Linguagem SDL - Signed Distance Language

Aluno: André Corrêa

Professor: Raul Ikeda

Insper - Instituto de Ensino e Pesquisa

andrecs11@al.insper.edu.br

05/06/202

Overview

Introdução

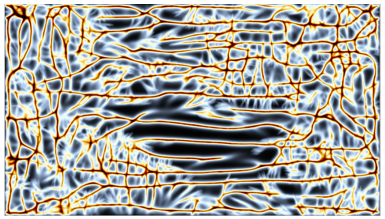
- SDL é inspirada em *"shading languages"*, linguagens nas quais a execução dos programas é inteiramente paralela e a saída de cada execução do programa normalmente é armazenada em buffers que compõem, de alguma forma, a imagem final apresentada na tela.
- Por limitações de tempo e capacidade, não foi possível compilar o código para SPIR-V de modo a executar o binário na placa de vídeo. Dessa forma a execução de programas na linguagem é bastante lenta quanto comparada à execução desses mesmos programas em outras linguagens como glsl que são passíveis de execução na gpu.

Motivação para criar a linguagem

- **Abstração de Overhead**: SDL surge da vontade de eliminar a barreira de ter que escrever dezenas de linhas para poder rodar o primeiro shader. Normalmente, é necessário já possuír um renderizador em Opengl ou vulkan (mais overhead ainda) para poder carregar qualquer shader já compilado para placa de vídeo e sincronizar a renderização e apresentação dos framebuffers.

- **Exemplos**: Sites como shadertoy e compute toys já permitem escrever e visualizar a saída de shaders (fragment shaders e compute shaders respectivamente) em navegadores web, entretanto, em ambos os casos não é possível injetar dados de cena como malhas de polígonos, por exemplo, de forma a obter uma forma concreta e definida:

(a) Imagem renderizada a partir de malha de polígonos. ()fonte



(b) Imagem renderizada com algoritmo bio-inspirado. (fonte

)

Solução

- **Gambiarra**: Para renderizar formas definidas e determinísticas mesmo sem input de objetos 3d, artistas técnicos utilizam de "campos de distância com sinal", que são uma forma de expressar um objeto tridimensional matemáticamente.
- **Signed Distance Fields**: SDFs mapeiam à todo ponto do espaço um valor que representa a distância do determinado ponto à superfície mais próxima e caso o ponto esteja dentro da superfície parametrizada o valor da SDF é negativo.

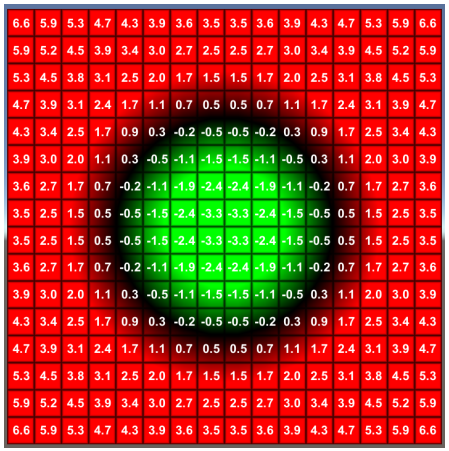


Figure: SDF bidimensional de um círculo fonte

- **Raios:** Para renderizar cenas parametrizadas por sdfs, é necessário partir da posição da câmera e lançar raios que atravessam o plano da imagem a ser renderizada. Cada raio deve atravessar um pixel correspondente no plano da imagem e marchar em direção à cena.

