



*ugr* | Universidad  
de **Granada**

**E.T.S de Ingenierías Informática y de Telecomunicación**  
**GRADO EN INGENIERÍA INFORMÁTICA**

TRABAJO FIN DE GRADO

**Funciones de distancia con signo**

Autor.  
Lukas Häring García

Director.  
Juan Carlos Torres Cantero

---

Granada  
Curso académico 2019-2020



## RESUMEN

---

Gracias al rápido avance tecnológico y la evolución de la unidad de procesamiento de gráficos o *GPU*, podemos experimentar con técnicas propuestas durante el siglo pasado, que eran poco eficientes debido al hardware del momento. El objetivo de este trabajo es hacer uso de este avance para probar nuevas técnicas de creación de escenas, utilizando una clase de funciones, conocidas como *Funciones de distancia con signo* o *FDS5*. Desarrollaremos el proyecto en el lenguaje *GLSL2*, donde veremos alguno de los nuevos tipos y operadores ya que su sintaxis es similar a C. Las escenas creadas a partir de las *Funciones de distancia con signo* son completamente analíticas, es decir, exactas, que a diferencia de otras técnicas que utilizan vértices. Por ejemplo, estas otras técnicas generan una esfera aproximada por un poliedro geodésico<sup>1</sup>. Para el renderizado de la escena tridimensional, en el cuarto capítulo, presentaremos un algoritmo con el nombre de «*Spheremarcher»*[3.1](#).

PALABRAS CLAVE *gráficos, marcher, GLSL, modelo de iluminación*

---

<sup>1</sup> Se trata de un poliedro convexo hecho de triángulos, el algoritmo: <https://stackoverflow.com/a/17795311>



## SUMMARY

---

In this project we are going to present a new graphic technique in thanks to the advance in technology and the continuously increasing of the efficiency of the GPU. We are going to divide it into five important chapters.

**CHAPTER 1** In this first chapter we are going to present the mathematical foundation used to create an analytical scene for our project. We will give an introduction to the *signed distance functions* and dive into them in later chapters. Because we are working with analytical scenes, we will present a theorem to calculate surface normal that will be in the Light Model. Finally, we will give two mathematical concepts: *homeomorphism* and *homotopies*, commonly used in textures.

**CHAPTER 2** In this second chapter, we will present the language we will use through the whole project, called *GLSL*. The syntax of this language is similar to C, we will be presenting some new mathematical types and primitive functions. Each section will also have tips and code examples.

**CHAPTER 3** In this chapter we will analyze the analytical algorithm used to approximate a sceene that is created using *signed distance functions*. We will be presenting the online enviroment used to create this project, called *Shadertoy*, designed by Iñigo Quilez and Pol Jeremias in 2013.

**CHAPTER 4** In this chapter, we will be presenting the basic operators to create an Illumination Model. We are going to implement an Illumination Model designed in the 70's, called Phong's Model. This model splits the light intensity into three different ones: ambient intensity, that affects with a constant value at the surface everywhere; diffuse intensity, dependent to the surface curvature and the light direction and specular intensity is calculated by the reflection of the light from the surface to the eye. Finally, we will be presenting the easiest method to create a shadow used to give a notion of depth.

**CHAPTER 5** In this chapter we will be covering the main theme of this project, that are indispensable for the creating of a scene. *Signed distance functions* are a special type of functions that returns

a signed distance to a surface or perimeter, when the distance is positive, we would say that we are outside the scene, when it is negative, we will be inside an object. The zeros will represent the surface, but because we will be using a numerical algorithm to march it, we will define the surface as an interval. We will be proving some of the most important primitives for 2-dimensional and 3-dimensional spaces and two types of operators, one that will not affect the marcher and the other that will create artefacts and aggravate the efficiency of the marcher.

**CHAPTER 6** In this penultimate chapter we will be presenting techniques to draw exact scenes, independently to the type of operator, we will talk about two problems, overestimation and underestimation. Overestimation happens when we are working with nonexact *signed distance fields* and the marcher goes inside the surface or break through it. The underestimation happens in both cases, it considers background surfaces that, with more iterations, will be traced.

**CHAPTER 7** Finally, in this last chapter, we will show how to use different colors and materials, we will give an ID to each object from the scene, that, will make some a lot of changes in code to work. We will present projection of a shape into texture coordinate space.

**KEYWORDS** *graphics, signed distance functions, illumination model, texturing*

## ÍNDICE GENERAL

---

INTRODUCCIÓN	8
1 PRELIMINARES	11
1.1 Funciones de distancia con signo . . . . .	11
1.2 La normal de una isosuperficie . . . . .	12
1.3 Homeomorfismos sobre $[0, 1]$ . . . . .	12
1.4 Homotopías . . . . .	14
2 LENGUAJE GLSL	17
2.1 Tipos . . . . .	17
2.2 Enlaces . . . . .	18
2.3 Vectores . . . . .	18
2.4 Matrices . . . . .	19
2.5 Operadores matemáticos . . . . .	21
2.6 Acceso a texturas . . . . .	24
3 MARCHER	25
3.1 Spheremarching . . . . .	26
4 MODELO DE ILUMINACIÓN	33
4.1 Luz e Intensidad . . . . .	35
4.1.1 Intensidad ambiente . . . . .	35
4.1.2 Intensidad difusa . . . . .	35
4.1.3 Intensidad especular . . . . .	36
4.1.4 Modelo de Phong . . . . .	37
4.2 Sombras . . . . .	39
5 FUNCIONES DE DISTANCIA CON SIGNO	43
5.1 Primitivas sobre $\mathbb{R}^2$ . . . . .	43
5.1.1 Circunferencia exacta . . . . .	44
5.1.2 Rectángulo exacto . . . . .	45
5.1.3 Recta exacta . . . . .	47
5.1.4 Segmento exacto . . . . .	48
5.2 Operadores sobre $\mathbb{R}^2$ . . . . .	50
5.2.1 Operador de traslación . . . . .	50
5.2.2 Operador de rotación . . . . .	51
5.2.3 Operador de simetría . . . . .	53
5.2.4 Operador de agregación . . . . .	55
5.2.5 Operador de substracción . . . . .	56
5.2.6 Operador de escalado . . . . .	58
5.2.7 Operador de deformación sin exactitud . . . . .	59
5.3 Primitivas en $\mathbb{R}^3$ . . . . .	60
5.3.1 Esfera exacta . . . . .	60
5.3.2 Prisma rectangular exacto . . . . .	61
5.3.3 Plano con signo . . . . .	62

5.3.4 Recta y segmento exacto . . . . .	63
5.4 Operadores sobre $\mathbb{R}^3$ . . . . .	64
5.4.1 Operadores Isométricos . . . . .	64
5.4.2 Operador de ensanchamiento . . . . .	66
5.4.3 Operador de revolución . . . . .	67
5.4.4 Operador de extrusión . . . . .	69
5.4.5 Operador de agregación y substracción . . .	72
5.4.6 Operador de deformación no exacta . . . .	74
6 RESOLUCIÓN DE ARTEFACTOS	77
6.1 Sobreestimación de la distancia . . . . .	77
6.1.1 Sobreestimación dentro de la superficie . .	78
6.2 Sobreestimación fuera de la superficie . . . . .	79
6.2.1 Subestimación de la distancia . . . . .	80
7 MATERIALES	83
AGRADECIMIENTOS	88

## INTRODUCCIÓN

---

Para la confección de este trabajo ha sido indispensable la ayuda de mi tutor del proyecto, Prof. Dr. Juan Carlos Torres Cantero y al departamento de Lenguajes y Sistemas Informáticos.

Nuestro trabajo está estructurado en siete grandes apartados. Un primer apartado en el que introducimos los principales conceptos que vamos a emplear a lo largo de nuestro trabajo, para demostrar el dominio de los conceptos trabajados durante estos cuatro años en la Universidad de Granada. Un segundo apartado en el que presentamos el lenguaje de programación utilizado, *GLSL2*. Un tercer apartado en el que desarrollamos el algoritmo *Spheremarcher*, esencial para la confección de este trabajo. En el cuarto capítulo, presentaremos algunos operadores importantes para la creación de un modelo de iluminación. Presentaremos dos tipos de luces: radiales y direccionales. Haremos uso de un modelo de iluminación conocido con el nombre de *Modelo Phong*. Un quinto capítulo, donde veremos en profundidad las *funciones de distancia con signo*<sup>5</sup>,



# 1. PRELIMINARES

---

## 1.1 FUNCIONES DE DISTANCIA CON SIGNO

Una función de distancia con signo (*FDS*) se trata de una aplicación  $f : \mathbb{R}^n \rightarrow \mathbb{R}$ . Esta aplicación transforma un punto  $\vec{p}$  de un espacio multidimensional en un valor. En particular, trabajaremos con espacios bidimensionales y tridimensionales, donde el valor devuelto es la distancia mínima con signo hasta un punto  $\vec{q}$  de un perímetro o una superficie  $S$ .

$$f(\vec{p}) = \pm \min_{\vec{q} \in S} \|\vec{p} - \vec{q}\|, \vec{p} \in \mathbb{R}^n, n \in \{2, 3\}$$

El signo de esta distancia contiene información sobre la escena: con una distancia positiva, nos encontramos en el exterior de una figura, en caso de ser cero, nos encontraremos en la superficie (corteza)  $S$ , por último, con una distancia negativa, estaremos dentro del volúmen de la figura, que en positivo representa la profundidad del punto respecto de la corteza.

DEFINICIÓN 1.1.1. Sea  $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ , una función de distancia con signo, definimos como *isoperímetro*,  $L = \{\vec{p} | f(\vec{p}) = 0\}$ .

DEFINICIÓN 1.1.2. Sea  $f : \mathbb{R}^3 \rightarrow \mathbb{R}$ , una función de distancia con signo, definimos como *isosuperficie*,  $S = \{\vec{p} | f(\vec{p}) = 0\}$ .

Para esta última definición, presentaremos un algoritmo de trazado de escena denominado *Spheremarcher*[3.1](#).

## 1.2 LA NORMAL DE UNA ISOSUPERFICIE

Vamos a presentar un teorema<sup>1</sup> esencial para el cálculo de la normal de una superficie ya que es fundamental para la confección de una escena, especialmente en un modelo de iluminación??.

**TEOREMA 1.2.1.** *El vector gradiente  $\nabla f(x_0, y_0, z_0)$  es perpendicular a la curva de la tangente de una isosuperficie en el punto  $\vec{p} = (x_0, y_0, z_0)$ .*

En realidad, nos quiere decir que la normal de una *isosuperficie* es proporcional a su gradiente o exacta en caso de su posterior normalización<sup>2</sup>, que denotaremos:

$$\text{norm} : \mathbb{R}^n \longrightarrow \mathbb{R}^n, \text{norm}(\vec{v}) = \frac{\vec{v}}{\|\vec{v}\|}$$

Se define formalmente el gradiente de una función tridimensional, como:

$$\nabla f(x, y, z) = < \partial_x f, \partial_y f, \partial_z f, >$$

donde « $\partial_{x_i}$ » es la derivada parcial de una función respecto de la variable  $x_i$ . Definida por:

$$\partial_{x_i} f = \lim_{\epsilon \rightarrow 0} \frac{f(x_0, \dots, x_i + \epsilon, \dots, x_n) - f(x_0, \dots, x_n)}{\epsilon}$$

Se puede aproximar computacionalmente, tomando  $\epsilon$  próximo a 0, por ejemplo,  $\epsilon = 0.001$ . Finalmente, definimos el gradiente computado como:

$$\vec{n} = \text{norm}(\nabla f(x, y, z)) \approx \text{norm} \left( \begin{pmatrix} \frac{f(x + 0.001, y, z) - f(x, y, z)}{0.001} \\ \frac{f(x, y + 0.001, z) - f(x, y, z)}{0.001} \\ \frac{f(x, y, z + 0.001) - f(x, y, z)}{0.001} \end{pmatrix} \right)$$

## 1.3 HOMEOMORFISMOS SOBRE $[0, 1]$

Vamos a definir una aplicación que nos ayudará a deformar funciones de manera continua. Esta aplicación nos será muy útil para manipular las propiedades de la escena, por ejemplo, la intensidad lumínica en nuestro modelo de iluminación, transformación de color, velocidad de un objeto de un punto a otro, etc.

---

<sup>1</sup> [https://ocw.mit.edu/courses/mathematics/18-02sc-multivariable-calculus-fall-2010/2.-partial-derivatives/part-b-chain-rule-gradient-and-directional-derivatives/session-36-proof/MIT18\\_02SC\\_notes\\_19.pdf](https://ocw.mit.edu/courses/mathematics/18-02sc-multivariable-calculus-fall-2010/2.-partial-derivatives/part-b-chain-rule-gradient-and-directional-derivatives/session-36-proof/MIT18_02SC_notes_19.pdf)

<sup>2</sup> Decimos que un vector está normalizado cuando su módulo es exactamente 1 .

DEFINICIÓN 1.3.1. Sea una función  $f : X \rightarrow Y$ , diremos que esta es homeomófica, si y solo si:

1.  $f$  es continua.
2.  $f$  es biyectiva.
3.  $f^{-1}$  es continua.

Restringimos los homeomorfismos a  $X = Y = [0, 1]$  con  $f(0) = 0$  y  $f(1) = 1$ . Esta restricción es importante para valores normalizados, ya que aseguramos que sus extremos quedan invariantes.

Supongamos una imagen en escala de grises con un único canal. El canal acepta un intervalo real  $[0, 1]$ , donde el color negro es el 0 y el blanco, el 1. Por ejemplo, definimos nuestro *homeomorfismo*:

$$f(x) = x^8, x \in [0, 1]$$

Así, podemos comprobar que esta función cumple con las propiedades descritas anteriormente. En la siguiente representación podemos distinguir una serie de exponentes desde 1 hasta 11, en azul nuestra función  $f$ .

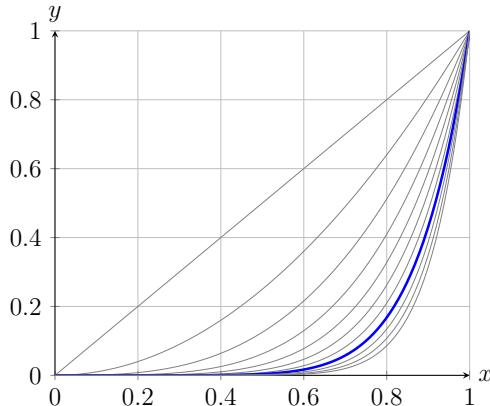


FIGURA 1.1: Gráfica de los distintos  $f_n(x) = x^n, n \in \{1, \dots, 11\} \subset \mathbb{N}$

Los valores inferiores a 0.6 se transforman a valores próximos a 0.0, oscureciendo los tonos grises, manteniendo los colores claros.

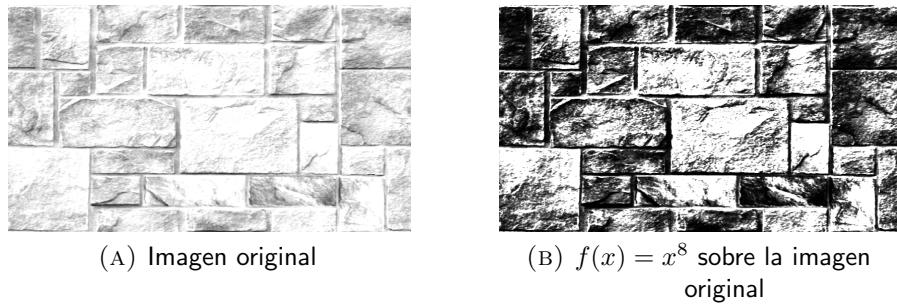


FIGURA 1.2: A la izquierda una imagen con un canal  $[0, 1]$ . A la derecha, la misma imagen, pero aplicado el *homeomorfismo*  $f$ . En código: <https://www.shadertoy.com/view/wljfR1>

Podemos definir nuestra función  $f$  sobre el intervalo  $[0, 1]$  con menos requerimiento computacional utilizando una *aproximación de Taylor para un intervalo*<sup>3</sup>:

$$f(x) = \sin\left(\frac{\pi}{2}x\right) \approx \max\left(\frac{\pi}{2}x - \frac{\pi^3}{48}x^3 + \frac{\pi^5}{3840}x^5, 1\right)$$

## 1.4 HOMOTOPÍAS

En esta última sección, vamos a ver una aplicación matemática que es utilizada para animación y texturización. Vamos a centrarnos en el intervalo  $[0, 1]$ , aunque podemos trabajar sobre cualquier intervalo, antes deberemos normalizar y finalmente, reescalar.

**DEFINICIÓN 1.4.1.** Dadas dos aplicaciones  $f, g : X \rightarrow Y$ , continuas, decimos que son homotópicas. Si existe una aplicación  $H$ , también continua, tal que:

$$H : X \times [0, 1] \rightarrow Y$$

$$H(x, 0) = f(x)$$

$$H(x, 1) = g(x)$$

**DEFINICIÓN 1.4.2.** Claramente  $H$  es una homotopía, llamaremos función de interpolación lineal o función de mezcla con peso<sup>4</sup> a la aplicación:

$$H(x, t) = (1 - t) \cdot f(x) + t \cdot g(x)$$

<sup>3</sup> Aproximación de taylor sobre un intervalo con tolerancia. Enlace: [https://mathinsight.org/achieving\\_desired\\_tolerance\\_taylor\\_polynomial\\_desired\\_interval\\_refresher](https://mathinsight.org/achieving_desired_tolerance_taylor_polynomial_desired_interval_refresher)

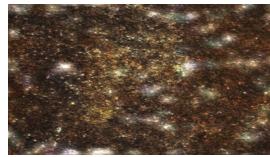
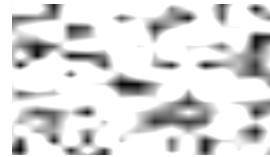
<sup>4</sup> Encontramos este término en libros como

Como podemos observar, la última definición es una homotopía, ya que, la suma de dos funciones continuas es siempre continua y los extremos resultan  $f(x)$  y  $g(x)$ , respectivamente.

Como  $t \in [0, 1]$ , podemos aplicar un *homeomorfismo*  $p(t)$  y tener así, una versión más general.

$$H(x, t') = H(x, p(t)) = (1 - p(t)) \cdot f(x) + p(t) \cdot g(x)$$

Veamos un ejemplo, supongamos que  $f(x)$  y  $g(x)$  son los colores de una imagen 3-canal y como  $t$ , una tercera imagen con un canal  $[0, 1]$ , que actuará como máscara.

(A) Imagen  $f(x)$ (B) Imagen  $g(x)$ (C) Máscara  $t$ FIGURA 1.3: Imagen  $f(x)$ , Imagen  $g(x)$  y Máscara  $t$ 

Los resultados obtenidos con diferentes homomorfismos.

(A) Mezcla con el homeomorfismo  
 $p(t) = t$ (B) Mezcla con el homeomorfismo  
 $p(t) = t^4$ 

FIGURA 1.4: Mezcla con distintos homomorfismos.

Un ejemplo práctico: <https://www.shadertoy.com/view/wt2fR1>



## 2. LENGUAJE GLSL

---

Vamos a presentar el lenguaje *OpenGL Shading Language, GLSL*<sup>1</sup> del que hace uso la tecnología web actual, *WebGL*<sup>2</sup>. Este lenguaje tiene una sintaxis similar a C y que comparte con muchos otros lenguajes del mismo propósito, por ejemplo, *High Level Shader Language*<sup>3</sup>, *HLSL*.

Estos lenguajes son utilizados para la creación de *shaders*, una aplicación ejecutada por la *GPU* capaz de modificar la geometría, *Vertex Shader* o el color, *Fragment Shader*. Estos son ejecutados en una de las etapas de procesado gráfico enlazados por una *API*. Una *API* define un conjunto de protocolos para la intercomunicación con otros, en particular, *OpenGL Shading Language* usa como *API* *WebGL*. Vamos a presentar algunos de los tipos presentes en el lenguaje, funciones matemáticas y operadores de acceso a texturas.

