Insper

# Fragment Shader




O Shadertoy não permite que você escreva vertex shaders e apenas permite que você escreva fragment shaders. Essencialmente, ele fornece um ambiente para experimentar e desenvolver no fragmento shader, tirando todo o proveito do paralelismo de pixels na tela.



Imagem:F.Andreussi(BUW)

# Shadertoy Uniforms

**uniform vec3 iResolution; // Resolução da Janela**
**uniform float iTime; // Float dos segundos passados**
uniform float iTimeDelta;
uniform float iFrame;
uniform float iChannelTime[4];
uniform vec4 iMouse;
uniform vec4 iDate;
uniform float iSampleRate;
uniform vec3 iChannelResolution[4];
uniform samplerXX iChanneli;

# Shadertoy

Exemplo quando se cria um novo Shader:

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    // Normalized pixel coordinates (from 0 to 1)
    vec2 uv = fragCoord/iResolution.xy;

    // Time varying pixel color
    vec3 col = 0.5 + 0.5*cos(iTime+uv.xyx+vec3(0,2,4));

    // Output to screen
    fragColor = vec4(col,1.0);
}
```

O que isso faz?

# Interpreta tipo de vetores

Quando o shader tem um tipo de chamada como a seguinte:

```
uv = fragCoord/iResolution.xy
```

Ele executa as divisões individualmente internamente:

```
uv.x = fragCoord.x/iResolution.x
uv.y = fragCoord.y/iResolution.y
```

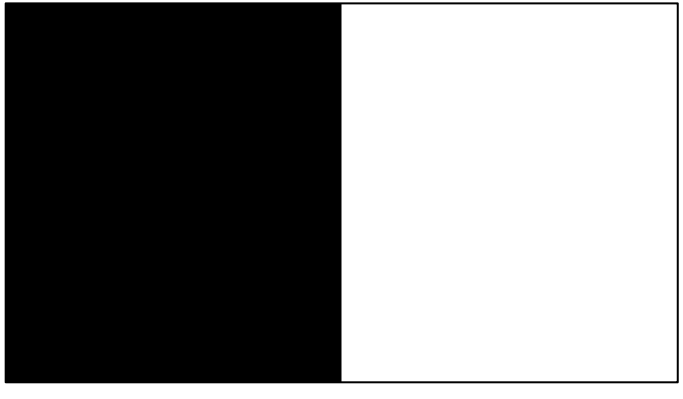
# Função step() do GLSL

```
genType step(genType edge, genType x);
```

Essa função retorna 0 se o valor x for menor que edge, senão retorna 1.

```
void mainImage( out vec4 fragColor, in vec2 fragCoord ){
    vec2 uv = fragCoord/iResolution.xy;
    vec3 col = vec3(step(0.5, uv.x));
    fragColor = vec4(col,1.0);
}
```

O que faz esse código?



# Signed Distance Function (SDF)

Em matemática, a Função de Distância com Sinal (Signed Distance Function) é a distância ortogonal de um determinado ponto x ao limite de um conjunto Ω em um espaço métrico, com o sinal determinado por x estar ou não no interior de Ω.

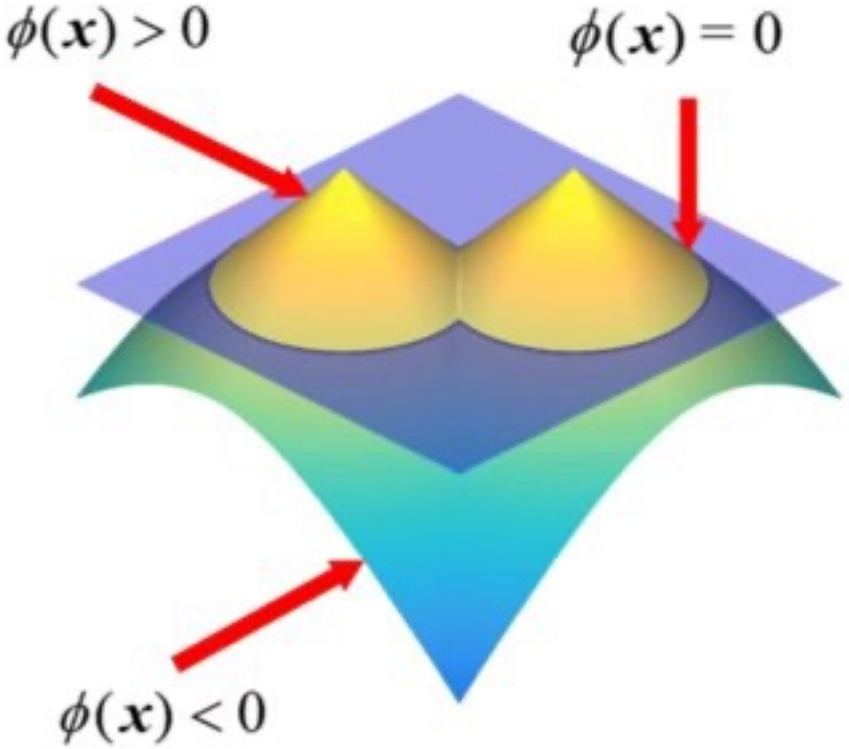
A função tem valores positivos nos pontos x dentro de Ω, diminui em valor quando x se aproxima do limite de Ω onde a função de distância com sinal é zero e assume valores negativos fora de Ω. No entanto, a convenção alternativa às vezes também é adotada (isto é, negativo dentro de Ω e positivo fora).

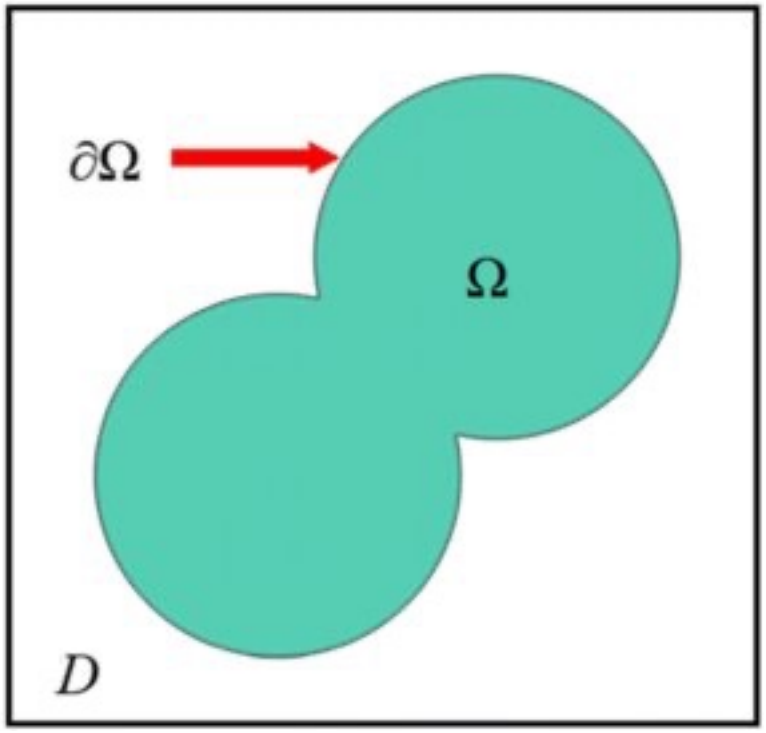


Adaptado de: https://en.wikipedia.org/wiki/Signed_distance_function

# Signed Distance Function (SDF)

Explicar...



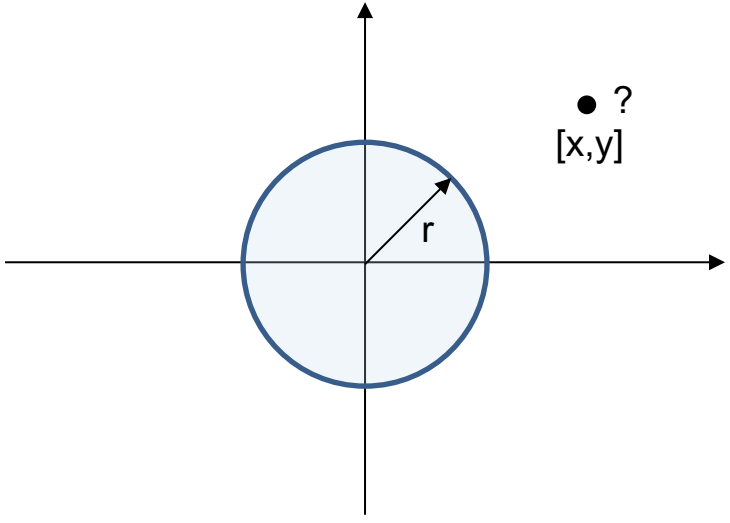



Stress-based shape and topology optimization with cellular level set in B-splines, por :Yelin Song & Qingping Ma & Yu He & Mingdong Zhou & Michael Yu Wang

# Função para círculo 2D

Imagine que estou testando um ponto *uv*.

Como saber se este ponto está dentro de um círculo de raio r?



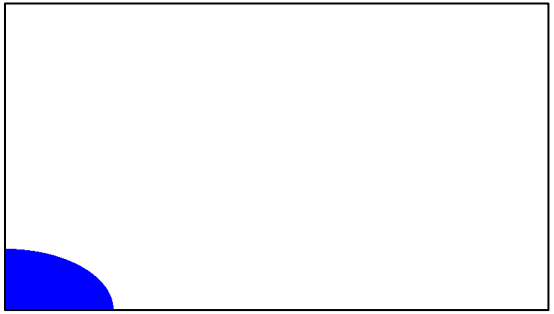