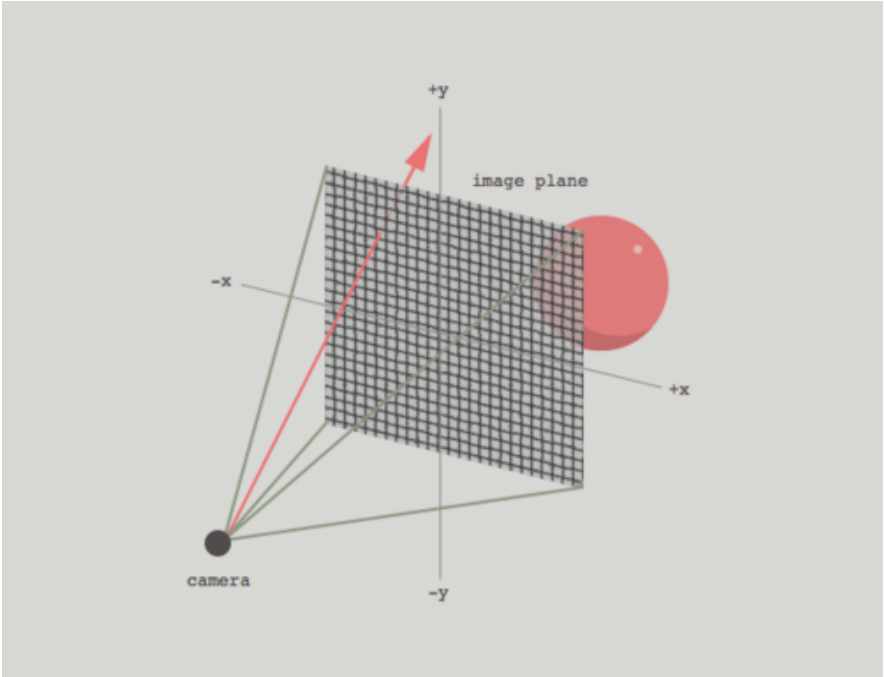


Figure: plano da imagem, fonte

- **Hit or Miss:** Os raios lançados são amostrados regularmente. Caso a distância em relação a alguma superfície calculada para uma dada amostragem de um raio seja menor que um *"threshold"*, considera-se que o raio atingiu a superfície em questão. A amostragem desse raio é interrompida e a cor da superfície atingida é desenhada no pixel atravessado pelo raio.

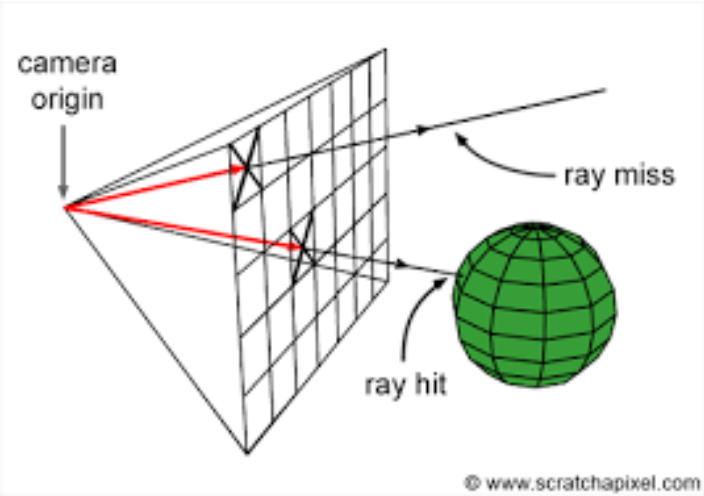


Figure: Amostragem de um raio que atingiu uma superfície, o pixel atravessado pelo raio que atingiu a esfera deverá ser colorido de verde. fonte

Exemplos

- As imagens finais renderizadas podem ser fantásticas e dependem apenas da capacidade do artista de escrever SDFs e funções de iluminação.