### 2.1 TIPOS

Tipo	Definición
int	Entero con signo como.
float	Número real, con precisión de 32 bits.
bool	Ocupa un 8bits y representa dos valores, <i>true</i> o <i>false</i> .
vecN	Un vector matemático, N-upla de floats, encontramos definidos: vec2, vec3, vec4.
matN	Se trata de una matriz cuadrada N-dimensional, de floats, definidas: mat2, mat3, mat4.
matNxM	Una matriz rectangular de dimensiones $N \times M$ , encontramos: mat2x2, mat2x3, mat2x4, mat3x2, mat3x3, mat3x4, mat4x2, mat4x3, mat4x4.

1 La documentación oficial del lenguaje: <https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.40.pdf>

2 Esta información es tomada del consorcio industrial *Khronos* de estándares abiertos. <https://www.khronos.org/webgl/>

3 Es un lenguaje desarrollado por la empresa *Microsoft* para su *pipeline* gráfico de su colección de *APIS*, *DirectX*. <https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl>

## 2.2 ENLACES

1. **out** <tipo><variable>. Enlaza la variable con otra de distinto nivel, tal que, al finalizar el shader, su valor es asignado al enlazado.
2. **in** <tipo><variable>. Enlaza la variable con otra de distinto nivel, tal que al inicio del shader, su valor es asignado por el enlazado otro.
3. **uniform** <tipo><variable>. Enlaza la variable con una variable global, de solo lectura.

## 2.3 VECTORES

El tipo vector,  $\text{vec}N$ , definido por una tupla:  $(x, y[, z[, w]])$  ó  $(r, g[, b[, a]])$ . Cuyos constructores aportan una riqueza semántica al lenguaje, por ejemplo:

Constructores	Definición
<code>vecN(float, ..., float)</code>	$N$ valores flotantes para cada componente.
<code>vecN(vecM, float)</code>	Asigna los primeros valores del vector los valores del segundo y el último, el elemento flotante, con $M + 1 = N$ .
<code>vecN(float, vecM)</code>	Asigna al primer atributo, el valor flotante. Los $M$ restantes, con las componentes del vector $\text{vec}M$ , donde $M + 1 = N$ .
<code>vecN(vecP, vecQ)</code>	Los $P$ primeros atributos del vector asignados por las componentes de $\text{vec}P$ y los $Q$ restantes, de las componentes del vector $\text{vec}Q$ , $N = P + Q$ .

El operador de acceso, «.», a las componentes del vector, devolviendo un nuevo vector o *float* formado por los valores en el orden accedido. Presentamos algunos de los operadores vectoriales implementados en el lenguaje:

Función	Definición
<code>length(vecN vector)</code>	Devuelve el módulo del vector.
<code>distance(vecN p1, vecN p2)</code>	Distancia entre dos puntos.
<code>normalize(vecN vector)</code>	Devuelve el vector normalizado.
<code>dot(vecN v1, vecN v2)</code>	<b>Producto escalar</b> de ambos vectores.
<code>cross(vecN v1, vecN v2)</code>	<b>producto vectorial</b> de los dos vectores.

Veamos algunos ejemplos de vectores puestos en práctica:

```

1 // Constructores
2 vec3 vector = vec3(1.0);
3 vec3 vector1 = vec3(1.0, 1.0, 1.0);
4 // vector1 y vector son idénticos.
5 vec3 vector2 = vec3(vec2(0.0), 1.0);
6 vec4 vector3 = vec4(1.0, vec2(0.0), 1.0);
7 vec4 vector4 = vec4(vec2(0.0), vec2(1.0));
8
9
10 // Operadores de acceso
11 float valor1 = vector1.r;
12 // Esto es siempre cierto
13 if(valor1 == vector1.x){...}
14 // Repetimos la componente accedida:
15 vec3 vector5 = vector1.xxx;
16 // Mezclamos las componentes:
17 vec3 vector6 = vector3.yzw;
18 // Utilizamos los atributos rgba
19 vec4 vector7 = vector1.rggb;
20
21 // Operadores
22 vec3 normal = normalize(vector6);
23 float modulo = length(vector4);
24 float distancia = distance(vector5, vector6);
25 float escalar = dot(vector3, vector4);
26 vec3 vectorial = cross(vector1, vector2);
27 vec2 vector8 = vec2(
28     length(vector.xz),
29     vector.y
30 );
31
32 // Ejemplo de errores:
33 // 1. Dimensionalidad incorrecta:
34 vec3 error = vec3(1.0, 0.5);
35 vec2 error = vector.rrr;
36 // 2. Mezcla de atributos.
37 vec2 error = vector.rx;
38 // 3. Error en argumentos
39 float error = dot(vector2, vector3);

```

## 2.4 MATRICES

Las matrices  $matNxM$  y  $matN$ , formadas por  $N \times M$  y  $N^2$  componentes, respectivamente. El operador de acceso a las componentes

es similar al lenguaje C, del tal forma que:  $[j][i]$  accede a la celda de la fila  $j$ -ésima y columna  $i$ -ésima.

Alguno de los constructores para la creación de estas matrices son:

Constructor	Definición
matN(matM)	Submatriz cuadrada $N$ superior izquierda de $matM$ .
matNxM(matQxP)	Submatriz $N \times M$ superior izquierda de $matQxP$ .
matN(float, ..., float)	$N^2$ valores flotantes.
matNxM(float, ..., float)	$N \times M$ valores flotantes.
matN(vecN,..., vecN)	Formado por $N$ vectores $N$ -dimensionales.
matNxM(vecM,..., vecM)	Formado por $N$ vectores $M$ -dimensionales.
matN( vecM,float, ..., vecM, float )	Formado por $N$ vectores $(M - 1)$ -dimensionales y $N$ valores flotantes.
matNxM( vecP,float, ..., vecP,float )	Formado por $N$ vectores $(M - 1)$ -dimensionales y $M$ valores flotantes.

Hemos visto algunos ejemplos, pero existen una gran variedad de combinaciones posibles. Estas combinaciones tienen que respetar siempre el tamaño de la matriz, pudiéndose intercalar vectores  $vecN$  y valores flotantes.

De la misma forma, vamos a ver alguno de los operadores entre matrices.

Función	Definición
transpose(mat matrix)	Devuelve la transpuesta de la matriz.
matrix1 * matrix2	Devuelve el producto de matrices, donde matrix1 es una matriz $N \times M$ y matrix 2 otra $M \times K$ .
determinant(matN matrix)	Devuelve el determinante de una matriz cuadradas.

Algunos ejemplos de construcción de matrices, acceso a sus componentes y operadores:

```

1
2 // Constructores
3 mat2 matrix2x2 = mat2(0.0);
4 mat3 matrix3x3 = mat3(
5     1.0, vec2(1.0),
6     vec2(0.0), 1.0,
7     1.0, 0.0, 1.0
8 );
9 // Submatrix
10 mat2 submatrix2x2 = mat2(matrix1);
11 mat2x3 matrix2x3 = mat2x3(
12     vec3(1.0),
13     1.0, 0.0, 1.0
14 );
15
16 // Acceso
17 // Desde j, i
18 float valor0 = matrix2x2[0][0];
19 // Desde j y su componente
20 float valor1 = matrix2x2[0].x;
21 // Esto siempre es cierto
22 if(valor0 == valor1){ ... }
23 // Obtenemos la fila al completo
24 vec2 vector1 = matrix2x2[0];
25
26 // Operadores
27 // Transpuesta
28 mat3 tmatrix3x3 = transpose(matrix3x3);
29 // Multiplicación
30 mat2x3 mulmatrix2x3 = matrix2x2 * matrix2x3;
31
32 // Errores
33 // 1. Error número de componentes
34 mat2 matriz = mat2(1.0, 0.0);
35 // 2. Error de dimensionalidad.
36 mat2 matriz = mat2(vec3(1.0), vec3(0.0));
37 // 3. Error en argumentos
38 mat2 matrix = matrix2x2 * matrix3x3;
```

## 2.5 OPERADORES MATEMÁTICOS

Encontramos los operadores usuales:  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\%$ . Para los números enteros, tenemos también los operadores binarios:  $<<$ ,  $>>$ ,  $|$ ,  $\&$ ,  $^$ .

Agrupamos *float* y *vecN* con el nombre de *genType*<sup>4</sup> para reunir los tipos de argumentos. Cuando utilizamos un operador sobre el tipo *vecN*, este se aplicará sobre cada una de sus componentes. Algunos operadores trigonométricos y exponenciales son los siguientes:

Función	Definición
radians( <i>genType var</i> )	Conversión de grados en radianes.
degrees( <i>genType var</i> )	Conversión de radianes en grados.
sin( <i>genType var</i> )	Aplica $f(x) = \sin(x)$ sobre la variable.
cos( <i>genType var</i> )	Aplica $f(x) = \cos(x)$ sobre la variable.
tan( <i>genType var</i> )	Aplica $f(x) = \tan(x)$ sobre la variable.
asin( <i>genType var</i> )	Aplica $f(x) = \arcsin(x)$ sobre la variable.
acos( <i>genType var</i> )	Aplica la $f(x) = \arccos(x)$ sobre la variable.
atan( <i>genType var</i> )	Aplica $f(x) = \arctan(x)$ sobre la variable.
pow( <i>genType a,</i> <i>genType b</i> )	Calcula la primera variable elevada a la segunda.
exp( <i>genType var</i> )	Aplica $f(x) = \exp x$ sobre la variable.
exp2( <i>genType var</i> )	Aplica $f(x) = 2^x$ sobre la variable.
log( <i>genType var</i> )	Aplica $f(x) = \log(x)$ sobre la variable.
sqrt( <i>genType var</i> )	Aplica $f(x) = \sqrt{x}$ sobre la variable.

Ejemplos de equivalencias de los operadores:

```

1 #define PI 3.1415
2 vec2 radians = vec2(PI, PI/2));
3 //resultado: aprox vec2(180, 90)
4 float sink = sin(PI / 2.); // aprox 1.0
5 float powk = pow(2., 2.); // 4.0
6 // Ecuacion compleja
7 float complejo = log(exp(-cos(0.))); // -1.0
8 vec2 complejo2 = cos(acos(vec2(0.))); //vec2
      (0)
9 // Error, los dos tipos no son iguales.
10 float error = pow(vec2(0., 2.), 3.);
11 // Solución
12 float solucion = pow(vec2(0., 2.), vec2(3.));

```

Otros operadores, también importantes, son:

---

4 Este término es utilizado en la documentación del lenguaje<sup>1</sup>, «8. Built-in Functions», páginas 140-141

Función	Definición
abs(genType var)	Aplica $f(x) =  x $ sobre la variable.
sign(genType var)	Aplica la función signo sobre la variable.
floor(genType var)	Aplica $f(x) = \lfloor x \rfloor$ sobre la variable, redondeo inferior.
ceil(genType var)	Aplica $f(x) = \lceil x \rceil$ sobre la variable, redondeo superior.
round(genType var)	Aplica $f(x) = [x]$ sobre la variable, redondeo normal.
fract(genType var)	Toma la parte fraccional de la variable, $f(x) = x - \lfloor x \rfloor$ .
min(genType a, genType b)	Toma el mínimo de las variables.
max(genType a, genType b)	Toma el máximo de las variables.
clamp( genType v, (genType ó float) min, (genType ó float) max )	Aplica la función acotado inferior <i>min</i> y superior <i>max</i> , sobre la variable.
mix( genType a, genType b, (genType ó float ó bool) h )	Aplica la función de mezcla con peso <a href="#">1.4.2</a> , sobre la variable, donde $f(x) = a, g(x) = b$ y $t = h$ .

Algunos otros ejemplos y sus equivalencias:

```

1 float absk = abs(-1.0); // 1.0
2 vec2 absv = abs(vec2(-1., 2.)); //vec2(1., 2.)
3 float roundk = 1.23; // 1.00
4 float floork = 1.53; // 1.00
5 float ceilk = 1.23; // 2.0
6 float fractk = 1.23; // 0.23
7 vec2 min = min(
8     vec2(1.0, -1.0),
9     vec2(0.0, 2.0)
10 ); // vec2(0.0, -1.0)
11 float mix1 = mix(1.0, 2.0, 0.5); // 1.5
12 float mix2 = mix(1.0, 3.0, true); // 3.0

```

Los operadores relacionales son los mismos que en el lenguaje C:

<, <=, >, >=, !, ==, !=

## 2.6 ACCESO A TEXTURAS

Desde *Shadertoy*, podemos utilizar diferentes canales y seleccionar distintos tipos de archivos, en nuestro caso, utilizaremos aquellas que aparecen en la pestaña: "Texturas". Una vez seleccionada una textura, podemos utilizar la variable de tipo *sampler2D* con nombre *iChannelN* como canal N-ésimo, para acceder a los valores del archivo.

Existen las siguientes funciones que permiten obtener el color de un píxel de una variable tipo *sampler2D* en las coordenadas de textura (*s, t*), que se encuentran normalizados.

Función	Definición
<pre>texture(     sampler2D iChannelN,     vec2 coord ) textureLod(     sampler2D iChannelN,     vec2 coord,     float lod,</pre>	Devuelve un píxel <i>vec4 rgba</i> en la coordenada noormalizada <i>coord</i> .
)	Devuelve un píxel <i>vec4 rgba</i> en la coordenada noormalizada <i>coord</i> con un nivel de detalle <i>lod</i> .

Un ejemplo de acceso a texturas, algoritmo de suavizado:

```

1 // Sea (x, y) las coordenadas normalizadas.
2 // Tamaño de la máscara para su suavizado.
3 #define K 2.0
4 // Imagen (w, h) en el canal iChannel0
5 float px = 1. / w, py = 1. / h;
6 // Suavizado
7 vec3 pixel = vec3(0.0);
8 for(float i = -K; i <= K; i += 1.0){
9     for(float j = -K; j <= K; j += 1.0){
10         // Pixel de la máscara
11         float cx = x + px * i;
12         float cy = y + py * j;
13         pixel += texture(
14             iChannel0,
15             vec2(cx, cy)
16         ).rgb;
17     }
18 }
19 // Calculamos la media, de los píxeles
20 pixel /= pow(2. * K + 1., 2.);
```

## 3. MARC HER

---

Un *fragment shader* es aplicado a cada píxel de nuestra pantalla, que es procesado por una *hebra* de la *GPU*, una especie de microprocesador que trabaja de manera individual. La hebra contiene información del pixel como es la posición y la resolución, esta devolverá un color en formato *rgba* con tipo *vec4*.

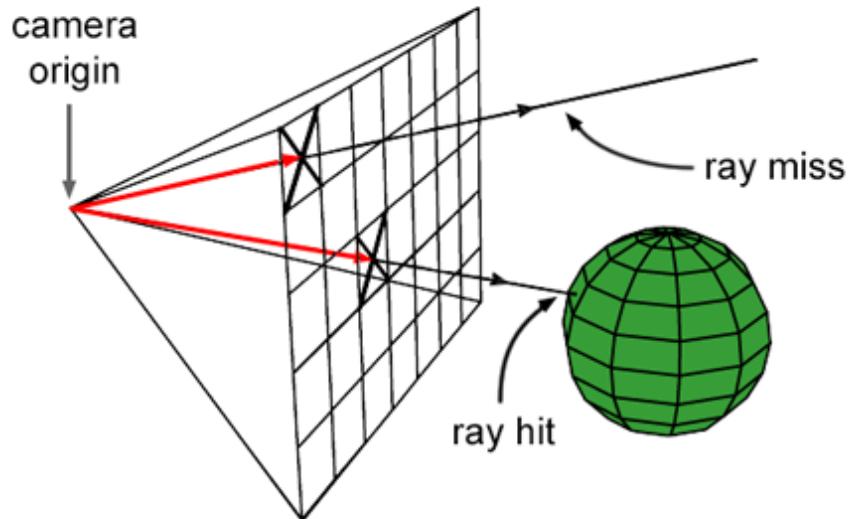
Utilizaremos la plataforma *Shadertoy*<sup>1</sup>, un entorno de trabajo online adecuado para escribir nuestros *shaders*. Este utiliza la *API WebGL* para crear una escena de una figura con 4 vértices formando un rectángulo de dos triángulos posicionado en el plano frontal (o *Viewport*) al que se le va a aplicar el *shader* escrito como un *Fragment Shader*, como podemos observar en [Figura ?? Acceso a texturas](#).

Dada una escena analítica, vamos a presentar técnicas de trazado de escenas, suponiendo que nuestra pantalla se encuentra en la escena y «lanzaremos un rayo» hacia cada pixel desde un punto que denotaremos como la posición de la cámara. Definimos «lanzar un rayo» como el proceso, analítico o numérico, del cálculo de una intersección desde un punto en una dirección. Como proceso analítico, o cálculo exacto, encontramos la técnica *analytical raytracing*<sup>2</sup>, por otro lado, en la técnica numérica, encontramos el algoritmo que vamos a utilizar, *Spheremarching*[3.1](#).

---

1 Creada por Iñigo Quilez y Pol Jeremias, 2013. <https://www.shadertoy.com/>

2 Podemos encontrar más información sobre el tema: <https://www.sciencedirect.com/science/article/pii/S0010465513002713>



© www.scratchapixel.com

FIGURA 3.1: Lanzamiento de rayos para el trazado de una escena

### 3.1 SPHEREMARCHING

Los algoritmos numéricos suelen ser menos costosos computacionalmente, aproximando con una variable de control que afecta al resultado trazado. En particular, se trata de una técnica reiterativa, conocidas como *Raymarcher*. John C. Hart presentó en 1995 un algoritmo de esta categoría con el nombre de *Spheremarching*<sup>3</sup>.

Esta técnica hace uso las *funciones de distancia con signo* como escena y recibe el atributo de iterativo, ya que, desde la posición de la *cámara u ojo*, incrementa el vector director hacia el pixel, conocido como «rayo» y es proporcional a la *función de distancia con signo* devuelto por el vector iterado. Recibe el prefijo «*Sphere*», ya que, podemos generar, para cada iteración, una esfera de radio igual al valor de la distancia con signo, sin ningún punto en su interior. Definimos el vector «rayo» en la iteración n-ésima, como:

$$\vec{p}_n = \vec{ojo} + \vec{direccion} \cdot d_n$$

donde  $d_n$  es la distancia total recorrida por todas las iteraciones:

$$d_n = d_{n-1} + f(\vec{p}_{n-1}) \text{ con } d_0 = 0$$

Al tratarse de un modelo iterativo, debemos describir las condiciones de parada del algoritmo, donde vamos a destacar tres:

<sup>3</sup> El enlace al documento publicado: [https://www.researchgate.net/publication/2792108\\_Sphere\\_Tracing\\_A\\_Geometric\\_Method\\_for\\_the\\_Antialiased\\_Ray\\_Tracing\\_of\\_Implicit\\_Surfaces](https://www.researchgate.net/publication/2792108_Sphere_Tracing_A_Geometric_Method_for_the_Antialiased_Ray_Tracing_of_Implicit_Surfaces)

1. **Primera condición.** El algoritmo finalizará cuando estemos «sobre» la *isosuperficie*. Al tratarse de un algoritmo numérico, vamos a aproximarla, utilizando una variable de control  $\epsilon$  que relajará la restricción de la definición de *isosuperficie*, haciendo  $f(\vec{p}_n) < \epsilon$ , ya que, si  $\epsilon = 0$ , trataríamos de un modelo analítico.
2. **Segunda condición.** Superar una cierta distancia recorrida,  $d_n \geq MAXIMO$ , creando una esfera de trazado sobre el punto de la cámara.
3. **Tercera condición.** Superar el número de iteraciones máximas,  $n \geq PASOS$ .

Este algoritmo devolverá  $d_n$  para el número de iteraciones fijadas, «PASOS». Cuando el algoritmo finaliza debido a la **segunda o tercera condición**, devolverá,  $d_n = MAXIMO$ , recibiendo el nombre de «*falló*». Un fallo, representando un pixel vacío, sin superficie trazada, pudiéndose considerar el fondo de la escena.