```
float sdfCircle(vec2 uv, float r) {
    float d = length(uv) - r;
    return d;
}
```

# Signed Distance Function (SDF)

Exemplo em GLSL para Shadertoy

```
vec3 sdfCircle(vec2 uv, float r) {
    float d = length(uv) - r;
    return d > 0. ? vec3(1.) : vec3(0., 0., 1.);
}

void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 uv = fragCoord/iResolution.xy;
    vec3 col = sdfCircle(uv, .2);
    fragColor = vec4(col,1.0);
}
```

O que acontece?



Referência: https://inspirnathan.com/posts/47-shadertoy-tutorial-part-1/

# Signed Distance Function (SDF)

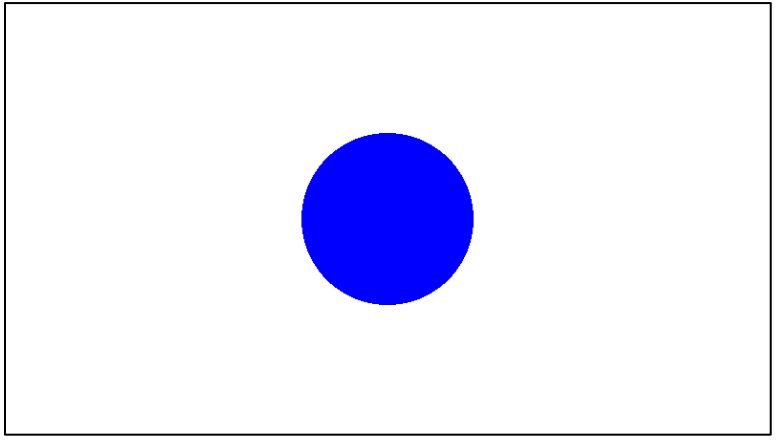
Como melhorar?

Centralizar:

```
uv -= 0.5;
```

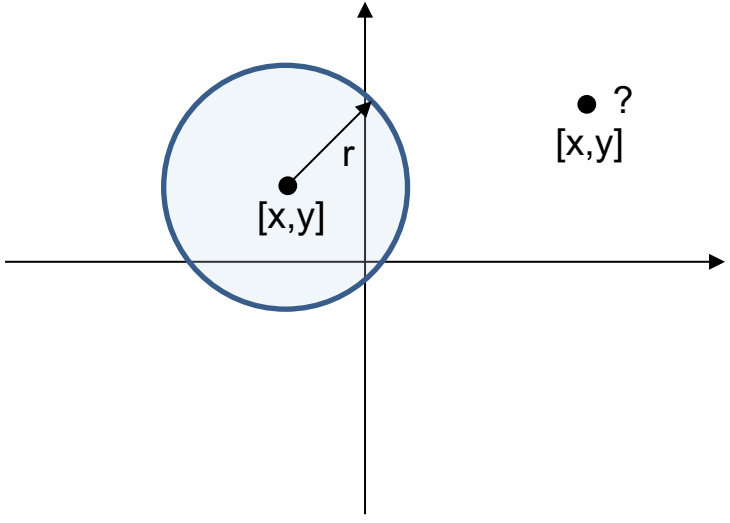
Acertar a razão de aspecto:

```
uv.x *= iResolution.x/iResolution.y;
```



# Círculo em outras posições

Como podemos fazer o círculo aparecer em outra posição?

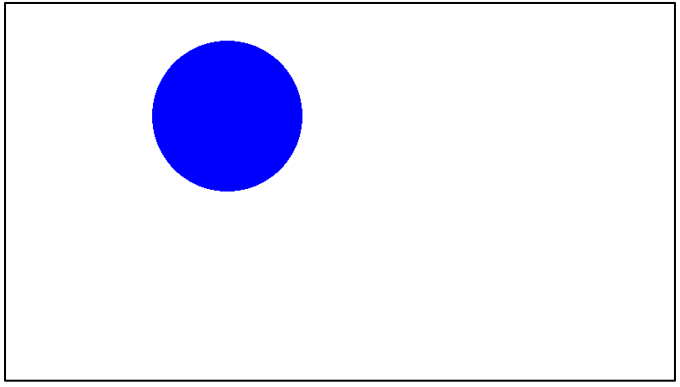



```
float sdfCircle(vec2 uv, float r, vec3 c) {
    float d = length(uv - c) - r;
    return d;
}
```

# Exemplo

Um exemplo com o seguinte ponto:

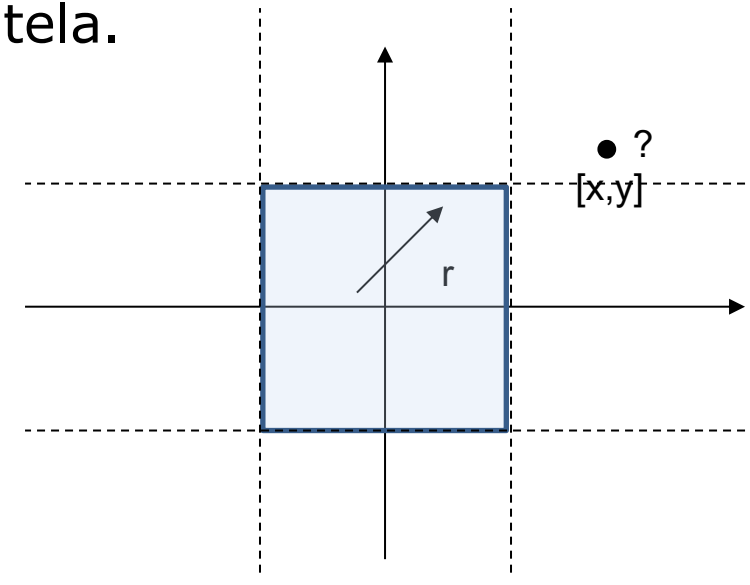
```
vec3 sdfCircle(vec2 uv, float r, vec2 c) {
    float d = length(uv - c) - r;
    return d > 0. ? vec3(1.) : vec3(0., 0., 1.);
}

void mainImage( out vec4 fragColor, in vec2 fragCoord ) {
    vec2 uv = fragCoord/iResolution.xy;
    uv -= 0.5;
    uv.x *= iResolution.x/iResolution.y;
    vec3 col = sdfCircle(uv, .2, vec2(-0.3, 0.2));
    fragColor = vec4(col,1.0);
}
```



Referência: https://inspirnathan.com/posts/47-shadertoy-tutorial-part-1/

# Atividade em Aula: Faça um quadrado

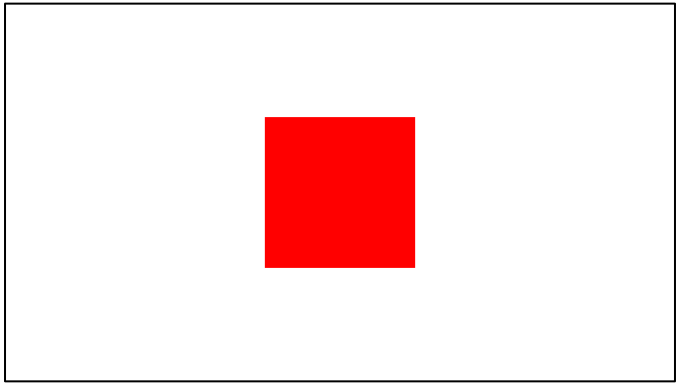
Usando os conceitos aprendidos de SDF. Desenhe um quadrado na tela.




```
bool sdfSquare(vec2 uv, float size, vec2 c) {
    float x = uv.x - c.x;
    float y = uv.y - c.y;
    float d = max(abs(x), abs(y)) - size;
    return d;
}
```

# Exemplo Completo

```
vec3 sdfSquare(vec2 uv, float size, vec2 c) {
    float x = uv.x - c.x;
    float y = uv.y - c.y;
    float d = max(abs(x), abs(y)) - size;
    return d > 0. ? vec3(1.) : vec3(1., 0., 0.);
}


void mainImage( out vec4 fragColor, in vec2 fragCoord ) {
    vec2 uv = fragCoord/iResolution.xy;
    uv -= 0.5;
    uv.x *= iResolution.x/iResolution.y;
    vec3 col = sdfSquare(uv, 0.2, vec2(0.0, 0.0));
    fragColor = vec4(col,1.0);
}
```



Referência: https://inspirnathan.com/posts/47-shadertoy-tutorial-part-1/

# Transformando objetos (Rotação)

Podemos aplicar a matriz de rotação em um objeto.

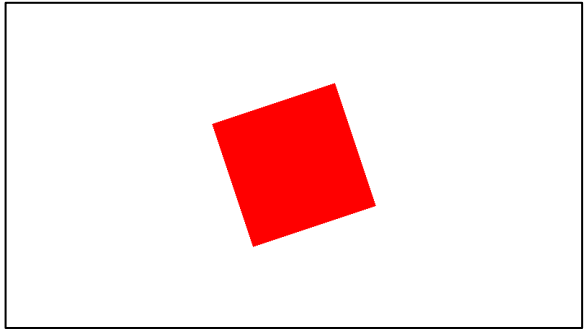
$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

Cuidado que as matrizes no GLSL são *column-first*