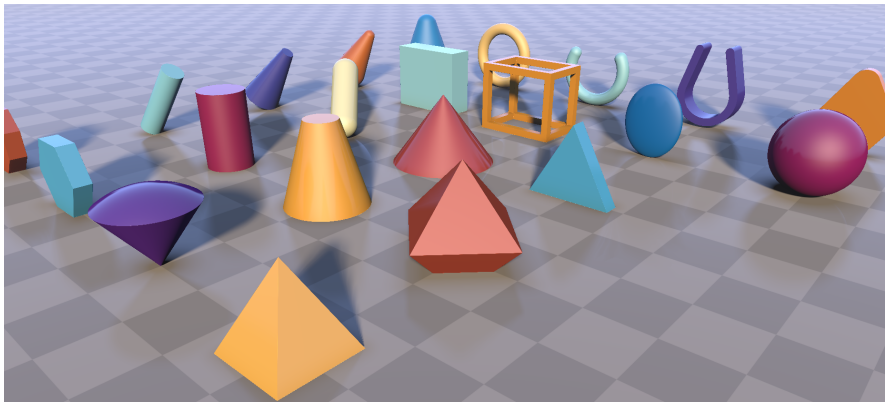


Figure: Artistas muito talentosos como Iñigo Quilez são capazes de criar cenas compostas por dezenas de SDFs diferentes. fonte

Linguagem

- **SDL:** O objetivo então da linguagem é mais do que abstrair a construção do renderizador, abstrair também o processo de amostragem volumétrica da cena que chamamos de raymarching. Dessa forma, objetiva-se poder escrever apenas as funções de distância, funções de cor da cena e a linguagem vale-se do algoritmo de raymarching para renderizar a cena descrita pelo usuário na linguagem.

Código 1 Exemplo

```
#in vec3 point, out vec3 color, out float distance, opt width 1000, opt height 1000, opt steps 5

function distance(p1, p2)
    return ((p1.x - p2.x)^2 + (p1.y - p2.y)^2 + (p1.z - p2.z)^2)^0.5
end

function sphereSDF(p)
    local center = vec3(0.0, 0.0, 0.0)
    local radius = 1.0
    return distance(p, center) - radius
end

function SignedDistance(p)
    return sphereSDF(p)
end

function Color(p)
    return vec3(1.0, 0.5, 0.0)
end

local distance = SignedDistance(point)
local color = Color(point)
```

Figure: exemplo de código aceito pela linguagem.