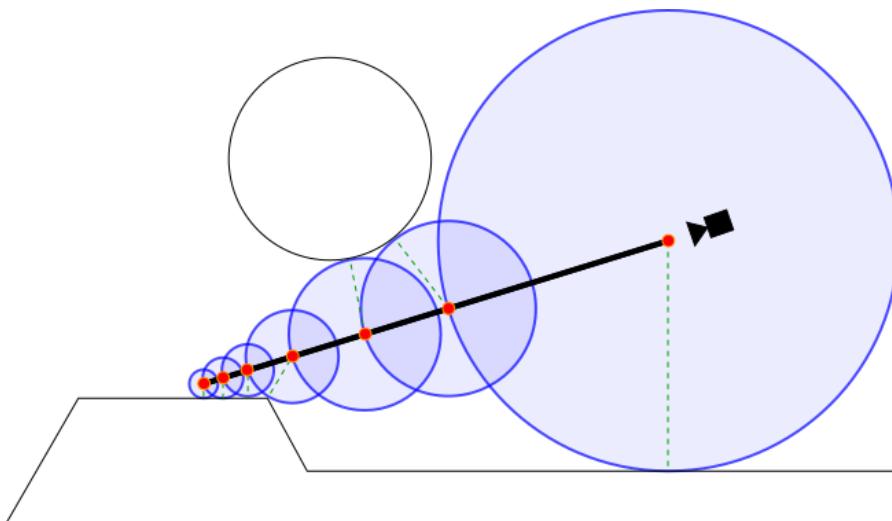


FIGURA 3.2: Ejemplo del algoritmo *Spheremarching*

```

1 #define PASOS 128 // Número Máximo de
    Iteraciones.
2 #define EPSILON 0.001
3 #define MAXIMO 20.0 // Distancia del Plano
    Trasero.
4
5 // Pasamos el origen del ojo y la dirección,
    nos devuelve la distancia al objeto más
    cercano del ojo en dicha dirección.
6 float SphereMarching(
7     in vec3 ojo,
8     in vec3 direccion
9 ){
10    float distancia = 0.0;
11    // Realizamos "PASOS" iteraciones de
        marching.
12    for(int i = 0; i < PASOS; ++i){
13        // Calculamos el vector (rayo).
14        vec3 rayo = ojo + direccion *
            distancia;
15        // Aproximamos el radio de la esfera
            más próxima a una isosuperficie
16        float radio = escena_sdf(rayo);
17        // Si el radio (distancia mínima a la
            isosuperficie), es muy pequeña,
            podemos decir que estamos sobre la
            distancia y devolvemos el módulo
            del rayo.
18        if(radio < EPSILON){
19            return distancia;
20        }
21        // Incrementamos la distancia
            recorrida si no estamos cerca de
            la isosuperficie.
22        distancia += radio;
23        // Comprobamos que no se haya
            superado la distancia de dibujado
            máximo. Podemos considerarlos el
            fondo de la escena.
24        if(distancia >= MAXIMO) break;
25        return MAXIMO;
26    }
27 }
```

Este algoritmo es implementado en un *shader*, aplicado para cada pixel de nuestra pantalla. En el caso de que el algoritmo no devuelva «fallo», podremos calcular a partir del valor devuelto  $d_n < MAXIMO$  el punto aproximado sobre la superficie, tal que:

$$\vec{p} = \vec{ojo} + d_n \cdot \vec{dirección}$$

Conociendo los puntos de una superficie, podremos añadir información a la escena, como por ejemplo, un modelo de iluminación o materiales y texturas.

En *Shadertoy*, crearemos un nuevo *shader* donde se nos ofrecerá un entorno para trabajar y un código de ejemplo:

```

1 void mainImage( out vec4 fragColor, in vec2
    fragCoord )
2 {
3     // Normalized pixel coordinates (from 0
4     // to 1)
5     vec2 uv = fragCoord/iResolution.xy;
6
7     // Time varying pixel color
8     vec3 col = 0.5 + 0.5*cos(iTime+uv.yx+
9     vec3(0,2,4));
10
11    // Output to screen
12    fragColor = vec4(col,1.0);
13 }
```

Observamos en el código tres variables enlazadas [2.2](#) importante:

1. **out vec4 *fragColor***. Enlaza la variable con el valor del pixel, una 4-upla, *rgba*, cuyas componentes están en el intervalo [0, 1].
2. **in vec2 *fragCoord***. Contiene la posición de la coordenada del pixel en pantalla, donde la primera componente representa la coordenada *x* y la segunda, la *y*.
3. **uniform vec2 *iResolution***. Se trata de una variable global que contiene información de las dimensiones en píxeles del *viewport*, su primera componente, el ancho y su segunda, el alto.

Vamos a modificar el ejemplo anterior: Situraremos la pantalla en la escena, posicionandola centrada en las coordenadas (0, 0, 0) y preservando la relación de aspecto. Asignaremos la posición de la cámara u ojo y calcularemos la dirección del ojo al pixel como dirección del rayo. Aplicaremos el algoritmo *spheremarching* desde el ojo en la dirección calculada, en caso de devolver un fallo, utilizaremos el color negro, por el contrario, el blanco.

```

1 void mainImage(
2     out vec4 fragColor,
3     in vec2 fragCoord
4 ){
5     // Normalizamos las coordenadas y las
6     // reescalamos para mantener el ratio de
7     // aspecto. Transladamos al centro de la
8     // pantalla.
9     vec2 uv = (fragCoord - iResolution.xy*.5)
10    / min(iResolution.y, iResolution.x);
11    // Definimos el ojo y la pantalla, que se
12    // encuentra en nuestra escena.
13    vec3 ojo = vec3(0.0, 0.0, -1.0);
14    vec3 pantalla = vec3(uv, 0.0);
15    // La dirección del rayo es el vector
16    // normalizado que apunta desde el ojo
17    // hasta la pantalla (pixel).
18    vec3 direccion = normalize(pantalla-ojo);
19    // Con esto, ya podemos utilizar nuestro
20    // Sphere marcher.
21    float distancia = SphereMarching(ojo,
22        direccion);
23    // El marcher nos ha devuelto una
24    // distancia inferior al plano trasero,
25    // estamos sobre la isosuperficie.
26    if(distancia < MAXIMO){
27        // Estamos aproximadamente sobre la
28        // isosuperficie.
29        // La posición aproximada es la
30        // siguiente.
31        vec3 p = ojo + direccion * distancia;
32        // Utilizamos el color blanco para
33        // dibujar la isosuperficie.
34        fragColor = vec4(1.0);
35    }else{ // El marcher ha fallado.
36        // El color negro para pintar el
37        // fondo.
38        fragColor = vec4(vec3(0.0), 1.0);
39    }
40 }
41 }
```

La función *escena\_sdf*, que se encuentra dentro de la función *SphereMarching*, contiene la escena como una *función de distancia con signo*. Veremos en el sexto capítulo 5, como crear nuestras escenas. Aunque, para probar nuestro algoritmo, vamos a definir una escena muy simple:

```

1  /*
2  Una esfera en el la coordenada (0,0,0) de
   radio 0.2 unidades.
3 */
4 float escena_sdf(vec3 p){
5     return length(p - vec3(0.0)) - 0.2;
6 }
```

Demos una pincelada de como se ha definido esta función, se calcula el módulo del punto  $\vec{p}$ , esto define una *Función de Distancia* y cuya isosuperficie es únicamente un punto,  $S = \{(0,0,0)\}$ , si a cada punto, restáramos  $r$  al la distancia, creamos una isosuperficie esférica de radio  $r$ . Ya que, aquellos puntos cuya distancias valen  $r$ , acabarán anulándose y definiendo una *función de distancia con signo*.

$$S = \{\vec{q} \in \mathbb{R}^3 / SDFEsfera_r(\vec{q}) = 0\}$$

$$SDFEsfera_r(\vec{p}) = ||\vec{p}|| - r$$

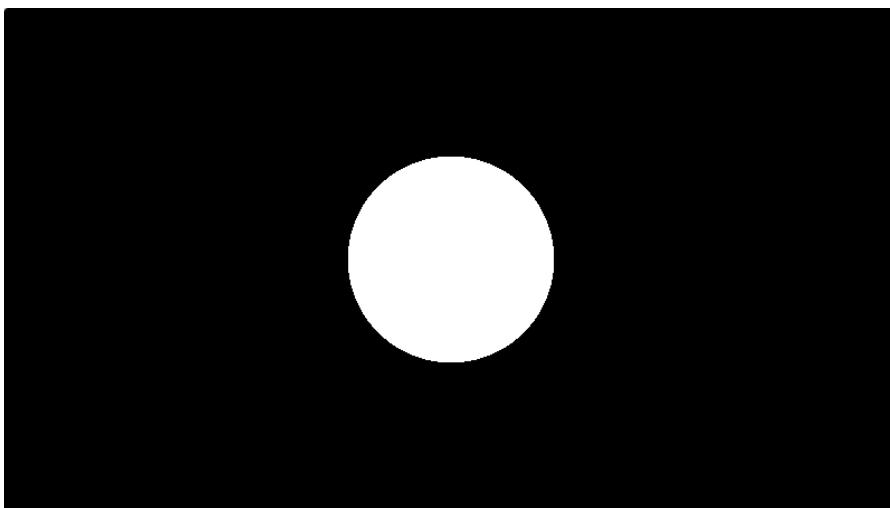


FIGURA 3.3: "Hola mundo" del algoritmo *SDF*.

Enlace del ejemplo <https://www.shadertoy.com/view/wtsfDn>

Al solo utilizar dos colores, blanco y negro, no tenemos sensación de profundidad, esto se conseguirá definiendo un modelo de iluminación.



## 4. MODELO DE ILUMINACIÓN

---

En este capítulo vamos a ver los principios de los modelos de iluminación y algunos operadores importantes. Vamos a presentar el modelo de iluminación Phong, que sigue presente desde su publicación<sup>1</sup> utilizado desde los años 70.

La percepción visual humana y las leyes de la óptica, se consideran indispensables en el desarrollo de una regla de sombreado, proporcionando una mejor calidad y un mayor realismo en las imágenes generadas.

(Bui Thuong Phong)

Dividiremos el capítulo en dos secciones, luces y sombras. Veremos que ambos hacen uso de la normal de la superficie, es por ello que vamos a definir una función de cálculo numérico visto en los *Preliminares*<sup>1.2</sup>.

```
1 // Normal de la isosuperficie en p.
2 vec3 Normal(vec3 p){
3     // f(x1, ..., xn)
4     float fxxyz = escena_sdf(p);
5     // f(x1, ..., xi+h, xn)
6     float fxhyz = escena_sdf(p + vec3(
7         EPSILON, 0., 0.));
8     float fxyhz = escena_sdf(p + vec3(0.,
9         EPSILON, 0.));
10    float fxyzh = escena_sdf(p + vec3(0.,
11        0., EPSILON));
12    // Usamos la definicion de derivadas
13    // parciales para calcular el gradiente,
14    // proporcional a la normal en el punto
15    return normalize(vec3(
16        (fxhyz - fxxyz) / EPSILON,
17        (fxyhz - fxxyz) / EPSILON,
18        (fxyzh - fxxyz) / EPSILON
19    ));
20 }
```

---

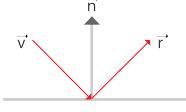
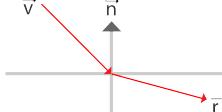
<sup>1</sup> «Illumination for Computer Generated Pictures», por Bui Tuong Phong en 1975: [https://users.cs.northwestern.edu/~ago820/cs395/Papers/Phong\\_1975.pdf](https://users.cs.northwestern.edu/~ago820/cs395/Papers/Phong_1975.pdf)

Vamos a ahora a presentar el operador, *producto escalar*, definido en *GLSL* como "dot", formalmente,  $\cdot : \mathbb{R}^2 \times \mathbb{R}^2 \longrightarrow \mathbb{R}^2$

$$\vec{r} \cdot \vec{v} = r_x v_x + r_y v_y + r_z v_z = |r| |v| \cos(\alpha)$$

Si ambos son vectores directores,  $\vec{r} \cdot \vec{v} = \cos(\alpha)$ . El ángulo  $\alpha$ , es el menor de los ángulos entre los dos vectores sobre el plano que forman en el origen. La imagen de este operador es  $[-1, 1]$ , es fácil observar que en vectores perpendiculares, el producto será cero y en caso de ser paralelos, este será  $\pm 1$  y el signo dependerá de la dirección de ambos.

Además, vamos a presentar dos nuevos operadores<sup>2</sup> que están implementados de manera nativa en el lenguaje *GLSL*, esenciales para los modelos de iluminación:

Función	Definición
<pre>reflect(     vecN v,     vecN n, )</pre>	<p>Operador de reflexión de un vector <math>\vec{v}</math> sobre el vector normal <math>\vec{n}</math>, definido como:</p> $\vec{r} = \vec{v} - 2(\vec{n} \cdot \vec{v})\vec{n}$ 
<pre>refract(     vecN v,     vecN n,     float k, )</pre>	<p>El operador de refracción, un vector <math>\vec{v}</math> refractado sobre el vector <math>\vec{n}</math> normalizado, con <math>k</math> como factor de medio, equivalente a la <i>ley de refracción de Snell-Descartes</i>:</p> $\vec{r} = k \left( \vec{v} - \left( (\vec{n} \cdot \vec{v}) + \sqrt{\frac{1}{k^2} - (\vec{v} \cdot \vec{n})^2} \right) \vec{n} \right)$ 

<sup>2</sup> La demostración de ambos la podemos encontrar en: <http://www.cs.utexas.edu/users/bajaj/graphics2012/cs354/lectures/lect14.pdf>, página 7-9

## 4.1 LUZ E INTENSIDAD

Definimos la *intensidad lumínica* en un punto como un factor multiplicativo al material asignado al punto de la *isosuperficie*, representa cómo de iluminado está. Como es un factor multiplicativo, el valor de 0.0, representa la intensidad nula u oscuridad. Mientras que el valor 1.0 representa el valor más iluminado.

El operador *producto escalar* nos debería dar una breve intuición del papel importante que juega en el cálculo de la intensidad. Como esta no puede ser negativa, definimos el operador producto escalar positivo y normalizado " $\cdot_{[0,1]}$ ".

$$\cdot_{[0,1]} : \mathbb{R}^2 \times \mathbb{R}^2 \longrightarrow [0, 1], \vec{a} \cdot_{[0,1]} \vec{b} = \max \left( \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\| \|\vec{b}\|}, 0 \right)$$

Vamos a presentar dos tipos de luces, *radiales* y *direccionales*, aplicados sobre el "*Modelo de Iluminación de Phong*", presentado en 1973 por *Tuong Phong* como un modelo de iluminación empírico. Para ello, vamos a ver como el modelo se descompone en tres.

### 4.1.1 Intensidad ambiente

Se trata de un valor  $I_a \in [0, 1]$  que indica cuanto de iluminada está la isosuperficie, de manera inicial o si no hubieran luces.

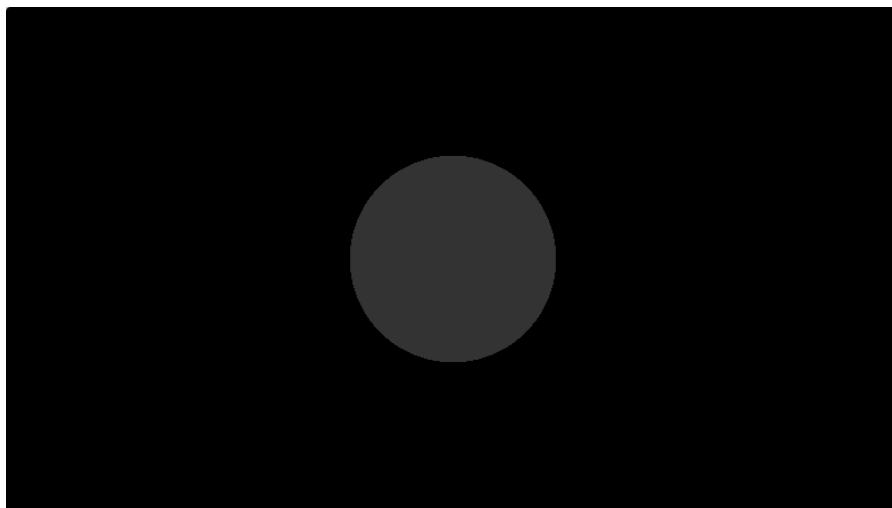


FIGURA 4.1: Intensidad Ambiental sobre la esfera.

### 4.1.2 Intensidad difusa

Esta intensidad representa cuanta luz es refractada por la superficie en un punto  $\vec{p}$  por cada una de las luces de la escena  $\vec{l}_i \in L$ ,

donde  $\vec{l}_i$  representa la posición de la luz, se comprueba como incide la luz sobre la superficie, utilizando la normal  $\vec{n}$  en el punto  $\vec{p}$ .

$$I_d = \sum_{\vec{l}_i \in L} \vec{n} \cdot_{[0,1]} (\vec{l}_i - \vec{p})$$

Es fácil observar que, la intensidad debería ser máxima cuando los rayos inciden en de manera paralela en sentido contrario al vector normal y nulo en caso de que sean perpendiculares u opuestos.

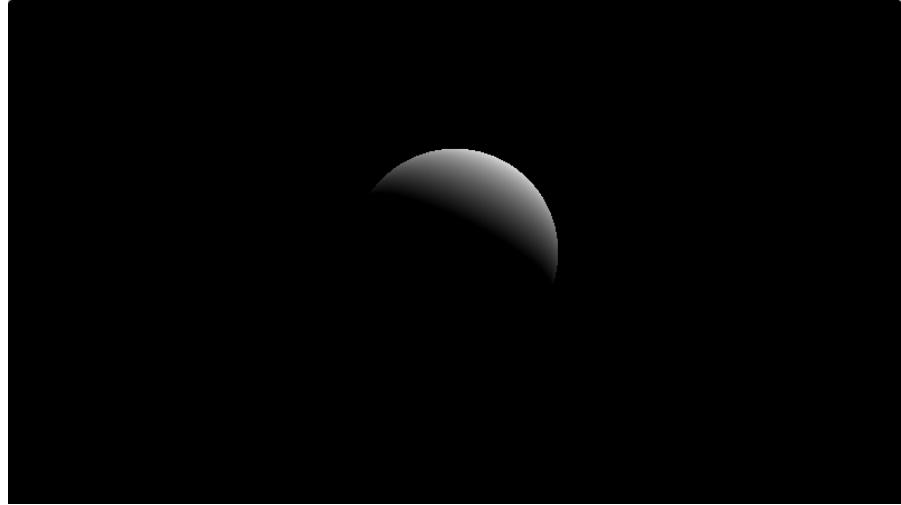


FIGURA 4.2: Intensidad Difusa sobre la esfera.

#### 4.1.3 Intensidad especular

Esta intensidad indica como incide la luz reflectada por la *isosuperficie* en el la dirección del ojo. Definimos « $\vee$ », como el operador de reflexión<sup>??</sup>. La ecuación final de la «*Intensidad Especular*» para todas las luces de la escena es,

$$I_e = \sum_{\vec{l}_i \in L} \vec{ojo} \cdot_{[0,1]} \left( (\vec{l}_i - \vec{p}) \vee \vec{n} \right)$$

Algunos autores aportan una leve modificación de esta ecuación, aplicando un *homeomorfismo* polinómico con grado exponencial, que es utilizado como factor de brillo<sup>3</sup>.

$$h_k : [0, 1] \longrightarrow [0, 1], h_k(x) = x^{2^k}$$

$$I_d = \sum_{\vec{l}_i \in L} h_k \left( \vec{ojo} \cdot_{[0,1]} \left( (\vec{l}_i - \vec{p}) \vee \vec{n} \right) \right)$$

<sup>3</sup> En *OpenGL*, el brillo por reflexión depende de un parámetro de *glMaterial*, *GL\_SHININESS*. <https://www.khronos.org/registry/OpenGL-Refpages/es1.1/xhtml/glMaterial.xml>

Donde  $k \in \mathbb{R}^+$  y este tiene efecto sobre el radio de rayos reflejados.