```
vec2 rotate(vec2 uv, float th) {
    return mat2(cos(th), sin(th), -sin(th), cos(th)) * uv;
}
```

# Exemplo com rotação animada

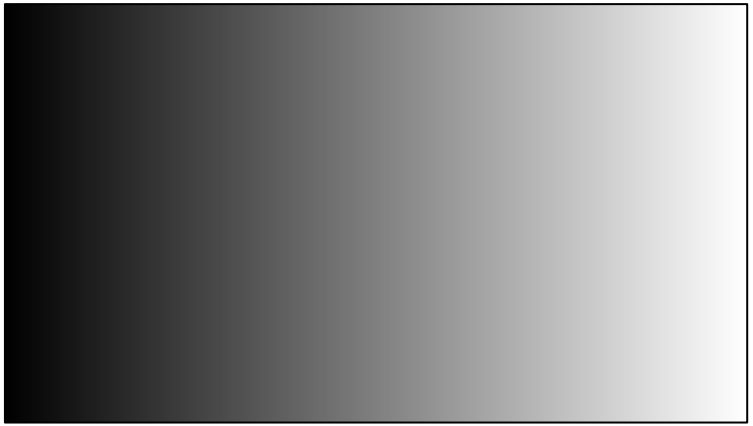
```
vec2 rotate(vec2 uv, float th) {
    return mat2(cos(th), sin(th), -sin(th), cos(th)) * uv;
}

vec3 sdfSquare(vec2 uv, float size, vec2 c) {
    float x = uv.x - c.x;
    float y = uv.y - c.y;
    vec2 rotated = rotate(vec2(x,y), iTime);
    float d = max(abs(rotated.x), abs(rotated.y)) - size;
    return d > 0. ? vec3(1.) : vec3(1., 0., 0.);
}

void mainImage( out vec4 fragColor, in vec2 fragCoord ) {
    vec2 uv = fragCoord/iResolution.xy;
    uv -= 0.5;
    uv.x *= iResolution.x/iResolution.y;
    vec3 col = sdfSquare(uv, 0.2, vec2(0.0, 0.0));
    fragColor = vec4(col,1.0);
}
```



Referência: https://inspirnathan.com/posts/47-shadertoy-tutorial-part-1/

# Exemplo mix (LERP)

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

    float interpolatedValue = mix(0., 1., uv.x);
    vec3 col = vec3(interpolatedValue);

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

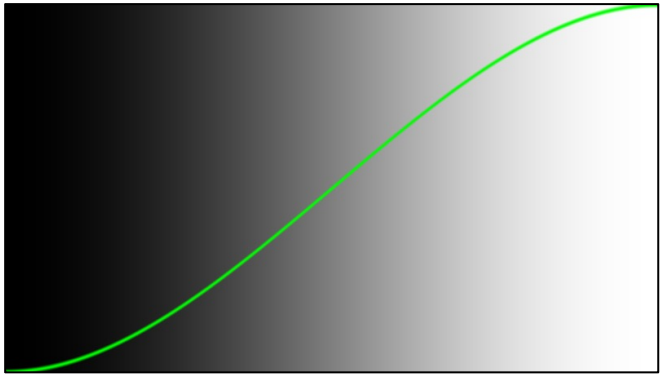




Referência: https://inspirnathan.com/posts/47-shadertoy-tutorial-part-1/

# Exemplo smoothstep (Hermite)

```
#define diameter 0.01

float smooth_step( float edge0, float edge1, float x ) {
    float p = clamp((x - edge0) / (edge1 - edge0), 0.0, 1.0);
    //float v = p * p * (3.0 - 2.0 * p); // smoothstep formula.
    float v = smoothstep( edge0, edge1, x ); // built-in
    return v;
}

float plot(vec2 st, float y)
{
    return smooth_step( y-diameter, y , st.y) -
    smooth_step( y , y+diameter, st.y);
}

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

    float y = smooth_step( 0.0, 1.0, st.x );
    // grey gradient
    vec3 color = vec3(y);

    // draw smoothstep curve in green
    float percent = plot(st,y);
    color = (1.0-percent)*color + percent*vec3(0.0,1.0,0.0);

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



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

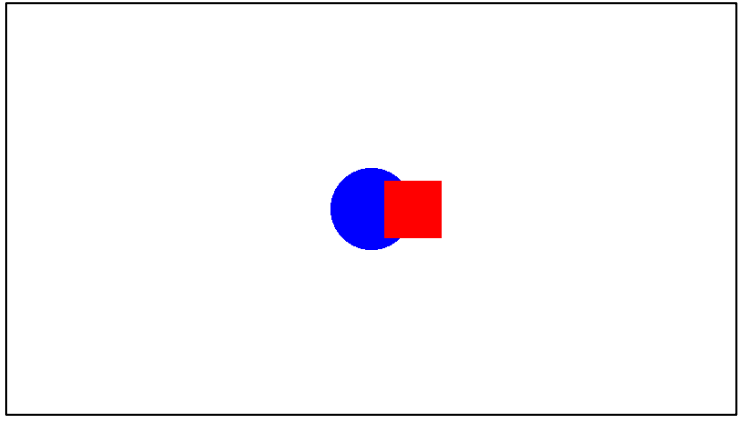
# Atividade: Faça um degrade para fundo de tela

Usando os conceitos aprendidos em aula, faça um degrade

```
void mainImage( out vec4 fragColor, in vec2 fragCoord ) {
    vec2 uv = fragCoord/iResolution.xy;
    vec3 gradientStartColor = vec3(0.6, 0.3, 0.0);
    vec3 gradientEndColor = vec3(0.2, 0.8, 1.);
    vec3 col = mix(gradientStartColor, gradientEndColor, uv.y);
    fragColor = vec4(col,1.0);
}
```

# Organizando código

```
float sdfCircle(vec2 uv, float r, vec2 c) {
  return length(uv - c) - r;
}

float sdfSquare(vec2 uv, float size, vec2 c) {
  float x = uv.x - c.x;
  float y = uv.y - c.y;
  return max(abs(x), abs(y)) - size;
}

vec3 drawScene(vec2 uv) {
  float circle = sdfCircle(uv, 0.1, vec2(0, 0));
  float square = sdfSquare(uv, 0.07, vec2(0.1, 0));

  vec3 col = vec3(1);
  col = mix(vec3(0, 0, 1), col, step(0., circle));
  col = mix(vec3(1, 0, 0), col, step(0., square));

  return col;
}

void mainImage( out vec4 fragColor, in vec2 fragCoord ) {
  vec2 uv = fragCoord/iResolution.xy;
  uv -= 0.5;
  uv.x *= iResolution.x/iResolution.y;

  vec3 col = drawScene(uv);
  fragColor = vec4(col,1.0);
}
```



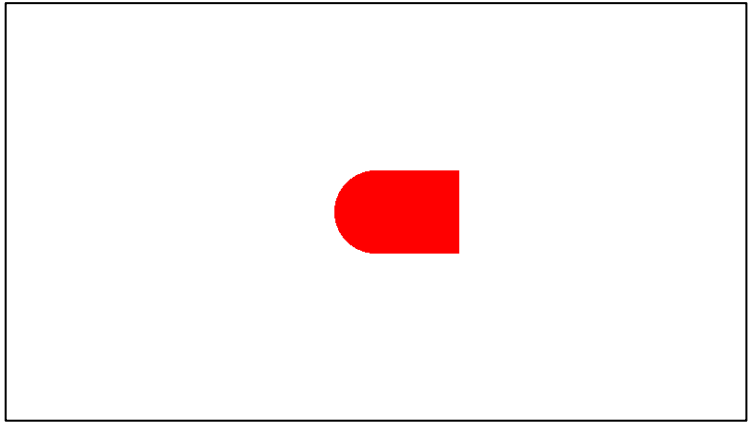
# Combinando formas

Um dos truques interessantes do SDF é poder combinar as formas de diversas formas.