Saída Exemplo 1

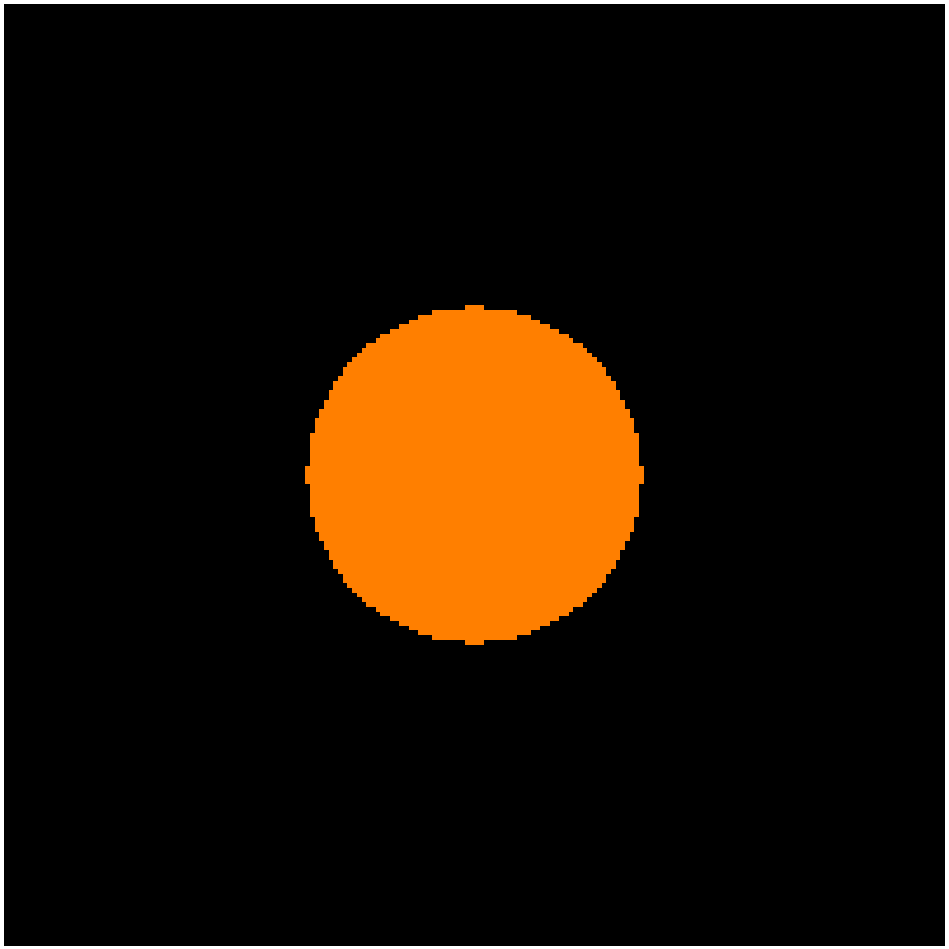


Figure: Saída obtida para o primeiro exemplo

Código Exemplo 2

```
#in vec3 point, out vec3 color, out float distance, opt width 200, opt height 200, opt steps 5

function max(a, b)
    local result
    if a > b then
        result = a
    else
        result = b
    end
    return result
end

function min(a, b)
    local result
    if a < b then
        result = a
    else
        result = b
    end
    return result
end

function abs(v)
    local result = v
    if result < 0.0 then
        result = -v
    else
        result = v
    end
    return result
end

function distance(p1, p2)
    return ((p1.x - p2.x)^2 + (p1.y - p2.y)^2 + (p1.z - p2.z)^2)^0.5
end
```

Figure: Funções e diretivas de compilação para segundo exemplo.

Código Exemplo 2

```
function cubeSDF(p, size)
    local halfSize = size / 2.0
    local dX = max(abs(p.x) - halfSize, 0.0)
    local dY = max(abs(p.y) - halfSize, 0.0)
    local dZ = max(abs(p.z) - halfSize, 0.0)
    local maxD = vec3(dX, dY, dZ)
    local minD = min(maxD.x, min(maxD.y, maxD.z))
    return minD + distance(maxD, vec3(0.0, 0.0, 0.0))
end

function SignedDistance(p)
    return cubeSDF(p, 1.0)
end

function Color(p)
    return vec3(0.0, 0.0, 1.0)
end

local distance = SignedDistance(point)
local color = Color(point)
```

Figure: Funções e diretivas de compilação para segundo exemplo.