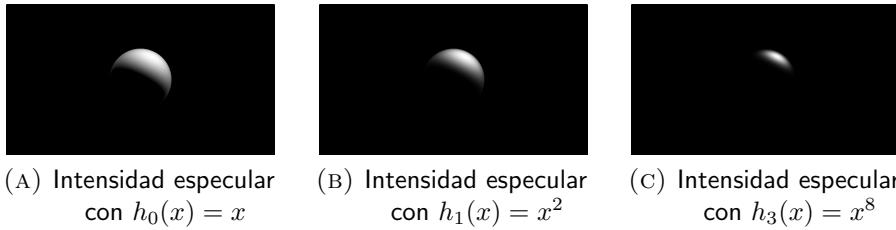


FIGURA 4.3: Intensidad Difusa con distintos homomorfismos

#### 4.1.4 Modelo de Phong

Resultado final del modelo, definido por la *Intensidad del modelo de Phong*, se calcula como la suma de las intensidades expuestas anteriormente.

$$I_{Phong} = I_a + \sum_{\vec{l}_i \in L} \underbrace{\vec{n} \cdot_{[0,1]} (\vec{l}_i - \vec{p})}_{\text{Intensidad Difusa}} + h_k \underbrace{\left( o_j o \cdot_{[0,1]} \left( (\vec{l}_i - \vec{p}) \vee \vec{n} \right) \right)}_{\text{Intensidad Especular}}$$

En luces direccionales, el punto se considera estar en el infinito, sustituyéndose  $\vec{l}_i - \vec{p}$  por el vector director de la luz direccional  $\vec{d}_i$ . En algunas implementaciones, se utiliza para cada luz, un factor de atenuación que depende de la distancia de la superficie a la luz, esta función converge a cero en el infinito. En caso de utilizar luces direccionales, el valor que tomará será cero si  $f(d)$  no es constante, anulándose el aporte de *Intensidad Especular* de esta luz.

En particular, OpenGL hace uso de la siguiente función de atenuación:

$$f(d) = \frac{1}{k_q d^2 + k_l d + k_c}$$

Donde  $k_q, k_l, k_c \in \mathbb{R}_0^+$  son tres parámetros de atenuación <sup>4</sup>.

La intensidad es un factor, que multiplica al valor del píxel a mostrar, que es tomado del material de la figura??, aunque en estos ejemplos, devolveremos como color, el valor de la intensidad, resultando de una imagen en escala de grises.

---

<sup>4</sup> Cada parámetro corresponde en OpenGL con un parámetro de la clase `glLight`:  $k_q$  con `GL_QUADRATIC_ATTENUATION`; el factor  $k_l$  con `GL_LINEAR_ATTENUATION`; finalmente,  $k_c$  con el parámetro, `GL_CONSTANT_ATTENUATION`. <https://www.khronos.org/registry/OpenGL-Refpages/gl2.1/xhtml/glLight.xml>

Veamos un ejemplo práctico, vamos a añadir dos luces, una direccional desde uno de los laterales de la escena y otra luz, radial. Además vamos a utilizar el *homeomorfismo*  $h_3(x) = x^{2^3}$ .

```

1 // Homomorfismo
2 float h3(float h){return pow(h,pow(2.,3.));}
3 // Producto escalar normalizado positivo.
4 float dot01(vec3 a, vec3 b){
5     return max(dot(a,b)/(normalize(a)*
6                 normalize(b)), 0.0);
7 }
8 float ModeloIluminacion(
9     in vec3 direccion,
10    in vec3 p
11 ) {
12     // Calculamos la normal del punto.
13     vec3 normal = Normal(p);
14     // Emujamos de la superficie
15     p = p + normal * 0.1;
16     float intensidad = 0.0;
17     // Intensidad Ambiente Global
18     intensidad += 0.2;
19     // Intensidad de cada Luz
20     // Luz 1.
21     vec3 posicion_luz_1 = vec3(3.0, 3.0, 1.);
22     vec3 d_luz_1 = posicion_luz_1 - p;
23     vec3 dir_luz_1 = normalize(d_luz_1);
24     float dst_luz_1 = length(d_luz_1);
25     // Intensidad Difusa
26     intensidad += dot01(d_luz_1, normal);
27     // Intensidad Especular (Si no es
28     // direccional)
29     vec3 r_luz_1 = reflect(d_luz_1, normal);
30     intensidad += f_difusa(dst_luz_1) * h3(
31         dot01(r_luz_1, direccion));
32     // Luz 2. Direccional
33     vec3 dir_luz_2 = -normalize(vec3(1., 0.,
34                                 0.));
35     // Intensidad Difusa
36     intensidad += dot01(dir_luz_2, normal);
37     // La intensidad debe ser <= 1.
38     return min(intensidad, 1.0);
39 }
```

El enlace del ejemplo: <https://www.shadertoy.com/view/wtlBWr>

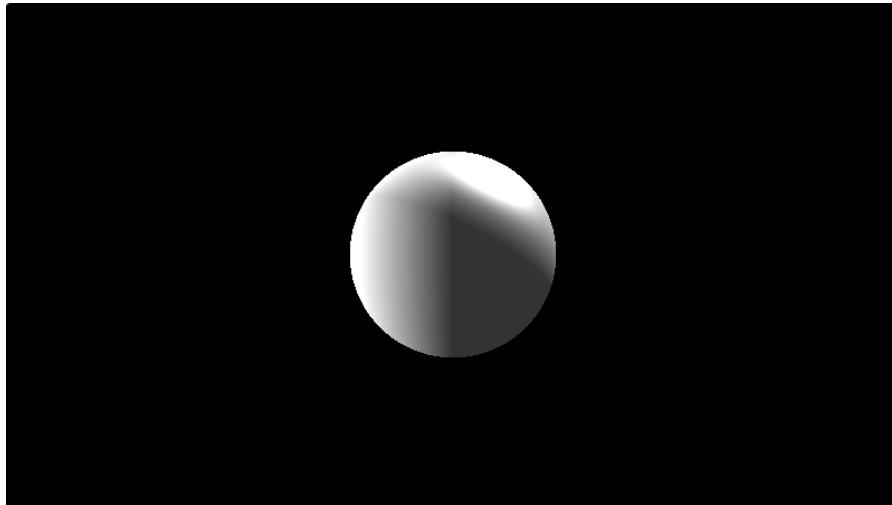


FIGURA 4.4: Modelo de Phong sobre la esfera con  $h_3(x) = x^8$ .

## 4.2 SOMBRAS

Vamos a ver la técnica más sencilla para calcular las sombras, es importante mencionar que hablaremos únicamente de la *umbra* de una la sombra, es decir, cuando la fuente de luz es oculta completamente por una superficie, haciendo que esa luz no aporte intensidad.

Dado un punto punto  $p$  sobre la superficie, lanzaremos otro rayo hacia la luz para ver si este es ocultado, en caso de trazar otro punto  $\vec{q}$  en esa dirección, la intensidad se mantendrá constante.

Al lanzar el rayo desde la una isosuperficie, las primeras iteraciones resultan de bolas pequeñas, afectando a la eficiencia del algoritmo, para solucionarlo, separaremos el punto  $\vec{p}$  de la superficie haciendo uso de la normal de la superficie y un factor de empuje  $k \in \mathbb{R}_0^+$ . Elegido de manera empírica según la escena, se aconseja  $k \in (0, 1]$ .

$$\vec{p}' = \vec{p} + \vec{n} \cdot k$$

Ahora el *Marcher* aceptará un tercer argumento, la distancia máxima recorrida, que anteriormente estaba fijado por la definición *MAXIMO*, de la condición de parada. El tercer argumento: en luces radiales será la distancia del punto a la luz; en luces direccionales, utilizaremos *MAXIMO*.

```

1 // Añadimos un tercer argumento.
2 float SphereMarching(
3     in vec3 ojo,
4     in vec3 direccion,
5     float distancia_maxima
6 ){
7     float distancia = 0.0;
8     for(int i = 0; i < PASOS; ++i){
9         vec3 rayo = ojo + direccion *
10            distancia;
11         float radio = escena_sdf(rayo);
12         if(radio < EPSILON){
13             return distancia;
14         }
15         distancia += radio;
16         // Ahora depende del tercer argumento
17         if(distancia>distancia_maxima)break;
18     }
19     return distancia_maxima;
}

```

Utilizaremos el modelo de iluminación descrito antes, con una luz radial y otra direccional. Además, utilizaremos un plano en donde proyectar la sombra. La luz direccional es perpendicular al plano.

```

1 // Escena plano + esfera
2 float escena_sdf(vec3 rayo){
3     vec3 pt = rayo - vec3(0., -0.3, 0.);
4     vec3 n = normalize(vec3(0., 1., 0.));
5     return min(dot(pt, n), length(rayo)-.2);
6 }
7
8 // Modelo Phong + Sombras
9 float ModeloIluminacion(vec3 direccion, vec3
10    p){
11     // Empujamos de la superficie el punto
12     p = p + Normal(p) * 0.1;
13     float intensidad = 0.0;
14     // Intensidad Ambiente Global
15     intensidad += 0.2;
16     // Luz 1. Radial
17     vec3 posicion_luz_1 = vec3(3.0, 3.0, 1.);
18     vec3 d_luz_1 = posicion_luz_1 - p;
19     vec3 dir_luz_1 = normalize(d_luz_1);

```

```

20     if(SphereMarching(p, dir_luz_1, dst_luz_1
21         ) >= dst_luz_1){
22         // Intensidad Difusa
23         intensidad += dot01(d_luz_1, normal);
24         // Intensidad Especular
25         vec3 r_luz_1 = reflect(d_luz_1,
26             normal);
27         intensidad += f_difusa(dst_luz_1) *
28             h3(dot01(r_luz_1, direccion));
29     }
30     // Luz 2. Direccional
31     vec3 dir_luz_2 =-normalize(vec3(1.,0.,0.))
32         );
33     if(SphereMarching(p, dir_luz_2, MAXIMO)
34         >= MAXIMO){
35         // Intensidad Difusa
36         intensidad +=dot01(dir_luz_2,normal);
37     }
38     return clamp(intensidad, 0.0, 1.0);
39 }

```

Enlace del ejemplo: <https://www.shadertoy.com/view/wtfBW8>

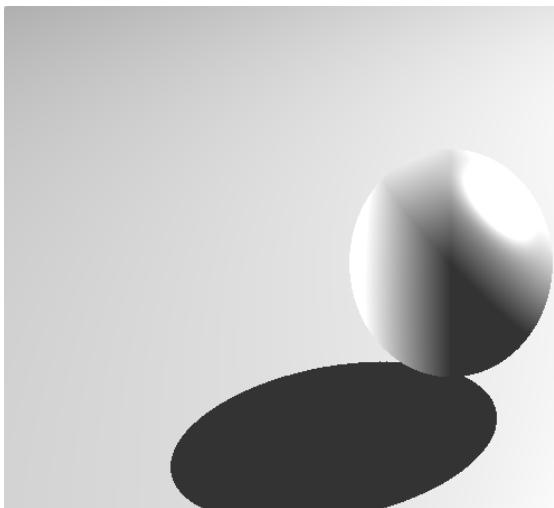


FIGURA 4.5: Modelo de iluminación y sombras sobre la escena definida.

Hemos visto como generar la umbra de una sombra, pero no la penumbra, la zona donde llega poca luz pero no está completamente a oscuras. Se han propuesto técnicas aproximadas ya que la penumbra exacta requiere de un alto coste computacional. Íñigo Quilez recopila varias de estas aproximaciones en una entrada en blog «Iquilezles - Soft Shadows in Raymarched SDFs - 2010»<sup>5</sup>

<sup>5</sup> <https://www.iquilezles.org/www/articles/rmshadows/rmshadows.htm>



## 5. FUNCIONES DE DISTANCIA CON SIGNO

---

Hemos presentado anteriormente las funciones de distancia con signo como definición matemática. Ahora debemos hacernos las siguientes preguntas: ¿Cómo podemos definir *funciones de distancia con signo*?; ¿qué tipo de funciones existen? y ¿qué operadores hay?.

**DEFINICIÓN 5.0.1.** Una función de distancia con signo, se dice exacta si está definido sobre la métrica euclídea.

Vamos a definir algunas *funciones de distancia con signo exactas*, que las denominaremos: «*primitivas*». En esta sección veremos primitivas sobre  $\mathbb{R}^2$  y  $\mathbb{R}^3$ , definiéndolas centradas en el origen. Veremos también operadores para aplicar transformaciones sobre estas, como por ejemplo la traslación y la rotación.

Se ha tomado como referencia algunas primitivas propuestas por el autor, *Iñigo Quilez* en su blog «*Iquilezles - Distance Functions*» <https://www.iquilezles.org/www/articles/distfunctions2d/distfunctions2d.htm>. Las demostraciones de las funciones presentadas en este trabajo son de propia cosecha.

### 5.1 PRIMITIVAS SOBRE $\mathbb{R}^2$

En primer lugar, veremos las primitivas en  $\mathbb{R}^2$  que serán fundamentales para definir las de una dimensión superior,  $\mathbb{R}^3$ . Las visualizaremos utilizando la plataforma *Shadertoy*. Vamos a crear un pequeño código que utilizaremos para presentar estas funciones:

1. El color rojo indicará el interior de la figura o distancias negativas.
2. El color azul, utilizado para el exterior de la figura o distancias positivas.
3. El color blanco para indicar que está sobre el isoperímetro con un margen de  $\epsilon = 0,01$ , para que esta sea visible.

Utilizaremos la parte fraccional, utilizando el operador «fract» de *GLSL*, de las distancias para crear líneas que representarán la métrica utilizada, si la distancia entre todos los pares consecutivos son iguales, podríamos asegurar que la métrica es constante y no hay deformaciones. El código utilizado:

```

1 #define EPSILON 0.01
2 void mainImage( out vec4 fragColor, in vec2
                 fragCoord )
3 {
4     vec2 p = (fragCoord-iResolution.xy * 0.5)
               /min(iResolution.x, iResolution.y);
5     // Aplicamos la FDS, f.
6     float d = f(p);
7     vec3 col = vec3(0.0);
8     // Estamos sobre el isoperímetro.
9     if(abs(d) < EPSILON){
10         col = vec3(1.0);
11     }else{
12         if(d < 0.0){ col.x = 1.0; }
13         else{ col.z = 1.0; }
14         // Número de repeticiones.
15         float k = 10.0;
16         col = col * (0.5 + 0.5 * (fract(abs(d
17             ) * k)));
18     }
19     fragColor = vec4(col, 1.);
}

```

### 5.1.1 Circunferencia exacta

La definición de esta figura se basa en la distancia desde cualquier punto  $\vec{p}$  hasta  $(0, 0)$ , el teorema de pitágoras, el módulo del vector  $\|\vec{p}\|$  crea una función de distancia positiva con un isoperímetro  $S = \{(0, 0)\}$ . Si a las distancias le restamos el radio  $r$ , anularemos aquellos puntos a distancia  $r$ . Cuando el vector está en el interior de la circunferencia, es decir, el módulo del vector es inferior al radio,  $\|\vec{p}\| < r \rightarrow \|\vec{p}\| - r < 0$ , haciendo la distancia negativa. En caso de que el módulo del vector coincida con el radio,  $\|\vec{p}\| = r \rightarrow \|\vec{p}\| - r = 0$ , es decir, estamos sobre el isoperímetro y finalmente, cuando módulo es mayor al radio,  $\|\vec{p}\| > r \rightarrow \|\vec{p}\| - r > 0$  la distancia será positiva, en el exterior.

```

1 // Circunsferencia Exacta
2 float SDFCircunferencia(vec2 p, float r){
3     return length(p) - r;
4 }

```

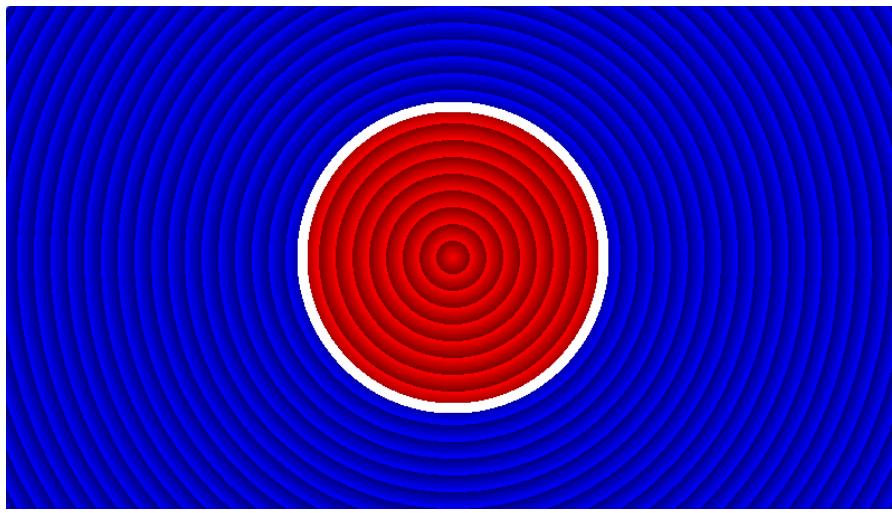


FIGURA 5.1: Circunsferencia FDS radio 0.3

El ejemplo: <https://www.shadertoy.com/view/WtXfRj>

### 5.1.2 Rectángulo exacto

Para calcular de forma exacta esta función, vamos a definir dos operadores sobre los vectores:

$$|(x_0, x_1, \dots, x_n)| = (|x_0|, |x_1|, \dots, |x_n|)$$

$$\max((x_0, x_1, \dots, x_n), k) = (\max(x_0, k), \max(x_1, k), \dots, \max(x_n, k))$$

El operador valor absoluto es útil ya que reduce el problema a un único cuadrante, que representa un cuarto de la figura original. Las medidas serán la mitad de la original  $\vec{s}$ , en forma vectorial:

$$\vec{s}' = (w', h') = \left( \frac{w}{2}, \frac{h}{2} \right) = \frac{\vec{s}}{2}$$

Situándonos en el primer cuadrante, dividimos el problema en 4 subproblemas:

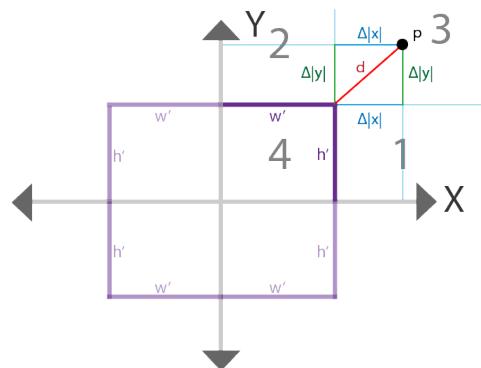


FIGURA 5.2: División del rectángulo en 4 regiones o subproblemas.

Para cada subproblema, vamos a preservar la métrica euclídea, posicionando  $|\vec{p}| = |(x, y)| = (|x|, |y|)$  en cada una de las regiones.

1. **Región 1.** Se trata de encontrar la distancia al lado derecho, una región positiva:  $\Delta|x| = \max(|x| - w', 0)$ .
2. **Región 2.** De manera similar al anterior, la distancia al lado superior:  $\Delta|y| = \max(|y| - h', 0)$ .
3. **Región 3.** La distancia a la esquina del rectángulo, también positiva, utilizando el teorema de pitágoras:  $d = \sqrt{(\Delta|x|)^2 + (\Delta|y|)^2}$ .
4. **Región 4.** La distancia en esta región es la máxima de las distancias negativas, al estar en el interior, a cada uno de los lados. Esta deberá anularse cuando nos encontremos en el exterior, definimos un nuevo operador:

$$\text{argmax}(\vec{a}) = \min(\max(\vec{a}_x, \vec{a}_y), 0)$$

Haciendo uso de este, la distancia interior será:  $\text{argmax}(|\vec{p}| - \vec{s}')$ .