Aqui veremos as principais possibilidades.

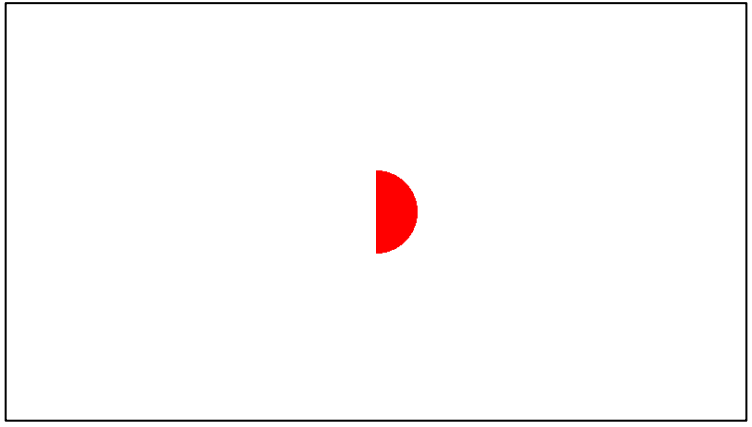
# União

```
float sdfCircle(vec2 uv, float r, vec2 c) {
  return length(uv - c) - r;
}

float sdfSquare(vec2 uv, float size, vec2 c) {
  float x = uv.x - c.x;
  float y = uv.y - c.y;
  return max(abs(x), abs(y)) - size;
}

vec3 drawScene(vec2 uv) {
  float circle = sdfCircle(uv, 0.1, vec2(0, 0));
  float square = sdfSquare(uv, 0.1, vec2(0.1, 0));

  vec3 col = vec3(1);
  float res = min(circle, square);
  col = mix(vec3(1, 0, 0), col, step(0., res));

  return col;
}

void mainImage( out vec4 fragColor, in vec2 fragCoord ) {
  vec2 uv = fragCoord/iResolution.xy;
  uv -= 0.5;
  uv.x *= iResolution.x/iResolution.y;

  vec3 col = drawScene(uv);
  fragColor = vec4(col,1.0);
}
```



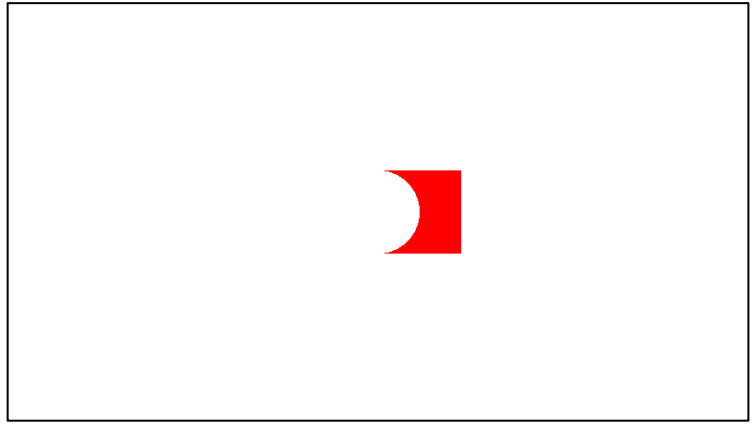
# Intersecção

```
float sdfCircle(vec2 uv, float r, vec2 c) {
  return length(uv - c) - r;
}

float sdfSquare(vec2 uv, float size, vec2 c) {
  float x = uv.x - c.x;
  float y = uv.y - c.y;
  return max(abs(x), abs(y)) - size;
}

vec3 drawScene(vec2 uv) {
  float circle = sdfCircle(uv, 0.1, vec2(0, 0));
  float square = sdfSquare(uv, 0.1, vec2(0.1, 0));

  vec3 col = vec3(1);
  float res = max(circle, square);
  col = mix(vec3(1, 0, 0), col, step(0., res));

  return col;
}

void mainImage( out vec4 fragColor, in vec2 fragCoord ) {
  vec2 uv = fragCoord/iResolution.xy;
  uv -= 0.5;
  uv.x *= iResolution.x/iResolution.y;

  vec3 col = drawScene(uv);
  fragColor = vec4(col,1.0);
}
```



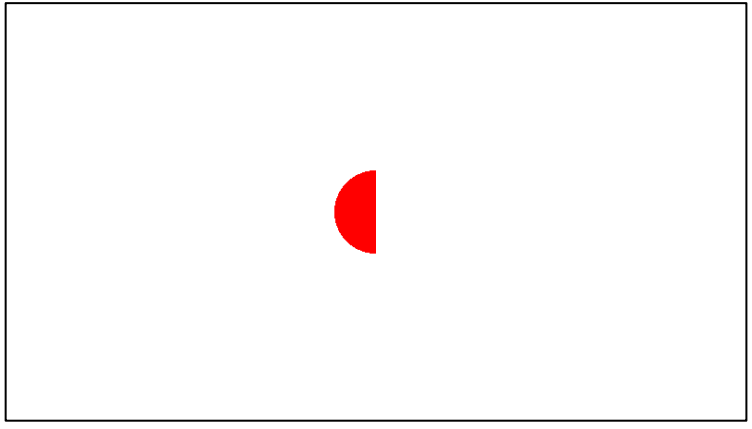
# Subtrair o círculo do quadrado

```
float sdfCircle(vec2 uv, float r, vec2 c) {
  return length(uv - c) - r;
}

float sdfSquare(vec2 uv, float size, vec2 c) {
  float x = uv.x - c.x;
  float y = uv.y - c.y;
  return max(abs(x), abs(y)) - size;
}

vec3 drawScene(vec2 uv) {
  float circle = sdfCircle(uv, 0.1, vec2(0, 0));
  float square = sdfSquare(uv, 0.1, vec2(0.1, 0));

  vec3 col = vec3(1);
  float res = max(-circle, square);
  col = mix(vec3(1, 0, 0), col, step(0., res));

  return col;
}

void mainImage( out vec4 fragColor, in vec2 fragCoord ) {
  vec2 uv = fragCoord/iResolution.xy;
  uv -= 0.5;
  uv.x *= iResolution.x/iResolution.y;

  vec3 col = drawScene(uv);
  fragColor = vec4(col,1.0);
}
```