Saída Exemplo 2

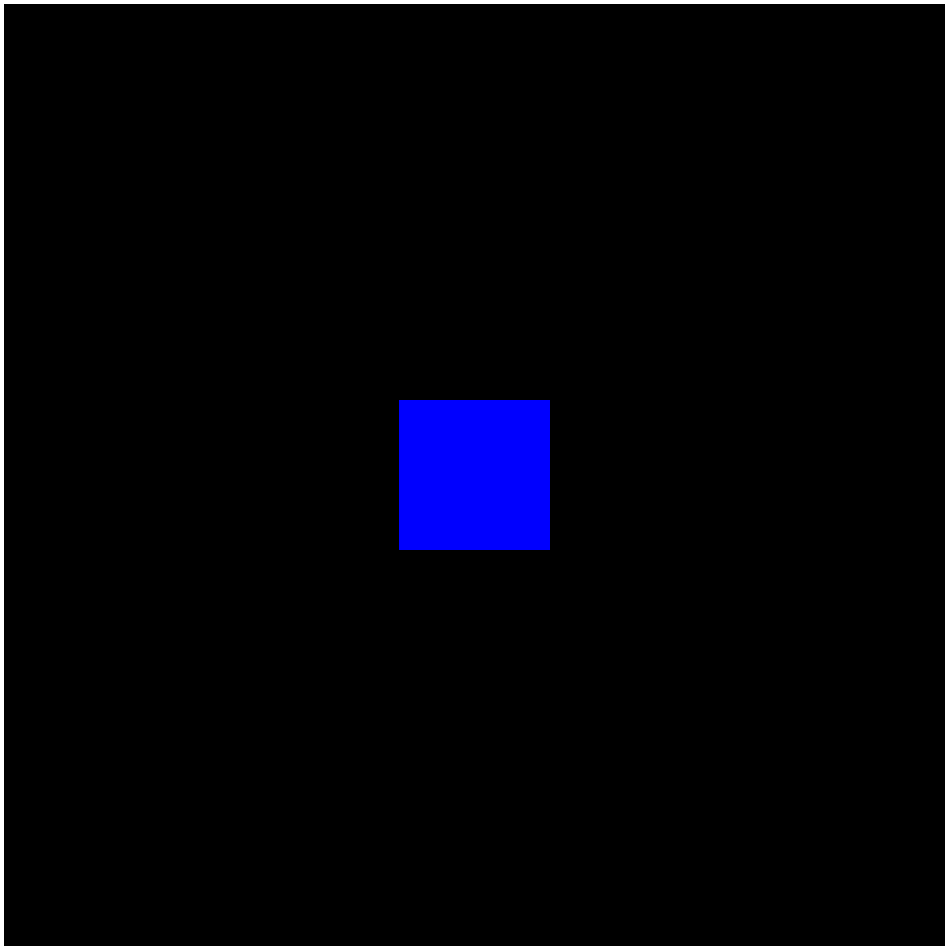


Figure: Saída obtida para o segundo exemplo

Código Exemplo 3

```
#in vec3 point, out vec3 color, out float distance, opt width 100, opt height 100, opt steps 5

function abs(p)
    local result
    if p < 0.0 then
        result = -p
    else
        result = p
    end
    return result
end

function Color(p)
    local normalized_x = (sin(10.0*p.x) +1.0)/2.0
    local normalized_y = (cos(10.0*p.y)+1.0)/2.0
    local r = normalized_x
    local g = normalized_y
    local b = 0.0
    return vec3(r, g, b)
end

function sdf(p)
    local distance = 1.0
    if p.z > 0.0 then
        distance = 0.0
    end
    return distance
end

local distance = sdf(point)
local color = Color(point)
```