Podríamos crear una función a trozos y utilizar condicionales para cada región, pero queremos que además de ser exacto, también sea eficiente, en general, las condiciones suelen ser lentas. Vamos a mirar la relación entre las diferentes regiones. Por ejemplo, observamos que la **Región 3** contiene en su ecuación a la **Región 1 y 2**, veamos casos particulares: cuando  $\Delta|x| = 0 \rightarrow \sqrt{(\Delta|y|)^2} = \Delta|y|$ , que concuerda con la **Región 2**, haciendo que la **Región 2 y 3** queden unificadas; Para la **Región 1 y 3**, de manera equivalente, tenemos  $\Delta|y| = 0 \rightarrow \sqrt{(\Delta|x|)^2} = \Delta|x|$ , vemos que la **Región 3** recoge los casos para las **Regiones 1, 2 y 3**, por lo que la distancia en el exterior será

$$d_{\text{exterior}} = \left\| \max \left( |\vec{p}| - \vec{s}', 0 \right) \right\|$$

Para terminar, observamos que estamos en la **Región 4**, si  $\Delta|x| < 0$  y  $\Delta|y| < 0$ , simplificándose  $d_{\text{exterior}} = 0$ , por lo que bastará sumar la distancia de esta región para obtener la ecuación exacta:

$$\text{SDFRectangulo}_{\vec{s}'}(\vec{p}) = \left\| \max \left( |\vec{p}| - \vec{s}', 0 \right) \right\| + \text{argmax}(|\vec{p}| - \vec{s}')$$

```

1 // Rectángulo Exacto
2 float SDFRectangulo(vec2 p, vec2 s){
3     vec2 a = abs(p) - s;
4     // Exterior
5     float extr = length(max(a, 0.0));
6     // Interior
7     float int = min(max(a.x, a.y), 0.0);
8     return extr + int;
9 }
```

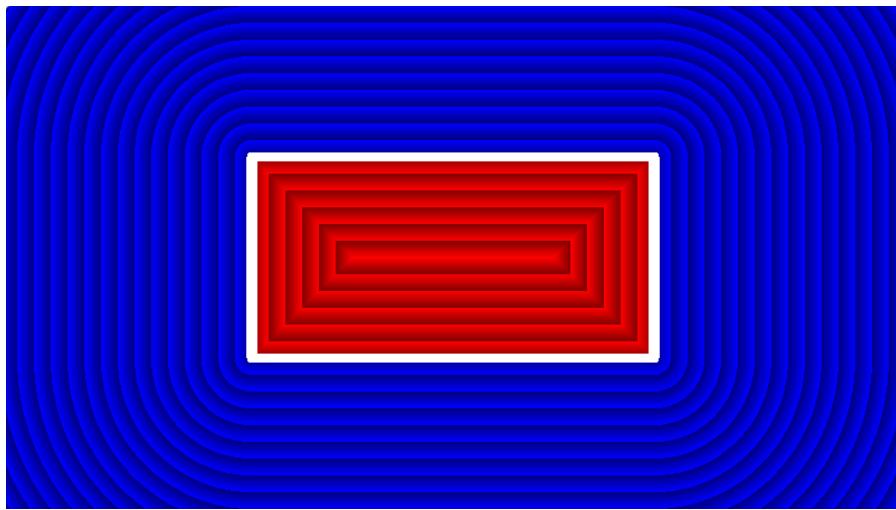


FIGURA 5.3: Rectángulo FDS de dimensiones  $\vec{s}' = (0.4, 0.2)$

Enlace del ejemplo: <https://www.shadertoy.com/view/3lXBR2>

### 5.1.3 Recta exacta

Se puede definir una recta de forma algebráica utilizando un punto y un vector director  $\vec{n}$ , para simplificar, tomaremos como punto el origen. Utilizaremos el operador de proyección<sup>1</sup> para calcular el punto más cercano a la recta. El operador de proyección se define como:

$$\text{proy}_{\vec{n}} \vec{p} = \left( \frac{\vec{p} \cdot \vec{n}}{|\vec{n}|} \right) \frac{\vec{n}}{|\vec{n}|} = \left( \frac{\vec{p} \cdot \vec{n}}{|\vec{n}|^2} \right) \vec{n} = \left( \frac{\vec{p} \cdot \vec{n}}{\vec{n} \cdot \vec{n}} \right) \vec{n}$$

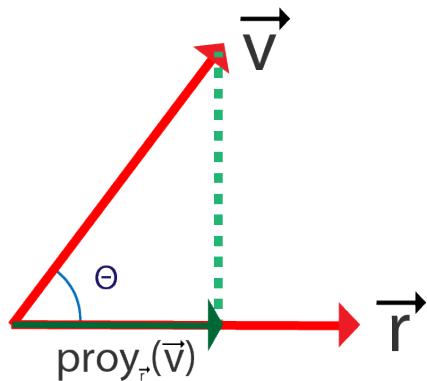


FIGURA 5.4: Proyección  $\vec{v}$  sobre  $\vec{r}$

<sup>1</sup> Este operador se demuestra a partir de la definición del producto escalar. Un ejemplo lo encontramos en el siguiente enlace: [https://en.m.wikibooks.org/wiki/Linear\\_Algebra/Orthogonal\\_Projection\\_Onto\\_a\\_Line](https://en.m.wikibooks.org/wiki/Linear_Algebra/Orthogonal_Projection_Onto_a_Line)

Podemos generalizar este problema para cualquier recta formada por los puntos  $\vec{a}, \vec{b} \in \mathbb{R}^2$ , cuyo vector director es  $\vec{n} = \vec{b} - \vec{a}$ . Utilizaremos uno de los dos puntos como eje de coordenadas, restando a cada punto uno de los dos vectores, por ejemplo, el vector  $\vec{a}$  y finalmente, el resultado lo trasladamos  $\vec{a}$ . La distancia, que siempre es positiva, será la distancia del punto  $\vec{p}$  a la proyección:

$$SDFRecta_{\vec{a}, \vec{b}}(\vec{p}) = \|(\vec{p} - \vec{a}) - \text{proy}_{\vec{p}-\vec{a}}(\vec{b} - \vec{a})\|$$

```

1 // Operador Proyección a sobre b
2 vec2 proy(in vec2 a, in vec2 b){
3     return b * dot(b, a) / dot(b, b);
4 }
5 // Línea Exacto
6 float SDFRecta(vec2 p, vec2 a, vec2 b){
7     vec2 v = p - a;
8     vec2 w = b - a;
9     return length(v - proy(v, w));
10 }
```

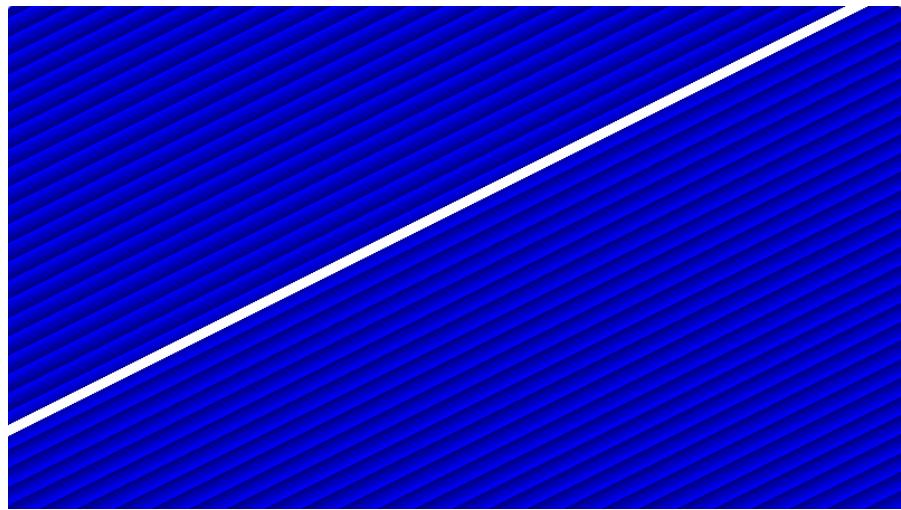


FIGURA 5.5: FDS Recta que pasa por  $\vec{a} = (0.2, 0.2), \vec{b} = (0.0, 0.1)$

Enlace del ejemplo: <https://www.shadertoy.com/view/ttlBzX>

#### 5.1.4 Segmento exacto

Se trata de un caso particular del operador proyección entre dos vectores  $\vec{a}, \vec{b}$ .

$$\text{proy}_{\vec{b}}\vec{a} = \left( \frac{\vec{a} \cdot \vec{b}}{\vec{b} \cdot \vec{b}} \right) \vec{b}$$

Observamos que  $\frac{\vec{a} \cdot \vec{b}}{\vec{b} \cdot \vec{b}}$  es el factor de proyección que está en  $\mathbb{R}$ .

Cuando este factor es cero, la proyección será el vector  $(0, 0)$ , en caso de que el factor tome el valor de 1, la proyección será el vector  $\vec{b}$ . Por lo que el segmento está formado por todos los puntos que toma el factor en el intervalo  $[0, 1]$ :

$$\text{proy}[0,1]_{\vec{b}} \vec{a} = \max \left( \min \left( \frac{\vec{a} \cdot \vec{b}}{\vec{b} \cdot \vec{b}}, 0 \right), 1 \right) \vec{b}$$

```

1 // Operador Proyección [0,1] a sobre b
2 vec2 proy01(in vec2 a, in vec2 b){
3     return b * clamp(dot(b, a) / dot(b, b),
4         0., 1.);
5 // Segmento Exacto
6 float SDFSegmento(vec2 p, vec2 a, vec2 b){
7     vec2 v = p - a;
8     vec2 w = b - a;
9     return length(v - proy01(v, w));
10 }
```

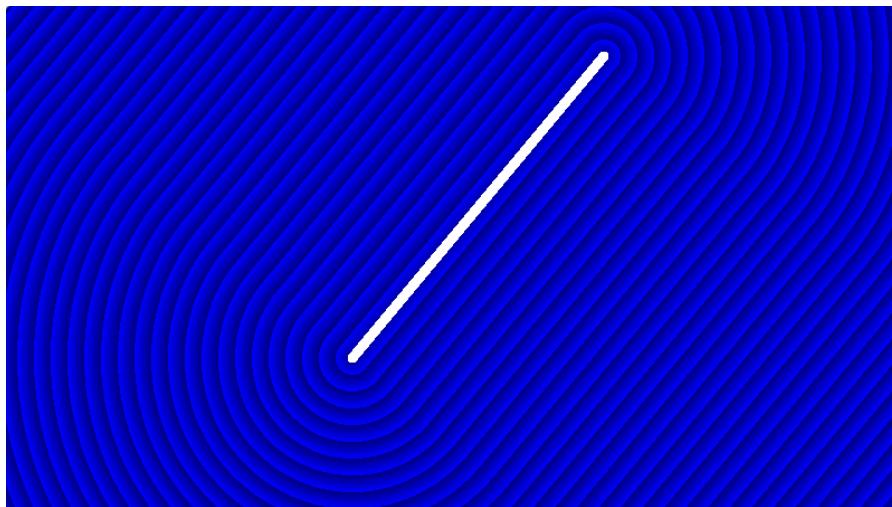


FIGURA 5.6: FDS Recta que pasa por  $\vec{a} = (-0.2, -0.2)$ ,  $\vec{b} = (0.3, 0.4)$

Enlace del ejemplo: <https://www.shadertoy.com/view/wllBR1>

Existen infinitas *funciones de distancia con signo exactas* y aún quedan muchas por definir. Vamos a presentar operadores que nos van a ayudar a transformar estas funciones, resultando de otras que pueden o no preservar la *exactitud*.

Algunas *funciones de distancia con signo* no presentadas aquí y que podemos encontrar en el blog de *Íñigo Quilez*, pueden ser demostradas utilizando las propiedades y operadores vistos en los apartados anteriores, por ejemplo, la definición exacta del triángulo hace uso de dos segmentos para dos de sus lados conectados por una arista y las coordenadas baricéntricas<sup>2</sup>.

## 5.2 OPERADORES SOBRE $\mathbb{R}^2$

Vamos a ver operadores imprescindibles para poder manipular nuestra escena, además de poder crear nuevas *funciones de distancia con signo exactas*. Estos operadores pueden preservar o no la exactitud de nuestras funciones, aquellos operadores que la preserven, los llamaremos *isometrías*.

**DEFINICIÓN 5.2.1.** Una aplicación  $g$  sobre dos espacios métricos se dice que es *isométrica* si se conservan la distancia entre todos los puntos antes y después de su aplicación.

$$\forall \vec{v}, \vec{w} \in \mathbb{R}^2, d(\vec{v}, \vec{w}) = d(g(\vec{v}), g(\vec{w}))$$

En secciones anteriores hemos visto por qué vamos a trabajar sobre la métrica euclídea, esto implicará que la función de distancia  $d$ , sea  $d(\vec{v}, \vec{w}) = d((x, y), (z, w)) = \|\vec{v} - \vec{w}\| = \sqrt{(x - z)^2 + (y - w)^2}$ . Veremos tres *isometrías*: la traslación, la rotación y la simetría sobre una recta.

### 5.2.1 Operador de traslación

Definimos la traslación como el operador donde un punto  $\vec{p}$  es desplazado por un vector  $\vec{t}$ , preservando las características:

$$\text{traslacion}(\vec{p}, \vec{t}) = \vec{p} \pm \vec{t}$$

Vamos a demostrar que la traslación es una isometría,  $\forall \vec{v}, \vec{w} \in \mathbb{R}^2$ :

$$d(f(\vec{v}), f(\vec{w})) = d(\vec{v} \pm \vec{t}, \vec{w} \pm \vec{t}) = \|(\vec{v} \pm \vec{t}) - (\vec{w} \pm \vec{t})\| = \|\vec{v} - \vec{w}\| = d(\vec{v}, \vec{w})$$

Este es uno de los motivos por el cual se ha definido las *funciones de distancia con signo* sobre el origen. Ahora, podemos trasladar una función a cualquier parte de nuestra escena, en código:

---

2 El sistema de coordenadas baricéntrico se genera a partir del baricentro, es decir, el punto de corte de las tres rectas que pasan por el centro de un lado y por el vértice opuesto. Podemos encontrar la demostración de la distancia en el siguiente enlace: <https://mathoverflow.net/a/36669>

```

1 float escena_sdf(vec2 p){
2     // Trasladamos el vector p hacia 0.1
      // izquierda y 0.2 a la derecha.
3     vec2 pt = p - vec2(0.1, 0.2);
4     return SDFCircunsferencia(pt, 0.3);
5 }
```

Hemos realizado una resta en vez de una suma para trasladarlo a la posición deseada. Aunque parece contraintuitivo, debemos imaginarlo como si nuestros ojos estuvieran sobre el plano, cambiado las orientaciones.

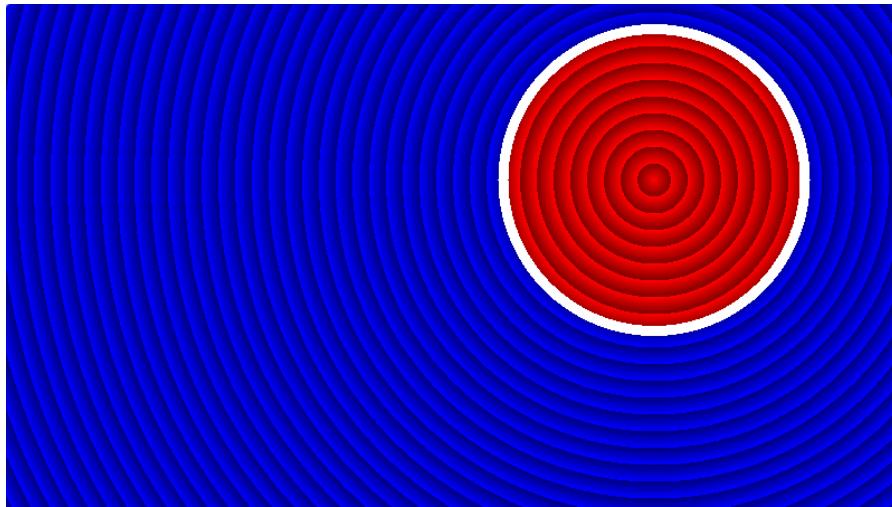


FIGURA 5.7: Traslación FDS  $\vec{t} = (0.1, 0.2)$

Enlace del ejemplo: <https://www.shadertoy.com/view/ttxfWS>

### 5.2.2 Operador de rotación

El segundo operador que vamos a presentar es el operador de rotación. Se define como la matriz de rotación<sup>3</sup>, donde  $\alpha$  es el ángulo de rotación en radianes y es una matriz ortogonal<sup>4</sup>:

$$\text{rot}(\alpha) = \begin{pmatrix} +\cos(\alpha) & -\sin(\alpha) \\ +\sin(\alpha) & +\cos(\alpha) \end{pmatrix}$$

<sup>3</sup> Esta matriz es definida por el mapeo de coordenadas polares y la linearidad. La demostración la encontramos en el siguiente enlace: <https://math.stackexchange.com/a/53181>

<sup>4</sup> Una matriz se dice ortogonal, si y solo si,  $A \cdot A^t = I$ . La demostración de que la matriz de rotación es ortogonal: <https://math.stackexchange.com/a/2471175>

que aplicada sobre un vector  $\vec{p} = \begin{pmatrix} x \\ y \end{pmatrix}$ ,

$$\text{rotacion}_\alpha(\vec{p}) = \vec{p} \cdot \text{rot}(\alpha) = \begin{pmatrix} x \\ y \end{pmatrix}^t \cdot \begin{pmatrix} +\cos(\alpha) & -\sin(\alpha) \\ +\sin(\alpha) & +\cos(\alpha) \end{pmatrix}$$

Resultando,

$$\text{rotacion}_\alpha(\vec{p}) = \begin{pmatrix} +x\cos(\alpha) + y\sin(\alpha) \\ -x\sin(\alpha) + y\cos(\alpha) \end{pmatrix}$$

Veamos que el operador también es una *isometría*,  $\forall \vec{v}, \vec{w} \in \mathbb{R}^2$ :

$$d(f(\vec{v}), f(\vec{w})) = d(\vec{v} \cdot \text{rot}(\alpha), \vec{w} \cdot \text{rot}(\alpha)) =$$

$$\|\vec{v} \cdot \text{rot}(\alpha) - \vec{w} \cdot \text{rot}(\alpha)\| = \|(\vec{v} - \vec{w}) \cdot \text{rot}(\alpha)\|$$

Como  $\text{rot}(\alpha)$  es ortogonal:  $\|A \cdot \text{rot}(\alpha)\| = \|A\|$ .<sup>5</sup>

$$d(f(\vec{v}), f(\vec{w})) = \|(\vec{v} - \vec{w}) \cdot \text{rot}(\alpha)\| = \|\vec{v} - \vec{w}\| = d(\vec{v}, \vec{w})$$

En código:

```

1 #define PI 3.1415
2 mat2 rot(float a){
3     return mat2(
4         +cos(a), -sin(a),
5         +sin(a), +cos(a)
6     );
7 }
8 // Escena
9 float escena_sdf(vec2 p){
10    // Rotacion del el vector p 45 grados o
11    // pi / 4 radianes.
12    vec2 pr = p * rot(45. * PI / 180.);
13    return SDFRectangulo(pr, vec2(0.3));
14 }
```

---

<sup>5</sup> La demostración de esta propiedad requiere un alto conocimiento matemático, como curiosidad, encontramos la demostración en el siguiente enlace: <https://math.stackexchange.com/a/2245861>

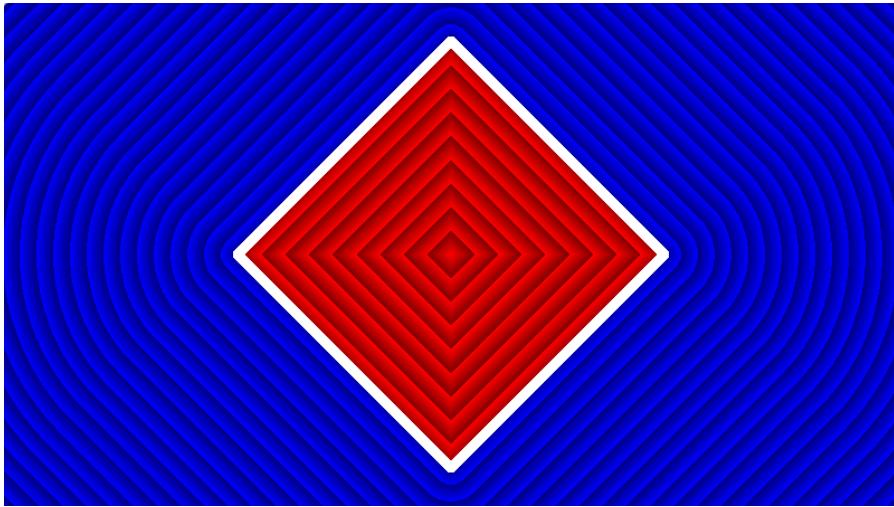


FIGURA 5.8: Rectángulo  $\vec{s} = 0.3$  FDS con rotación  $\alpha = \frac{\pi}{4}$

Enlace del ejemplo: <https://www.shadertoy.com/view/w1XfDS>

### 5.2.3 Operador de simetría

La simetría traslada los puntos que están sobre un lado de la recta hasta el otro lado tal que estos equidistán.