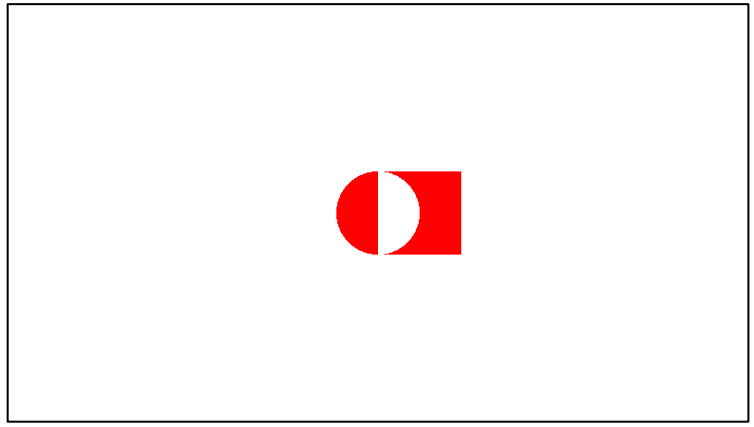
# Subtrair o quadrado do círculo

```
float sdfCircle(vec2 uv, float r, vec2 c) {
  return length(uv - c) - r;
}

float sdfSquare(vec2 uv, float size, vec2 c) {
  float x = uv.x - c.x;
  float y = uv.y - c.y;
  return max(abs(x), abs(y)) - size;
}

vec3 drawScene(vec2 uv) {
  float circle = sdfCircle(uv, 0.1, vec2(0, 0));
  float square = sdfSquare(uv, 0.1, vec2(0.1, 0));

  vec3 col = vec3(1);
  float res = max(circle, -square);
  col = mix(vec3(1, 0, 0), col, step(0., res));

  return col;
}

void mainImage( out vec4 fragColor, in vec2 fragCoord ) {
  vec2 uv = fragCoord/iResolution.xy;
  uv -= 0.5;
  uv.x *= iResolution.x/iResolution.y;

  vec3 col = drawScene(uv);
  fragColor = vec4(col,1.0);
}
```



# Ou exclusivo (XOR)

```
float sdfCircle(vec2 uv, float r, vec2 c) {
  return length(uv - c) - r;
}

float sdfSquare(vec2 uv, float size, vec2 c) {
  float x = uv.x - c.x;
  float y = uv.y - c.y;
  return max(abs(x), abs(y)) - size;
}

vec3 drawScene(vec2 uv) {
  float circle = sdfCircle(uv, 0.1, vec2(0, 0));
  float square = sdfSquare(uv, 0.1, vec2(0.1, 0));

  vec3 col = vec3(1);
  float res = max(min(circle, square), -max(circle, square));
  col = mix(vec3(1, 0, 0), col, step(0., res));

  return col;
}

void mainImage( out vec4 fragColor, in vec2 fragCoord ) {
  vec2 uv = fragCoord/iResolution.xy;
  uv -= 0.5;
  uv.x *= iResolution.x/iResolution.y;

  vec3 col = drawScene(uv);
  fragColor = vec4(col,1.0);
}
```



# Resumindo

```
res = min(d1, d2); // união

res = max(d1, d2); // intersecção

res = max(-d1, d2); // subtração - d1 menos d2

res = max(d1, -d2); // subtração - d2 menos d1

res = max(min(d1, d2), -max(d1, d2)); // xor
```

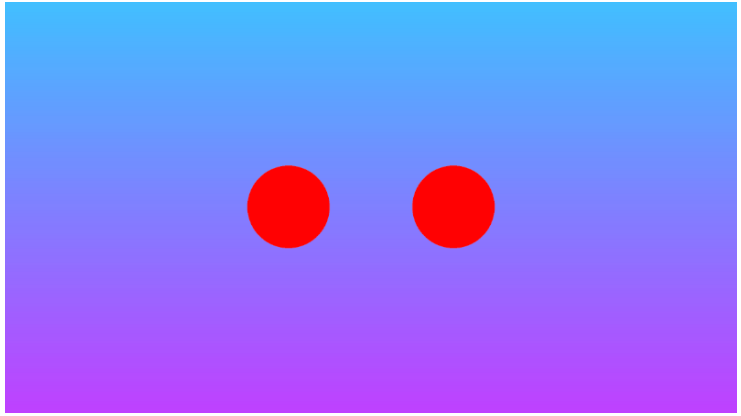
# Posicionamento 2D

Inspirado originalmente no trabalho de Inigo Quilez. A seguir serão apresentadas algumas estratégias de posicionar e exibir padrões de imagens.

# opSymX

Repete o padrão na horizontal.

```
float opSymX(vec2 p, float r)
{
  p.x = abs(p.x);
  return sdCircle(p, r, vec2(0.2, 0));
}

vec3 drawScene(vec2 uv) {
  vec3 col = getBackgroundColor(uv);

  float res; // result
  res = opSymX(uv, 0.1);

  res = step(0., res);
  col = mix(vec3(1,0,0), col, res);
  return col;
}
```



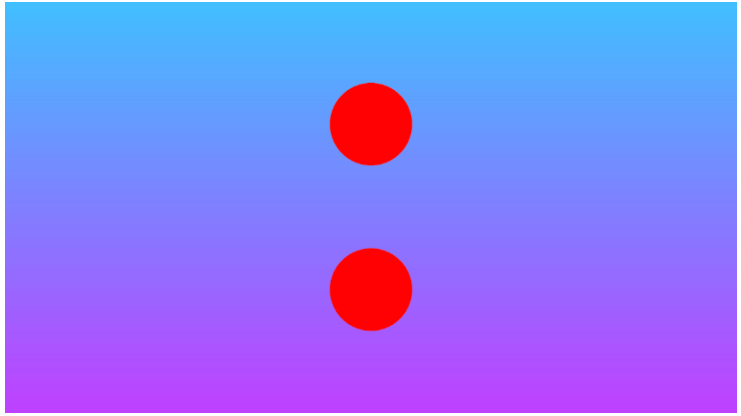


Referência: https://inspirnathan.com/posts/47-shadertoy-tutorial-part-1/

# opSymY

Repete o padrão na horizontal.

```
float opSymY(vec2 p, float r)
{
  p.y = abs(p.y);
  return sdCircle(p, r, vec2(0, 0.2));
}

vec3 drawScene(vec2 uv) {
  vec3 col = getBackgroundColor(uv);

  float res; // result
  res = opSymY(uv, 0.1);

  res = step(0., res);
  col = mix(vec3(1,0,0), col, res);
  return col;
}
```