Figure: Código do terceiro exemplo

Saída Exemplo 3

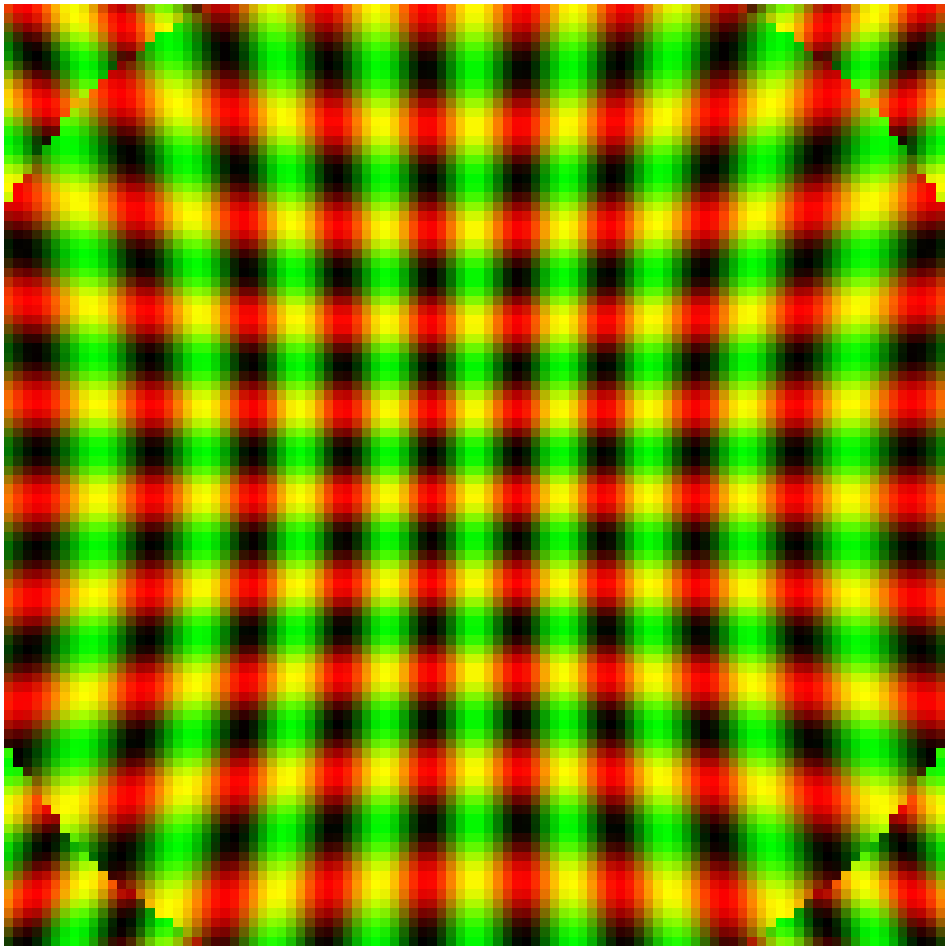


Figure: Saída obtida para o terceiro exemplo

Resumo de implementações

- **float** interpretação e operação com números de ponto flutuante.
- **Vectors** construção de vec2, vec3, vec4.
- **Property Access** acesso de propriedade de vetores.
- **Operações** suporte para operações vetoriais.
- **Diretivas de Compilação** parâmetros que determinam como a árvore sintática será executada podem ser escritos no início do programa.
- **Renderização via sdf** O output de um programa sdl já é uma imagem renderizada com base nas variáveis de cor e distância fornecidos pelo usuário.
- **Funções trigonométricas** Funções trigonométricas in-built foram implementadas no factor para possibilitar uma maior variedade de funções de cor e distância.

Discussão execução

Injeção de variáveis: O processo de raymarching exige a avaliação da árvore sintática para cada pixel na tela e para cada passo do raio lançado, ou seja, é necessário injetar um valor novo na variável vec3 descrita como "in" no topo do código para cada unidade de espaço da cena sendo marchada.
Para isso, a cada reavaliação da árvore é necessário criar uma nova symbol table e injetar o valor atual da variável "in" nessa nova symbol table.

```
def march(self,x,y,camera_pos,fov,original_block,aspect_ratio,opt_width,opt_height,opt_steps,in_variable_name,out_distance_name,out_color_name):

    px = (2 * (x + 0.5) / float(opt_width) - 1) * np.tan(fov / 2) * aspect_ratio
    py = (1 - 2 * (y + 0.5) / float(opt_height)) * np.tan(fov / 2)
    ray_dir = np.array([px, py, 1])
    ray_dir /= np.linalg.norm(ray_dir)

    march_pos = np.copy(camera_pos)
    color = [0,0,0]
    for step in range(opt_steps):
        block = copy.deepcopy(original_block)
        point = march_pos + ray_dir * step



        global funcTable
        funcTable = FuncTable()
        table = SymbolTable()

        table.create(in_variable_name, (Vec3(*point),'vec3'))
        block.Evaluate(table)


        dist = table.get(out_distance_name)[0]
        if dist < 0.01:
            color = table.get(out_color_name)[0]
            color = [color.x, color.y, color.z]
            break

        march_pos += ray_dir * dist
    return color
```

Figure: Porção do código responsável por avaliar a árvore sintática