Dado un vector director  $\vec{n} \in \mathbb{R}^2$  de una recta que pasa por el  $(0, 0)$  y utilizando el operador de proyección de cualquier punto  $\vec{p} \in \mathbb{R}^2$  sobre  $\vec{n}$ . La simetría consiste en trasladar  $\vec{p}$  hacia el otro lado de la recta en dirección al punto proyectado.

$$\text{simetria}_{\vec{n}}(\vec{p}) = \vec{p} + 2(\text{proy}_{\vec{n}}(\vec{p}) - \vec{p})$$

Para cualquier recta formada por dos puntos  $\vec{a}, \vec{b}$ , el operador de simetría será:

$$\text{simetria}_{\vec{a}, \vec{b}}(\vec{p}) = \vec{a} + \text{simetria}_{\vec{b} - \vec{a}}(\vec{p} - \vec{a})$$

Veamos algunos casos particulares, los definidos sobre los ejes:

Sobre el eje  $X$ ,  $\vec{n} = (1, 0)$ , definimos el operador de simetría sobre el eje x como

$$\text{simetria}_X(\vec{p}) = \text{simetria}_{\vec{n}}(\vec{p}) = (-x, y)$$

De manera equivalente para el eje  $Y$ ,  $\vec{n} = (0, 1)$ :

$$\text{simetria}_Y(\vec{p}) = \text{simetria}_{\vec{n}}(\vec{p}) = (x, -y)$$

Se deja una cuestión abierta sobre estos dos últimos operadores de simetría, ¿qué ocurriría si «dobláramos» el eje y?, es decir:

$$\text{simetria}_{|Y|}(\vec{p}) = \text{simetria}_{\vec{n}}(\vec{p}) = (x, -|y|)$$

Veamos la definición en código,

```

1 // Simetría sobre el origen
2 vec2 simetria(vec2 n, vec2 p){
3     return p + 2. * (proy(p, n) - p);
4 }
5 // Sobre cualquier recta genérica formada que
6 // pasa por a y b.
7 vec2 simetria(vec2 p, vec2 a, vec2 b){
8     return a + simetria(b - a, p - a);
9 }
```

Veamos un ejemplo en práctica, sobre la recta definida en Figura ?? Recta exacta, con la función de distancia con signo de Figura ?? Segmento exacto.

```

1 float escena_sdf(vec2 p){
2     // Recta Simetría
3     vec2 a = vec2(0.2, 0.2);
4     vec2 b = vec2(0.0, 0.1);
5     // Simetría
6     vec2 ps = simetria(p, a, b);
7
8     return SDFSegmento(
9         ps,
10        vec2(-0.2, -0.2),
11        vec2(0.3, 0.4)
12    );
13 }
```

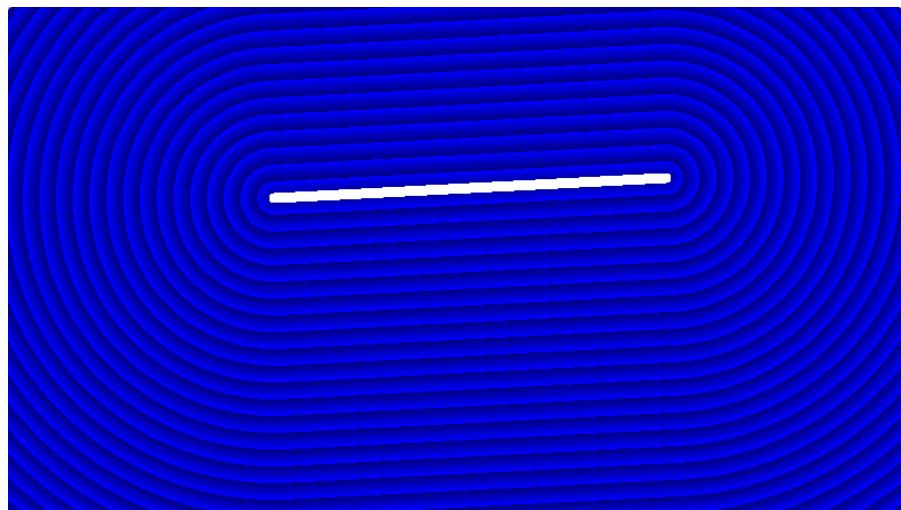


FIGURA 5.9: Simetría FDS ejemplos anteriores

Enlace del ejemplo: <https://www.shadertoy.com/view/3lsBWS>

Veamos ahora algunos operadores entre *funciones de distancia con signo*. Estos operadores van a ser vitales para crear escenas complejas, ya que van a permitir agregar o sustraer *funciones de distancia con signo exactas*.

#### 5.2.4 Operador de agregación

Ya vista la función de traslación, ahora nos puede ser útil ver como agregar dos *funciones de distancia con signo*. El resultado de este operador será otra *función de distancia con signo* formado por la distancia menor en cada punto de las dos funciones a agregar. Esta descripción se define como:

$$\text{Agregacion}(\vec{p}, f, g) = \min(f(\vec{p}), g(\vec{p}))$$

Utilizaremos la función «*min*» que está definida en el lenguaje *GLSL*. Veamos un ejemplo, utilizaremos los dos últimos ejemplos de *isometrías*, la traslación [Figura ?? Operador de traslación](#) y la rotación [Figura ?? Operador de rotación](#):

```

1 // Escena
2 float escena_sdf(vec2 p){
3     // Rotamos el rectángulo 45 grados - f
4     vec2 pr = p * rot(PI / 180. * 45.0);
5     // Trasladamos el rectangulo hacia 0.1 -
6     // g izquierda y 0.2 a la derecha.
7     vec2 pt = p - vec2(0.4, 0.15);
8     // Unión de dos FDS
9     return min(
10         SDFRectangulo(pr, vec2(0.3)), // f
11         SDFCircunferencia(pt, 0.3)   // g
12     );
}

```

El resultado de la adición es el siguiente:

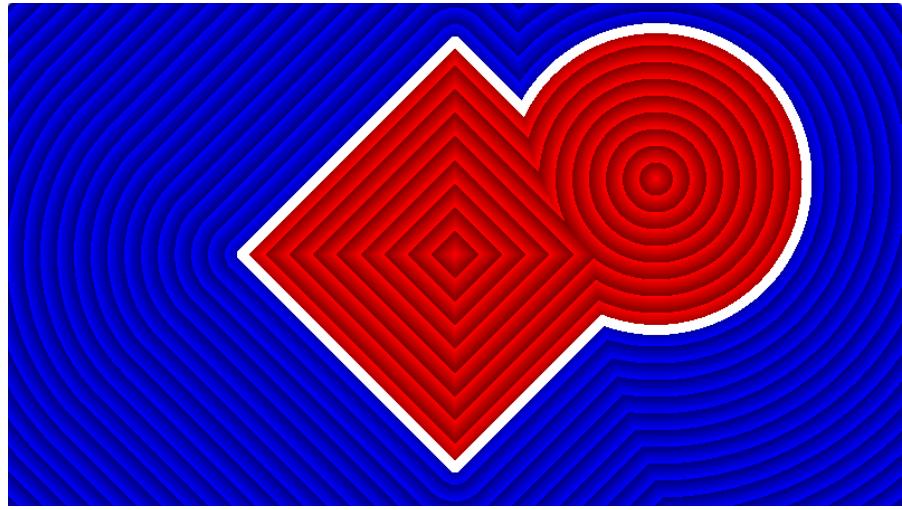


FIGURA 5.10: Adición de dos FDS de ejemplos anteriores

Enlace del ejemplo: <https://www.shadertoy.com/view/WlsBDS>

### 5.2.5 Operador de substracción

Antes hemos visto que el operador «mín» que combina dos *FDS*, podemos imaginarnos que hace el operador «máx». Este devuelve la distancia más lejana entre las dos, es decir, devuelve la intersección de ambos, ya que si alguna distancia es positiva, devolverá siempre la positiva, en caso de ambas ser negativas, devolverá la menor.

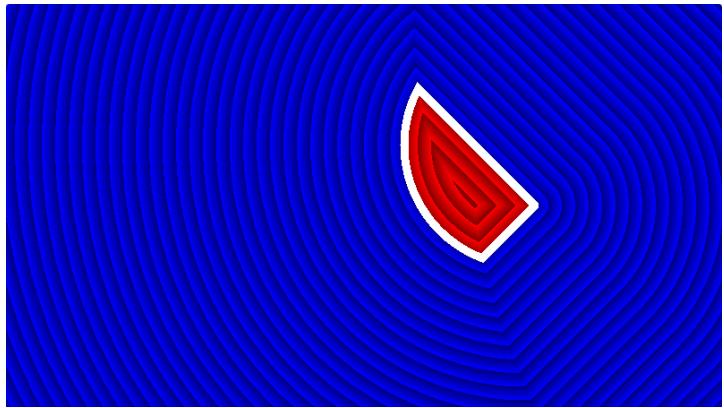


FIGURA 5.11: Intersección de las dos FDS de ejemplos anteriores

Enlace del ejemplo: <https://www.shadertoy.com/view/3llfDS>

Vamos a crear el operador de substracción, utilizando el de intersección, pero antes, es fácil observar que podemos cambiar el interior por el exterior, que llamaríamos *inversión de una función de distancia con signo*. Bastaría con cambiar el signo, multiplicando por  $-1$ .

$$\text{Inversión}(\vec{p}, f) = -f(\vec{p})$$

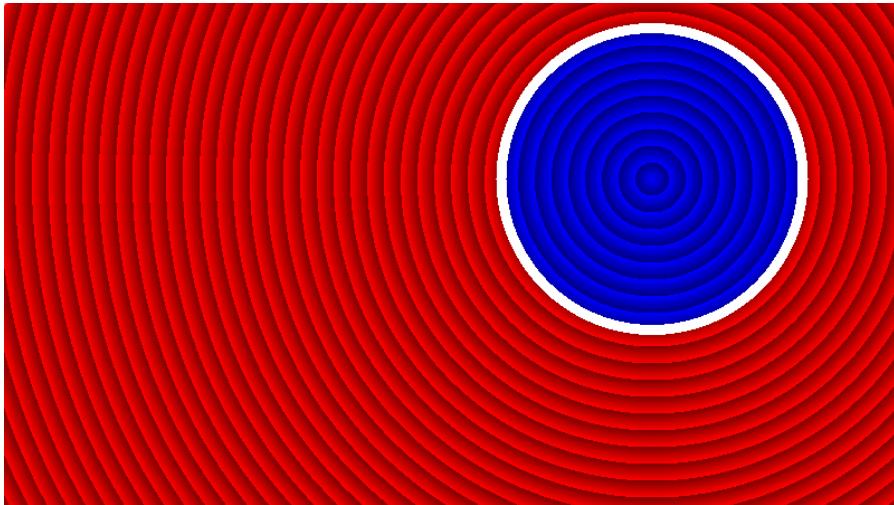


FIGURA 5.12: Interior por Exterior Circunsferencia FDS

Enlace del ejemplo: <https://www.shadertoy.com/view/WlsfDS>

Ahora, las distancias negativas representan el exterior de la figura anterior. Esto junto a la definición de intersección, se eliminará de una figura el interior de la figura anterior. Definimos la substracción de una función de distancia con signo  $f$  otra  $g$ , tal que:

$$\text{substraccion}(\vec{p}, f, g) = \max(f(\vec{p}), -g(\vec{p}))$$

Un ejemplo en código:

```

1 float escena_sdf(vec2 p){
2     // Rotamos el rectángulo 45 grados
3     vec2 pr = p * rot(PI / 180. * 45.0);
4
5     // Trasladamos la circunferencia
6     vec2 pt = p - vec2(0.4, 0.15);
7     // Sustraemos Una circunsferencia de un
8     // rectángulo.
9     return max(
10         SDFRectangulo(pr, vec2(0.3)),
11         -SDFCircunsferencia(pt, 0.3)
12     );
13 }
```

El resultado de la ejecución del código:

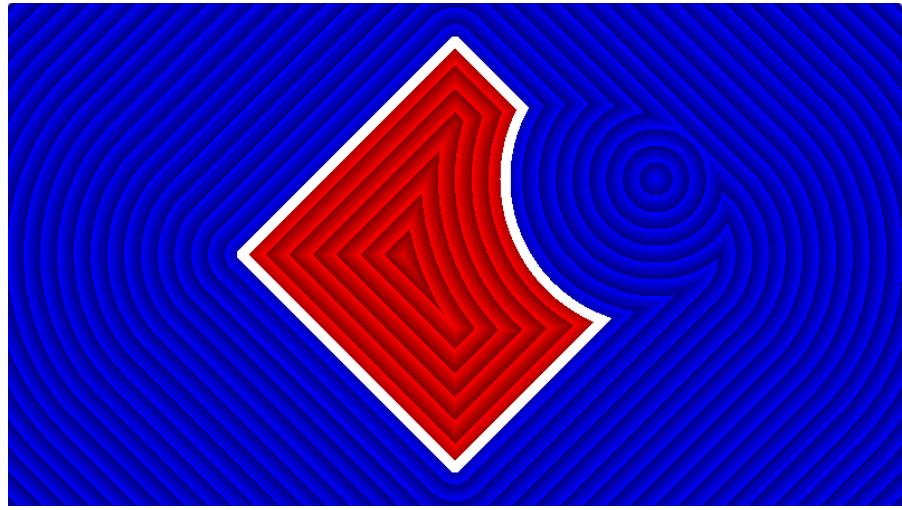


FIGURA 5.13: Substracción de una circunsferencia a un rectangulo FDS

Enlace del ejemplo: <https://www.shadertoy.com/view/W11BWB>

Vamos a ver otro tipo de transformaciones, las que manipulan el espacio, esto puede hacer que el resultado de las funciones no sean exacto o que debamos hacer una transformación que las transforme en exactas, por ejemplo, el escalado.

#### 5.2.6 Operador de escalado

Veamos como escalar una función de distancia con signo y como podemos solucionar esta deformación. Supongamos que el escalado fuera una isometría, es decir:

$$g(\vec{p}) = \vec{p} \cdot \frac{1}{k}, k \in \mathbb{R}_0^+$$

Donde  $k$  es el factor de escalado, veamos que no es una isometría y como lo solucionamos,  $\vec{v}, \vec{w} \in \mathbb{R}^2$ .

$$d(g(\vec{v}), g(\vec{w})) = d\left(\vec{v} \cdot \frac{1}{k}, \vec{w} \cdot \frac{1}{k}\right) = \left\| \vec{v} \cdot \frac{1}{k} - \vec{w} \cdot \frac{1}{k} \right\| = \left\| (\vec{v} - \vec{w}) \cdot \frac{1}{k} \right\|$$

Vemos que la distancia transformada es proporcional a la distancia inicial con factor  $\frac{1}{k}$ . Para solucionar esto, multiplicamos la distancia por el factor  $k$ , esto hará que el factor se cancele, consiguiendo así una isometría.

Según los valores que tome  $k$ : cuando  $k < 1$ , estamos reduciendo el tamaño; si  $k = 1$  el tamaño original se preserva y finalmente, con  $k > 1$  agrandamos.

Veamos un ejemplo, por ejemplo, escalamos el ejemplo anterior con un factor de  $k = 0.5$ , reduciendo a la mitad:

```

1 float escena_sdf(vec2 p){
2     // Factor de escalado
3     float k = 0.5;
4     // Escalamos todas las figuras
5     p = p / k;
6     // Rotamos el rectángulo
7     vec2 pr = p * rot(PI / 180. * 45.0);
8     // Trasladamos la circunferencia
9     vec2 pt = p - vec2(0.4, 0.15);
10    float d = max(
11        SDFRectangulo(pr, vec2(0.3)),
12        -SDFCircunsferencia(pt, 0.3)
13    );
14    // Multiplicamos por la distancia para
15    // hacerlo una isometría.
16    return d * k;
17 }
```

El resultado del escalado es el siguiente:

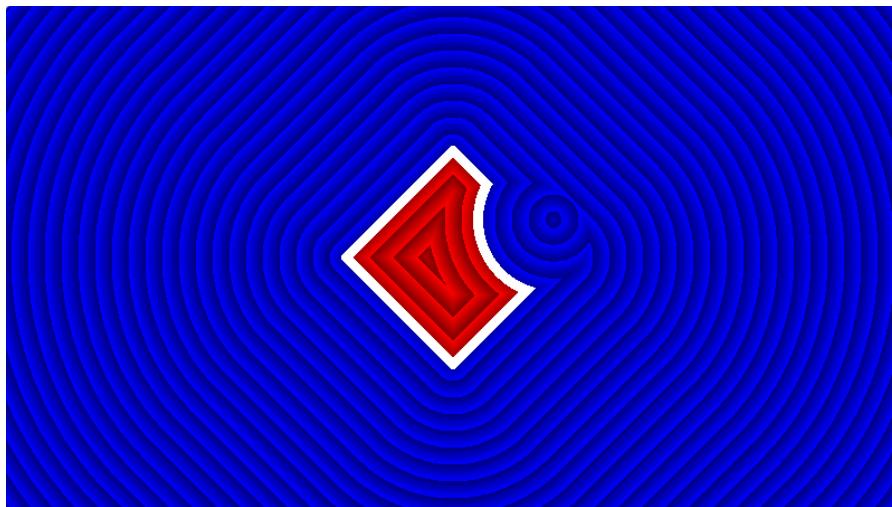


FIGURA 5.14: Ejemplo anterior escalado con  $k = 0.5$

Enlace del ejemplo:<https://www.shadertoy.com/view/WlsBDX>

### 5.2.7 Operador de deformación sin exactitud

Como se ha comentado antes, hay deformaciones que no preservan la métrica y por tanto las funciones no son exactas. En general, consiste en aplicar una función no *isomórfica* al vector  $\vec{p}$ .

Si  $g : \mathbb{R}^2 \rightarrow \mathbb{R}^2$  es una función no *isométrica*:

$$\forall \vec{x}, \vec{y} \in \mathbb{R}^2, d(g(x), g(y)) \neq d(x, y)$$

Haciendo que el resultado no sea una *función de distancia con signo exacta*. Se recomienda utilizar aplicaciones  $g$  que sean continuas y derivables, para así evitar transformaciones fuertes o discontinuidades. Por ejemplo,

$$g(x, y) = (x * \cos(y \cdot \pi), y * \sin(y \cdot \pi))$$

En código,

```

1 float sdf(vec2 p){
2     // Vec2 No Isometría
3     vec2 pn = vec2(
4         p.x * cos(p.y * PI),
5         p.y * sin(p.y * PI)
6     );
7     return SDFCircunferencia(pn, 0.1);
8 }
```

El resultado de la deformación:

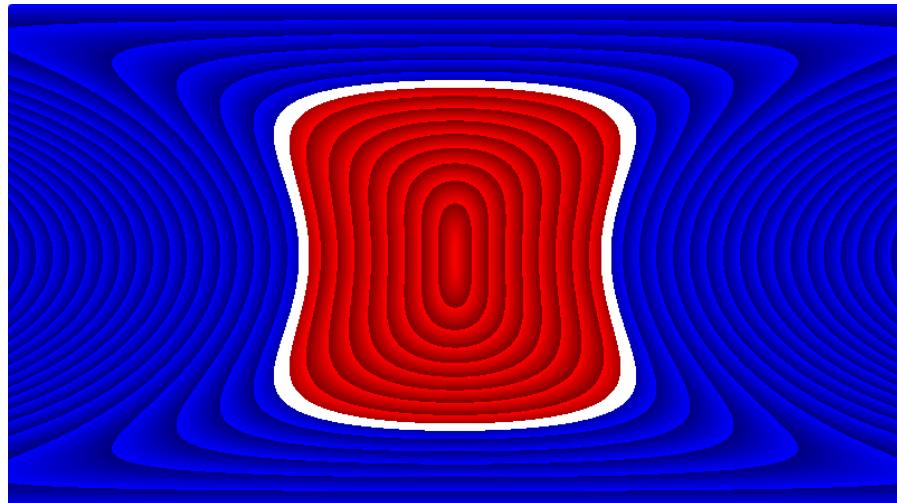


FIGURA 5.15: Deformación del espacio de una circunsferencia FDS

Enlace del ejemplo:<https://www.shadertoy.com/view/3lsfWj>

Las aplicaciones explicadas anteriormente, en general, se extienden a  $\mathbb{R}^3$ . Por ejemplo, vamos a ver como primitivas generalizan a las de una dimensión superior.