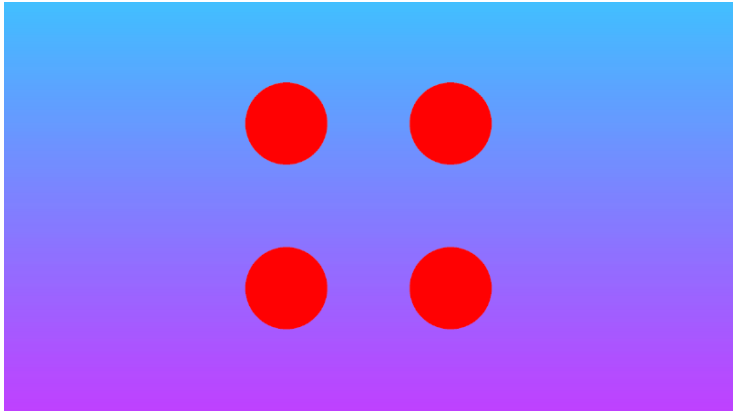


Referência: https://inspirnathan.com/posts/47-shadertoy-tutorial-part-1/

# opSymXY

Repete o padrão na horizontal.

```
float opSymXY(vec2 p, float r)
{
  p = abs(p);
  return sdCircle(p, r, vec2(0.2));
}

vec3 drawScene(vec2 uv) {
  vec3 col = getBackgroundColor(uv);

  float res; // result
  res = opSymXY(uv, 0.1);

  res = step(0., res);
  col = mix(vec3(1,0,0), col, res);
  return col;
}
```



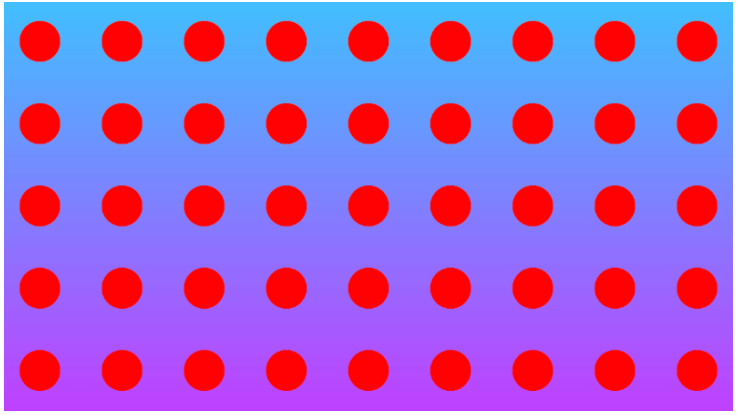


Referência: https://inspirnathan.com/posts/47-shadertoy-tutorial-part-1/

# opRep

Repete o padrão na horizontal.

```
float opRep(vec2 p, float r, vec2 c)
{
  vec2 q = mod(p+0.5*c,c)-0.5*c;
  return sdCircle(q, r, vec2(0));
}

vec3 drawScene(vec2 uv) {
  vec3 col = getBackgroundColor(uv);

  float res; // result
  res = opRep(uv, 0.05, vec2(0.2, 0.2));

  res = step(0., res);
  col = mix(vec3(1,0,0), col, res);
  return col;
}
```



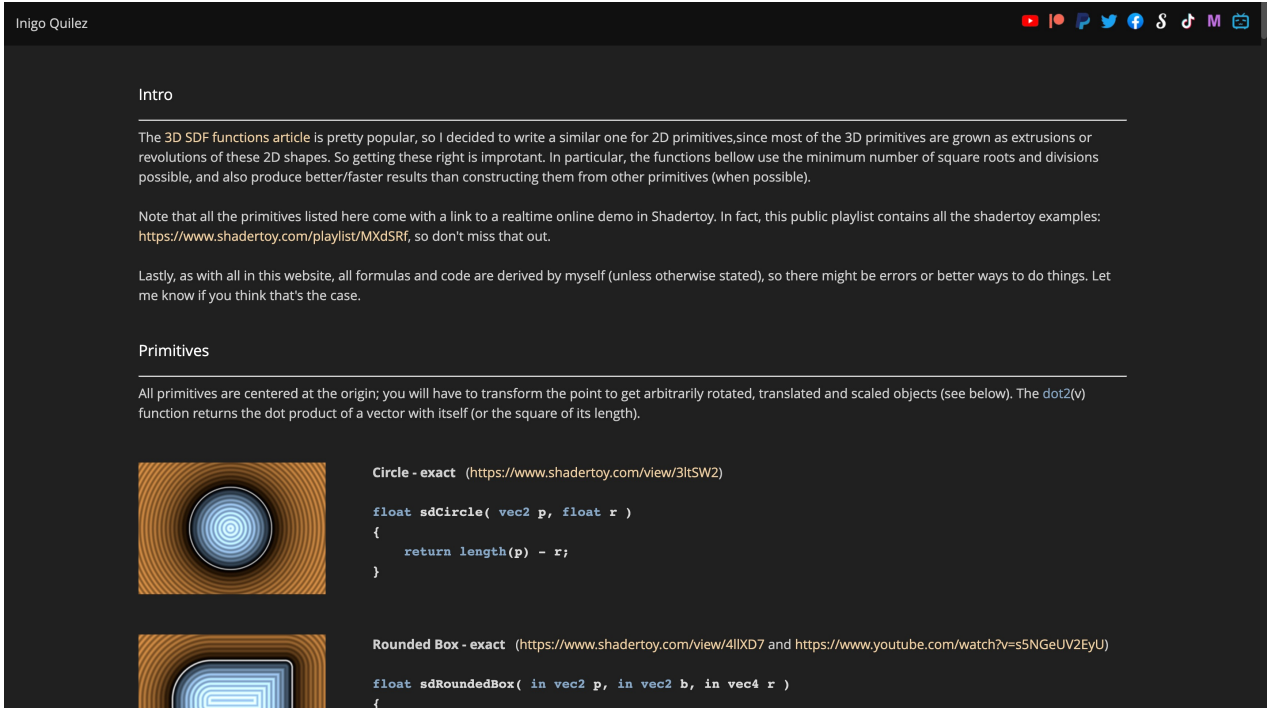


Referência: https://inspirnathan.com/posts/47-shadertoy-tutorial-part-1/

# Funções SDF prontas

Muitas funcionalidades para SDF já existem. Um bom repositório é o site do Inigo Quilz:

https://iquilezles.org/articles/distfunctions2d/

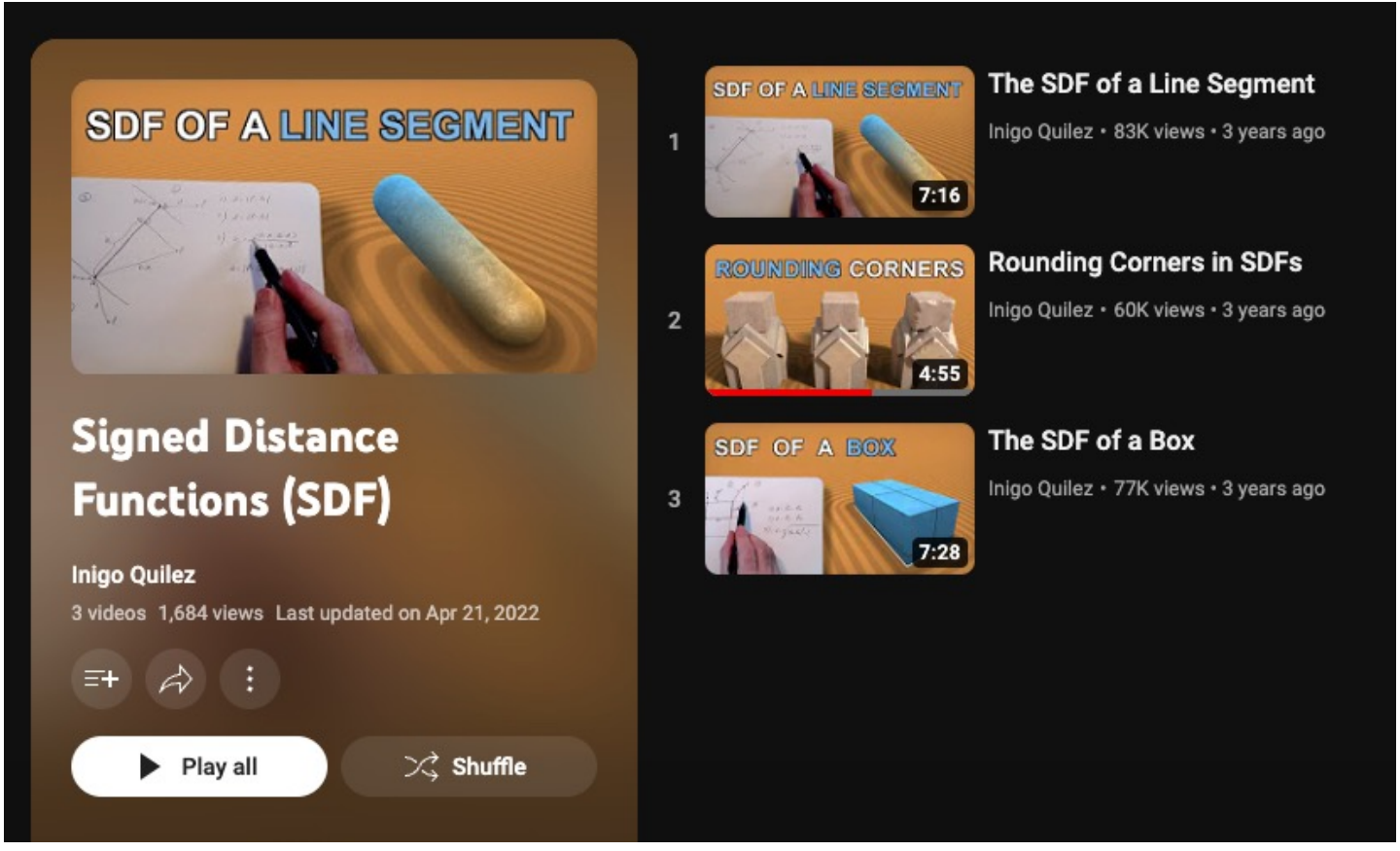





Insper

# Vídeos sobre SDFs

https://www.youtube.com/playlist?list=PL0EpikNmjs2AUFqRi3vmpkrO3j-zWuoyq

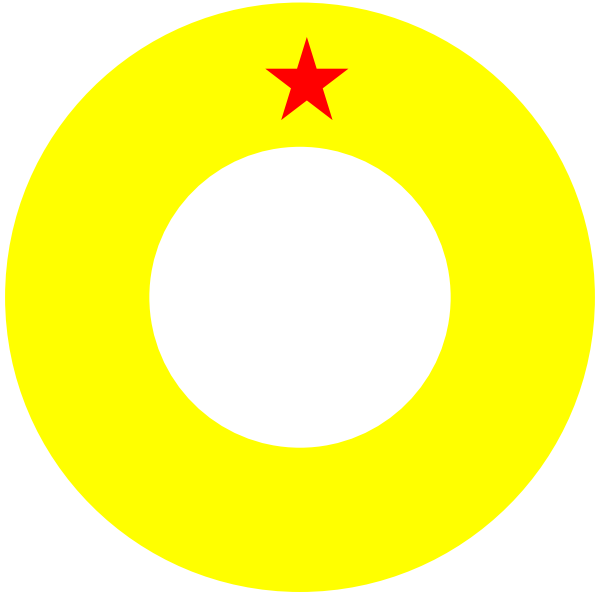





Insper

# Projeto 2.1

Crie uma animação 2D no Fragment Shader de uma estrela (qualquer tipo) sobre um anel. A volta toda deve demorar 5 segundos. A velocidade na parte superior deve ser zero.



# Referências

Baseado:
https://www.shadertoy.com/

Usando:
https://inspirnathan.com/posts/49-shadertoy-tutorial-part-3

Documentações:
https://iquilezles.org/
https://thebookofshaders.com/

Insper

# Computação Gráfica

Luciano Soares
<lpsoares@insper.edu.br>