### 5.3 PRIMITIVAS EN $\mathbb{R}^3$

#### 5.3.1 Esfera exacta

Esta función es generalizada de la ecuación en 2D, la idea es la misma, aquellas distancias que antes eran positivas con valor  $r$ , quedan anuladas al restarles  $r$  y por tanto, forman una *isosuperficie*. En código,

```

1 // Esfera R3
2 float SDFEsfera(vec3 p, float r){
3     return length(p) - r;
4 }
```

Utilizando el *Marcher* 3 y el *modelo de iluminación* 4 definido, el resultado es el observado en la [Figura ?? Modelo de Phong](#).

### 5.3.2 Prisma rectangular exacto

Este es generalizado del *Rectángulo exacto* en  $\mathbb{R}^2$ , el valor absoluto situará las coordenadas en el cuadrante positivo, de los ocho presentes. Las medidas utilizadas serán la mitad, es decir,  $\vec{s}' = \frac{\vec{s}}{2}$ . Aunque no vamos a ver la demostración, una idea de esta es similar a la utilizada para demostrar el rectángulo, cada región exterior de cada cara es anulada, el interior se utiliza la distancia al lado más próximo y en la arista, la distancia euclídea. El resultado.

```

1 // Prisma R3
2 float SDFPrisma(vec3 p, vec3 s){
3     vec3 pa = abs(p) - s;
4     return length(max(pa, 0.)) +
5         min(max(max(pa.x, pa.y), pa.z), 0.);
6 }
```

Para la visualización de este resultado, vamos a aplicar una rotación sobre el eje *YZ* que lo veremos para  $\mathbb{R}^3$  en la siguiente sección.

```

1 float escena_sdf(vec3 p){
2     // Rotacion plano yz
3     vec3 pr = vec3(p.x, p.yz * rot(PI/4.0));
4     // Cubo s=0.3 => s '=0.15
5     vec3 sp = vec3(0.15);
6     return SDFPrisma(pr, sp);
7 }
```

El resultado,

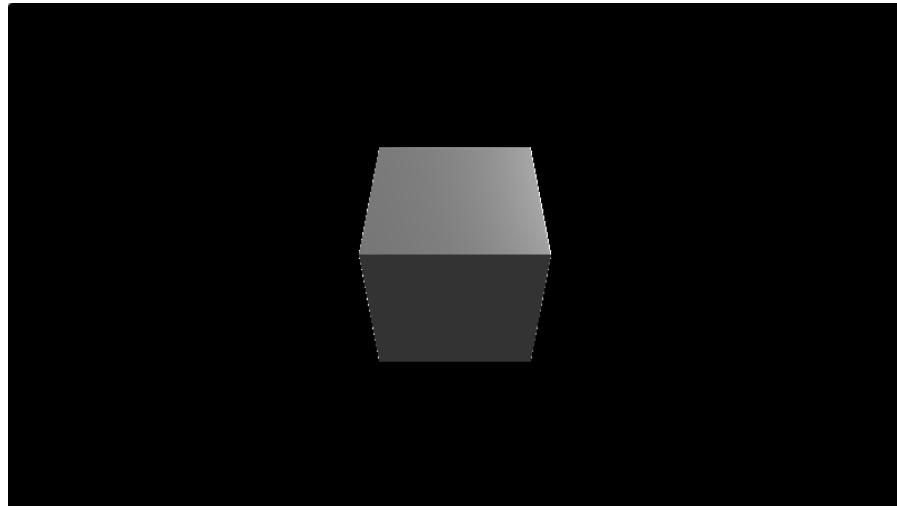


FIGURA 5.16: Prisma Rectangular  $\vec{s} = 0.3$  rotado  $\alpha_{YZ} = \frac{\pi}{4}$  FDS

Enlace del ejemplo:<https://www.shadertoy.com/view/Wt1BDj>

### 5.3.3 Plano con signo

Vamos a ver una figura muy importante, el plano con signo divide a  $\mathbb{R}^3$  en dos, uno tendrá valores positivos y el otro, negativos. El signo del producto escalar con la normal del plano indicará en que subespacio se encuentran los puntos. Para esto, definimos el operador del signo de un número:  $\text{sign}(a) = \pm 1, 0$ .

En secciones siguientes, veremos por qué es útil el signo de esta función, definiendo así otro operador. Su demostración se basa en la proyección de un punto  $\vec{p}$  en el plano<sup>6</sup>, centrado en el origen y con un vector normal  $\vec{n}$ .

$$\text{proy}_{\vec{n}}(\vec{p}) = \vec{p} - (\vec{n} \cdot \vec{p})\vec{n}$$

Por lo que la distancia con signo a este:

$$SDF_{Plano_{\vec{n}}}(\vec{p}) = \text{sign}((\vec{p} - \text{proy}_{\vec{n}}(\vec{p})) \cdot \vec{n}) \cdot \|\vec{p} - \text{proy}_{\vec{n}}(\vec{p})\|$$

Simplificamos esta ecuación,

$$SDF_{Plano_{\vec{n}}}(\vec{p}) = \text{sign}((\vec{n} \cdot \vec{p})\vec{n}) \cdot \|(\vec{n} \cdot \vec{p})\vec{n}\|$$

Utilizando las siguientes propiedades:  $(\vec{n}k) \cdot \vec{n} = k$  y  $\|k\vec{n}\| = k\|\vec{n}\|$ .

$$SDF_{Plano_{\vec{n}}}(\vec{p}) = \text{sign}(\vec{n} \cdot \vec{p}) \cdot (\vec{n} \cdot \vec{p})\|\vec{n}\| = \vec{n} \cdot \vec{p}$$

En código,

---

6 Podemos definir un plano a partir de su normal y un punto, en particular, el punto será  $0,0,0$  y la normal será un parámetro introducido por el usuario. Formalmente lo encontramos en el siguiente enlace <https://mathworld.wolfram.com/Plane.html>.

```

1 // Plano R3
2 float SDFPlano(vec3 p, vec3 n){
3     return dot(p, n);
4 }
```

Veamos un ejemplo, pero antes, vamos a desplazar el plano hacia abajo con el operador de translación, visto en la sección anterior para una dimensión inferior, pero que veremos que es equivalente para esta dimensión.

```

1 float escena_sdf(vec3 p){
2     // Desplazamiento
3     vec3 pt = p - vec3(0., -1., 0.);
4     // Plano con normal hacia arriba
5     vec3 n = normalize(vec3(0., 1., 0.));
6     return SDFPlano(pt, n);
7 }
```

El resultado,



FIGURA 5.17: Plano  $\vec{n} = (0, 1, 0)$  desplazado  $\vec{t} = (0, -1, 0)$

Enlace del ejemplo:<https://www.shadertoy.com/view/3llBW2>

#### 5.3.4 Recta y segmento exacto

Podemos generalizar de la definición vista para la recta en  $\mathbb{R}^2$  para una dimensión superior, así como para la proyección y el segmento. En código,

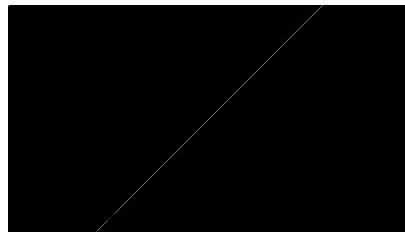
```

1 // Proyección a sobre b
2 vec3 proy(in vec3 a, in vec3 b){
3     return b * dot(b, a) / dot(b, b);
4 }
```

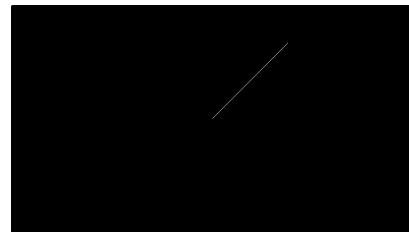
```

5 // Proyección a sobre b restringido 0, 1
6 vec3 proy01(in vec3 a, in vec3 b){
7     return b * clamp(dot(b, a) / dot(b, b),
8         0., 1.);
9 }
10 // Recta SDF
11 float SDFRecta(vec3 p, vec3 a, vec3 b){
12     vec3 v = p - a;
13     vec3 w = b - a;
14     return length(v - proy01(v, w));
15 }
16 // Segmento SDF
17 float SDFSegmento(vec3 p, vec3 a, vec3 b){
18     vec3 v = p - a;
19     vec3 w = b - a;
20     return length(v - proy01(v, w));

```



(A) FDS de una recta



(B) FDS de un segmento

FIGURA 5.18: Recta y segmento exactos con  $\vec{a} = \vec{0}$ ,  $\vec{b} = 0.5\vec{i}$ , FDS

Enlace ejemplo recta: <https://www.shadertoy.com/view/3tXBWX>  
 y ejemplo segmento: <https://www.shadertoy.com/view/WtXBWX>

Veremos un operador para incrementar el ancho de la recta o del segmento. Algunos de los operadores que vamos a ver también son generalizaciones de los operadores vistos en  $\mathbb{R}^2$ .

## 5.4 OPERADORES SOBRE $\mathbb{R}^3$

### 5.4.1 Operadores Isométricos

Tenemos a los mismos *Operadores isométricos sobre  $\mathbb{R}^2$*  vistos en secciones anteriores: el operador de traslación, el operador de rotación, el de simetría y el operador de escalado. Se puede demostrar que los operadores son isometrías de manera equivalente a como se demostraron para  $\mathbb{R}^2$ .

Vamos a ver la rotación ya que esta se ha visto sobre el plano  $\overline{XY} = \mathbb{R}^2$  en  $\mathbb{R}^3$ . Definimos una rotación sobre cada uno de los tres planos formados por los ejes, tales que:

Para el plano  $\overline{XY}$ , mantenemos la coordenada  $Z$  como libre:

$$\text{rotacion}_{\overline{XY}}^\alpha(\vec{p}) = (x \cos(\alpha) + y \sin(\alpha), -x \sin(\alpha) + y \cos(\alpha), z)$$

Para el plano  $\overline{YZ}$ , con la coordenada  $X$  como libre:

$$\text{rotacion}_{\overline{YZ}}^\alpha(\vec{p}) = (x, y \cos(\alpha) + z \sin(\alpha), -y \sin(\alpha) + z \cos(\alpha))$$

Finalmente, el plano  $\overline{XZ}$ , con la coordenada  $Y$  libre:

$$\text{rotacion}_{\overline{XZ}}^\alpha(\vec{p}) = (x \cos(\alpha) + z \sin(\alpha), y, -x \sin(\alpha) + z \cos(\alpha))$$

Aunque su definición parece bastante compleja, en código es bastante simple de implementar,

```

1 // Matriz de Rotacion
2 mat2 rot(float a){
3     return mat2(
4         +cos(a), -sin(a),
5         +sin(a), +cos(a)
6     );
7 }
8 // Rotación del plano XY
9 vec3 rotXY(vec3 p, float a){
10    vec2 pr = p.xy * rot(a);
11    return vec3(pr.x, pr.y, p.z);
12 }
13 // Rotación del plano YZ
14 vec3 rotYZ(vec3 p, float a){
15    vec2 pr = p.yz * rot(a);
16    return vec3(p.x, pr.x, pr.y);
17 }
18 // Rotación del plano XZ
19 vec3 rotXZ(vec3 p, float a){
20    vec2 pr = p.xz * rot(a);
21    return vec3(pr.x, p.y, pr.y);
22 }
```

El resultado de rotar un cubo sobre el plano  $\overline{YZ}$  lo podemos ver en la [Figura ?? Prisma rectangular exacto](#), con un ángulo  $\alpha = \frac{\pi}{4}$ .

Vamos a presentar un operador capaz de incrementar la *isosuperficie* de una figura, similar a como hicimos con la esfera, pudiendo también redondear superficies, que es consecuencia directa de la métrica euclídea.

### 5.4.2 Operador de ensanchamiento

Presentamos el operador de ensanchamiento, este operador va a crear otras funciones de distancia con signo con mayor superficie, ¿ó menor?. Por ejemplo, puede ser útil en situaciones cuando el marcher no es capaz de trazar la figura debido a la poca superficie, como nos pasa con la recta o el segmento.

Restaremos un valor  $k \in \mathbb{R}$  a la *isosuperficie*, algunos autores llaman a este operador «salto de *isosuperficie*»<sup>7</sup>, aunque también se puede utilizar para *isoperímetros*.

Una idea de la demostración es observar que sobre cualquier punto de una isosuperficie se puede posicionar una esfera cuya superficie es proporcional al radio, la resta de  $k$  sobre las distancias provoca un incremento del radio de la esfera y por tanto, de la superficie.

$$\text{ensanchar}_k(\vec{p}, f) = f(\vec{p}) - k$$

Veamos el ensanchamiento del segmento visto en [5.18b Subfigure 5](#) [5.18b](#), esta nueva función es llamada cápsula.

```

1 // FDS Cápsula
2 float SDFCapsula(vec3 p, vec3 a, vec3 b,
3                     float k){
4     return SDFSegmento(p, a, b) - k;
5 }
```

El resultado es el siguiente,

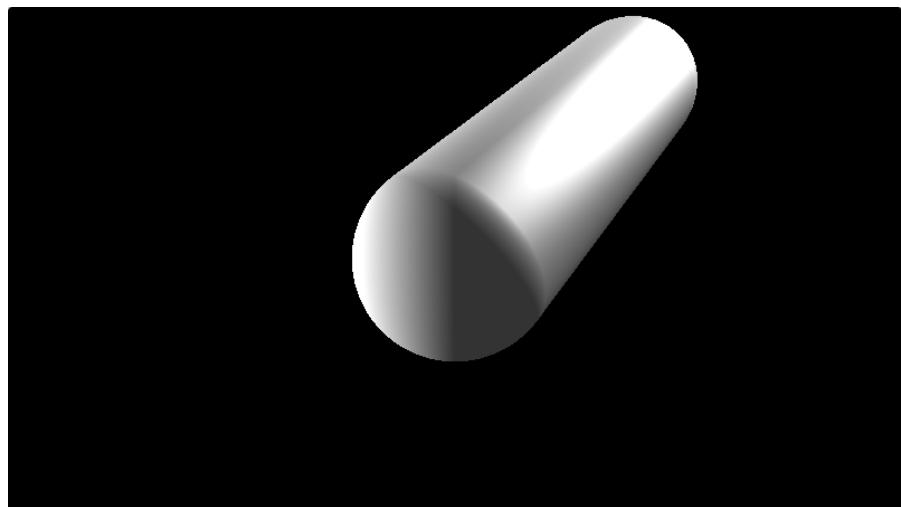


FIGURA 5.19: Segmento ensanchado exacto FDS

---

<sup>7</sup> Íñigo Quilez utiliza esta terminología en su blog «Iquilezles» en una de sus entradas: «3D Distance Functions, Rounding - exact». El enlace, <https://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm>

Enlace del ejemplo:<https://www.shadertoy.com/view/3tXfD1>

Podemos observar que este operador devuelve una figura ensanchada con los bordes redondeados, debido a la métrica, aunque podemos conseguir una figura con bordes redondeados y con las dimensiones originales, sin ensanchar, combinándolo con el operador de escalado.

Finalmente, vamos a presentar operadores que transforman figuras exactas de  $\mathbb{R}^2$  en otras en  $\mathbb{R}^3$  de manera exacta. Esto nos será útil ya que podemos calcular una *función de distancia con signo exacta* en  $\mathbb{R}^2$  y utilizar uno de estos operadores para crear uno nuevo que deducirlo matemáticamente, hubiera sido tedioso. Vamos a ver dos, el operador de *revolución* y el operador de *extrusión*.

#### 5.4.3 Operador de revolución

Vamos a demostrar para el caso particular de la revolución sobre el plano  $\overline{XZ}$ , ya que las demostraciones para los demás ejes son equivalentes. Veamos una intuición general de la demostración, para cualquier punto  $\vec{p}$ , rotamos el eje  $\overline{XZ}$  hasta hacer que el plano  $\overline{XY}$  lo contenga, o lo que es lo mismo, anulando la componente  $z$  del vector proyectado  $\vec{p}_{\overline{XZ}} = \begin{pmatrix} x & z \end{pmatrix}$ , a partir de ahí, basta con aplicar la *función de distancia con signo 2D* sobre el punto calculado:

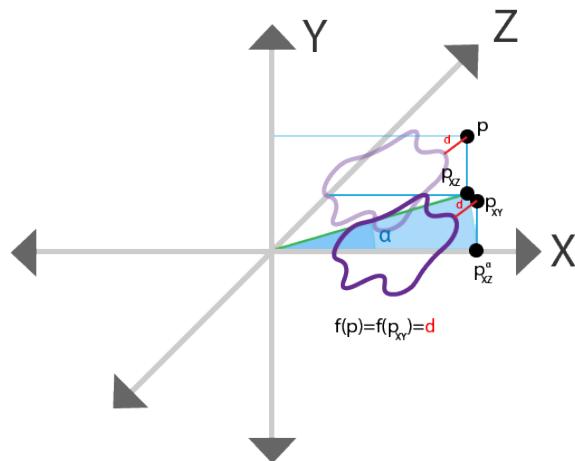


FIGURA 5.20: Operador de revolución, ejemplo sobre  $\overline{XZ}$

Utilizaremos el ángulo  $\alpha$  sobre la proyección  $\vec{p}_{\overline{XZ}}$  y utilizaremos la matriz de rotación en sentido horario para anular  $z$ :

$$\vec{p}_{\overline{XZ}}^\alpha = \vec{p}_{\overline{XZ}} \cdot \text{rot}(\alpha)$$

Conociendo las equivalencias trigonométricas de sin y cos, transformamos nuestra matriz de rotación, tal que:

$$\text{rot}(\alpha) = \begin{pmatrix} \frac{x}{\|\vec{p}_{XZ}\|} & \frac{-z}{\|\vec{p}_{XZ}\|} \\ \frac{z}{\|\vec{p}_{XZ}\|} & \frac{x}{\|\vec{p}_{XZ}\|} \end{pmatrix} = \begin{pmatrix} x & -z \\ z & x \end{pmatrix} \cdot \frac{1}{\|\vec{p}_{XZ}\|}$$

haciendo,

$$\vec{p}_{XZ}^\alpha = \begin{pmatrix} x \\ z \end{pmatrix}^t \cdot \begin{pmatrix} x & -z \\ z & x \end{pmatrix} \cdot \frac{1}{\|\vec{p}_{XZ}\|} = \begin{pmatrix} \frac{x^2 + z^2}{\|\vec{p}_{XZ}\|} \\ 0 \end{pmatrix}^t = \begin{pmatrix} \|\vec{p}_{XZ}\| \\ 0 \end{pmatrix}^t$$

en esta rotación, la coordenada  $y$  es invariante, por lo que el vector sobre el plano  $\overline{XY}$  será:

$$\vec{p}_{XY} = \begin{pmatrix} \|\vec{p}_{XZ}\| \\ p_y \end{pmatrix}^t$$

Sobre este plano, vamos a utilizar nuestro *FDS* 2D,  $f$ , definiendo el operador de revolución como:

$$\text{revolucion}_{\overline{XY}}(\vec{p}, f) = f((\|\vec{p}_{XZ}\|, \vec{p}_y))$$

Podemos utilizar una isometría de traslación para añadir un radio de revolución, tal que:

$$\text{revolucion}_{\overline{XY}}(\vec{p}, f) = f((\|\vec{p}_{XZ}\|, \vec{p}_y) - \vec{t})$$

Con esta definición, podemos generar de forma exacta, un *toro*. Utilizaremos la función de una circunferencia trasladada horizontalmente  $r_x$  unidades, al que llamaremos, radio interior del toro y utilizaremos otra variable  $r$  que será el radio del toro. En código:

```

1 // FDS Toro
2 float SDFToro(vec3 p, float rx, float r){
3     vec2 rev = vec2(
4         length(p.xz),
5         p.y
6     );
7     // Trasladamos, radio de revolucion
8     // o radio interior del toro.
9     vec2 pt = rev - vec2(rx, 0.);
10    // Radio Toro
11    return SDFCircunferencia(pt, r);
12 }
```

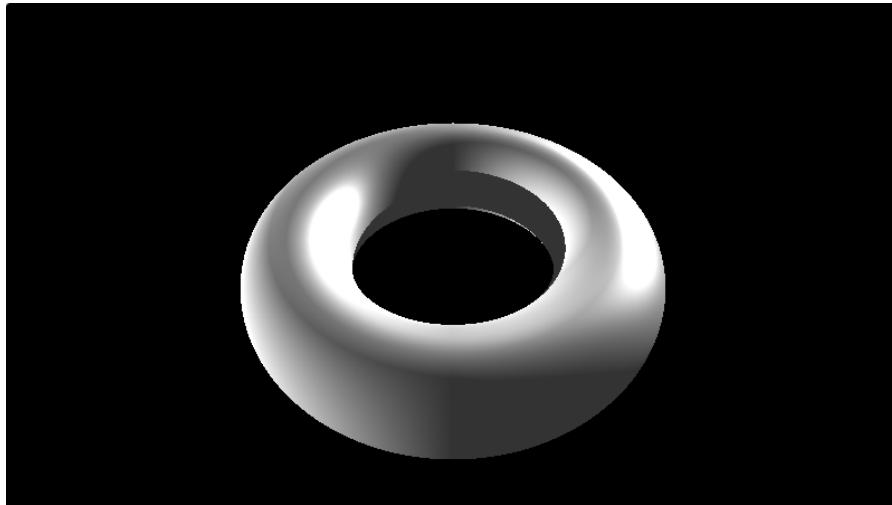


FIGURA 5.21: Toro exacto por revolución FDS

Enlace del ejemplo:<https://www.shadertoy.com/view/WtsBDf>

#### 5.4.4 Operador de extrusión

Este último operador consiste en elongar una figura plana hacia un eje. Pudiendo ser de manera finita o infinita. Cada punto  $\vec{p} \in \mathbb{R}^3$  se proyecta sobre el plano a extruir y se calcula la función de distancia  $\mathbb{R}^2$ . Sea  $\vec{n}$  el vector normal del plano de extrusión y  $f$  un *FDS*, el operador se define como:

$$\text{extrusion}_{\vec{n}}^\infty(\vec{p}, f) = f(\text{proj}_{\vec{n}}(\vec{p}))$$

En caso de una extrusión finita, de longitud  $h$ , definimos como  $\Delta c$  la distancia a la tapadera, haremos uso de la simetría sobre el plano, reduciendo el problema, tal que, la «altura» será:

$$\Delta c = ||\vec{p} - \text{proj}_{\vec{n}}(\vec{p})|| - h$$

Dividimos el ejercicio en dos subproblemas, por encima de la tapa  $\Delta c \geq 0$  y aquellas por debajo  $\Delta c < 0$ . Como referencia, ?? ??.

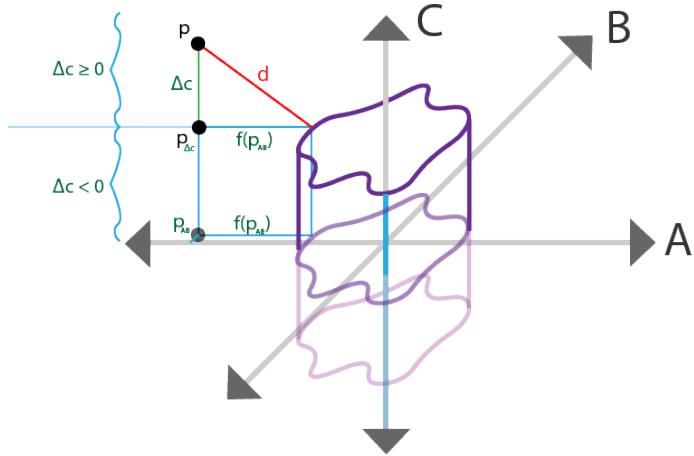


FIGURA 5.22: Visualización de una extrusión genérica

- Subproblema 1.** La distancia puede ser negativa o positiva y corresponde con la distancia que devuelve nuestra función  $f$  sobre la proyección del punto en el plano.

$$d_1 = f(\text{proy}_{\vec{n}}(\vec{p}))$$

- Subproblema 2.** Se trata de distancias positivas, observamos en la imagen que se forma un triángulo rectángulo de altura  $\Delta c$  y de base  $f(\text{proy}_{\vec{n}}(\vec{p}))$  como el **Subproblema 1**. La diagonal  $d$  corresponde a la distancia de  $\vec{p}$  a la superficie de la tapadera.

$$d_2 = \sqrt{(\Delta c)^2 + f(\text{proy}_{\vec{n}}(\vec{p}))^2}$$

Veamos como podemos unificar ambos subproblemas como hicimos con el rectángulo. Cuando  $\Delta c < 0$ , queremos que  $d_1 = d_2$ , esto se consigue haciendo que  $\Delta c$  sea 0 cuando este sea negativo,

$$d_2 = \sqrt{\max(\Delta c, 0)^2 + d_1^2} = \|(\max(\Delta c, 0), d_1)\|$$

Vamos a generar el interior y exterior de la figura, los cuales se anularán con independencia. Por eso,  $d_2$  se anulará cuando el punto esté en el interior. Para ello, en la ecuación,  $d_1$  debe anularse cuando este sea negativo.

$$d_{exterior} = \|(\max(\Delta c, 0), \max(d_1, 0))\|$$

El interior de la figura ocurre cuando  $\Delta c < 0$  y  $d_1 < 0$ , por lo que se anularán cuando sean positivos. Tomaremos el máximo de las dos distancias para encontrar la mínima negativa.

$$d_{interior} = \text{argmax}(\Delta c, d_1) = \min(\max(\Delta c, f(\text{proy}_{\vec{n}}(\vec{p}))), 0)$$

Finalmente, como se anulan, podemos sumar y la ecuación resultante será:

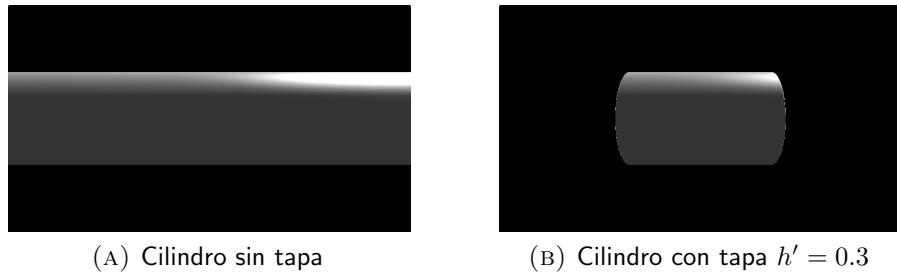
$$\text{extrusion}_{\vec{n}}^h(\vec{p}, f) = \|\max((\Delta c, d_1), 0)\| + \arg\max(\Delta c, d_1)$$

Unos ejemplos de estas técnicas es la fabricación del cilindro exacto con y sin tapa de radio  $r$ . En código:

```

1 // Proyección sobre plano
2 vec3 proyPlano(vec3 p, vec3 n){
3     return p - dot(p, n) * n;
4 }
5 // FDS circunsferencia
6 float SDFCircunferencia(vec2 p, float r){
7     return length(p) - r;
8 }
9 // FDS Cilindro
10 float SDFCilindro(vec3 p, float r){
11     // Proyección sobre el plano YZ => n = X
12     vec3 n = normalize(vec3(1, 0, 0));
13     // Cogemos el plano YZ
14     vec2 proy = proyPlano(p, n).yz;
15     // Devolvemos la circunsferencia
16     return SDFCircunferencia(proy, r);
17 }
18 // FDS Cilindro Finito (Con Tapa)
19 float SDFCilindro(vec3 p, float h, float r){
20     // Proyección sobre el plano YZ => n = X
21     vec3 n = normalize(vec3(1, 0, 0));
22     // Calculamos la proyección
23     vec3 proy = proyPlano(p, n);
24     // Altura sobre la tapa.
25     float dc = length(p - proy) - h;
26     // FDS Circunferencia de radio r sobre la
27     // proyección en el plano YZ.
28     float fproy = SDFCircunferencia(proy.yz,
29         r);
30     // Exterior figura
31     float dint = length(max(vec2(dc, fproy)
32         , 0.));
33     // Interior de la figura
34     float dext = min(max(dc, fproy), 0.);
35     // Sumamos ambos
36     return dint + dext;
37 }
```

Creamos una circunsferencia en el plano  $\overline{YZ}$  y la extruimos utilizando el vector normal  $\vec{n} = (0, 0, 1)$ .

FIGURA 5.23: Cilindro exacto por extrusión con  $r = 0.2$ , FDS

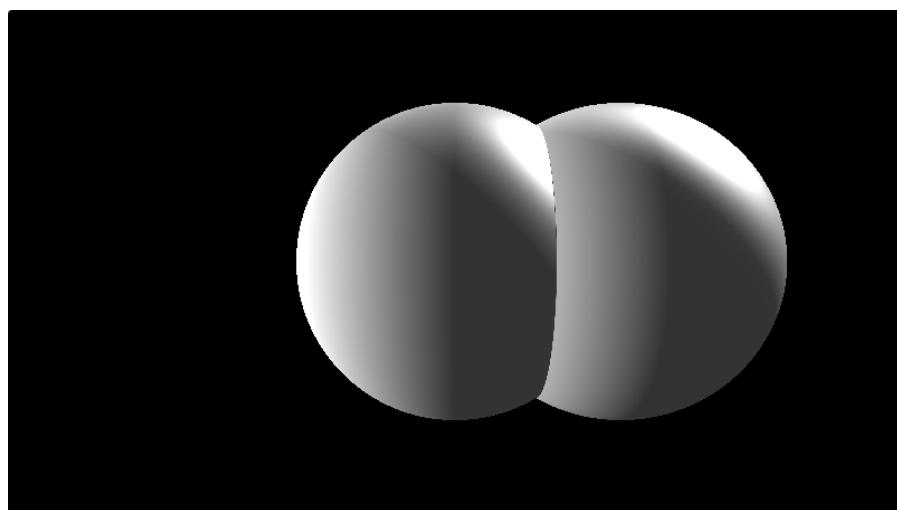
Enlace ejemplo cilindro: <https://www.shadertoy.com/view/3lsfDX> , cilindro acotado, <https://www.shadertoy.com/view/3t1BWf>

#### 5.4.5 Operador de agregación y substracción

Los operadores de agregación y de substracción son también generalizados para  $\mathbb{R}^3$ . Vamos a presentar también un nuevo operador, el operador de sección en el que utilizaremos el plano con signo, visto anteriormente. Para el primer ejemplo, vamos a utilizar dos esferas con el mismo radio y una separada de la otra. En código:

```

1 float escena_sdf(vec3 p){
2     // Dos esferas, una trasladadas
3     // Agregacion
4     return min(
5         SDFEsfera(p, 0.3),
6         SDFEsfera(p - vec3(0.3, 0., 0.), 0.3)
7     );
8 }
```

FIGURA 5.24: Agregación de dos esferas  $r = 0.3$  y una trasladada

Enlace del ejemplo:<https://www.shadertoy.com/view/3llBW1>  
 Como ya se ha comentado, el operador «máx» devuelve la intersección de ambas figuras. Vamos a utilizar la definición del plano con signo para seccionar una figura, ya que una región es sólida y la otra no.

```

1 float escena_sdf(vec3 p){
2     // Rotamos el plano XZ, pi / 4 rad
3     p = rotXZ(p, PI / 2. * 1.2);
4     // Sección de una esfera
5     return max(
6         SDFEsfera(p, 0.3),
7         SDFPlano(p - vec3(0., 0., 0.15), vec3
8             (0., 0., 1.))
9     );

```

Se trata de una sección del eje  $\vec{z}$ , desplazando el plano con el operador de translación con  $\vec{t} = (0, 0, 0.3)$ , pudiendo así modificar la posición de la sección. El resultado es el siguiente:

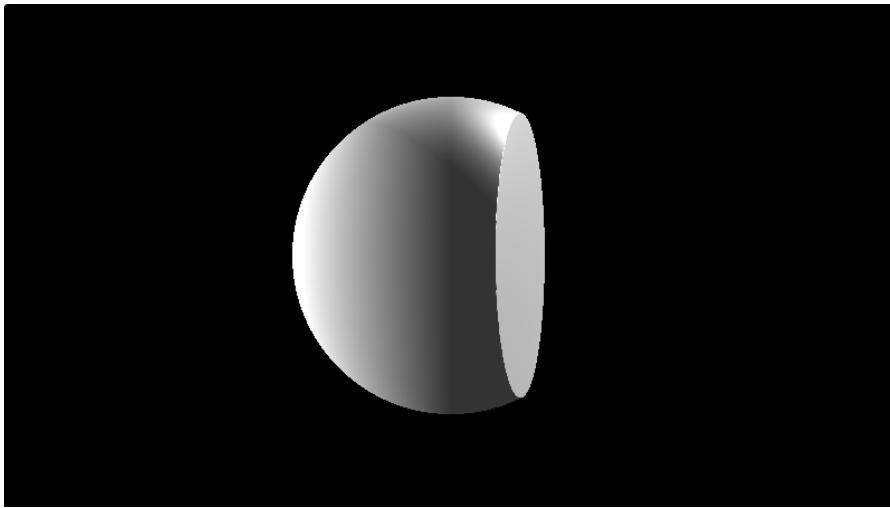


FIGURA 5.25: Una esfera  $r = 0.3$  seccionada por un plano  $\vec{n} = (0, 0, -1)$  desplazado

Enlace del ejemplo:<https://www.shadertoy.com/view/WtsBW1>

Finalmente, veamos un ejemplo de *subtracción*, utilizando las dos esferas vistas antes. En código:

```

1 float escena_sdf(vec3 p){
2     // Rotamos el plano XZ, pi / 4 rad
3     p = rotXZ(p, PI / 4.);
4     // Substracción b en a => max(a, -b)

```

```

5      return max(
6          SDFEsfera(p, 0.3),
7          -SDFEsfera(p - vec3(0.3, 0., 0.),
8                      0.3)
9      );

```

Rotaremos la escena para poder ver así el interior, utilizaremos el operador de rotación del plano  $\overline{XZ}$  con  $\alpha = \frac{\pi}{4}$ . El resultado será una esfera en el centro a la que se le ha carvado otra esfera, trasladada:

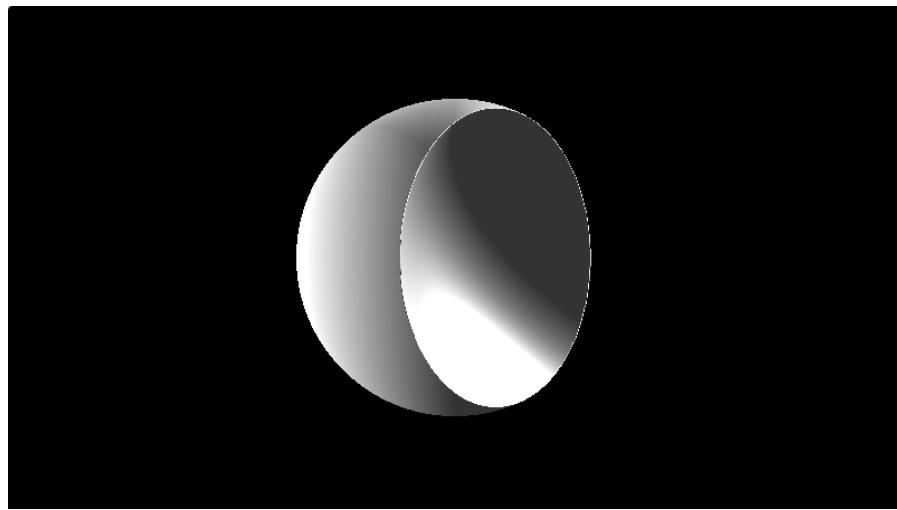


FIGURA 5.26: Dos esferas  $r = 0.3$  y substracción de la trasladada

Enlace del ejemplo: <https://www.shadertoy.com/view/WlsBW1>

#### 5.4.6 Operador de deformación no exacta

Veamos el último operador el de deformación, recordemos que este no siempre conserva la métrica, provocando lo que llamaremos «artefactos». En la siguiente sección veremos como resolverlos. Vamos a definir la siguiente deformación continua y derivable:

$$g(\vec{p}) = (\vec{p}_x \cos(10\vec{p}_y) + \vec{p}_z \sin(10\vec{p}_y), \vec{p}_y, \vec{p}_x \sin(10\vec{p}_y) + \vec{p}_z \cos(10\vec{p}_y))$$

Esta deformación es conocida como *torsión*, consiste en rotar un eje, o plano, a medida que incrementa la distancia a este. Utilizaremos un cubo para este ejemplo, en código:

```

1 float escena_sdf(vec3 p){
2     // Rotamos la escena
3     vec2 ry = p.yz * rot(PI / 4.0);
4     p = vec3(p.x, ry.x, ry.y);

```

```

5   // Deformación "g"
6   float k = 10.0; // periodo.
7   float a = p.y * k;
8   p = vec3(
9       +p.x * cos(a) + p.z * sin(a),
10      +p.y,
11      -p.x * sin(a) + p.z * cos(a)
12  );
13  // Dibujamos un prisma.
14  return SDFPrisma(p, vec3(0.2));
15 }

```

El resultado obtenido es el siguiente:

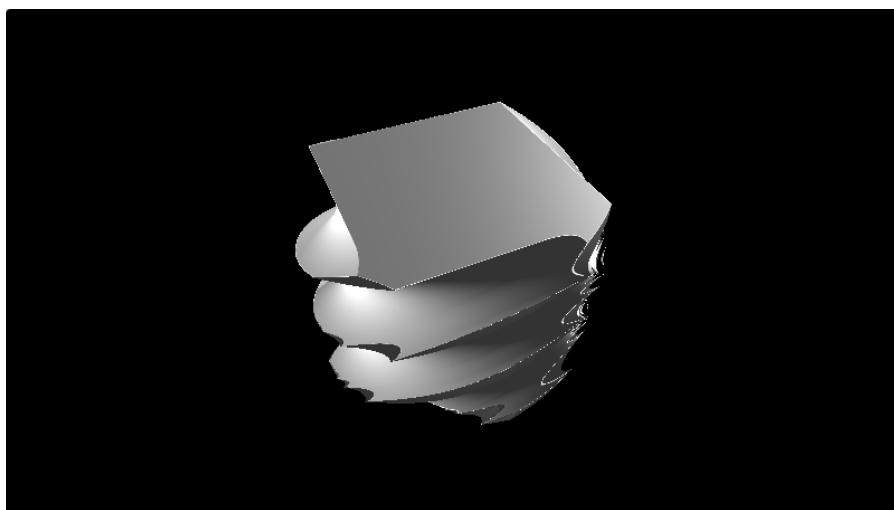


FIGURA 5.27: Deformación torsión de un cubo  $\vec{l} = 0.2$

Enlace del ejemplo: <https://www.shadertoy.com/view/ttlfWl>

Como vemos, esta deformación ha creado artefactos, por ejemplo, la tapadera es no es completamente cuadrada y vemos que se ha deformado. Esto ocurre ya que la bola también ha sido deformada y esta ahora puede contener algún punto en su interior, sobreestimando.



## 6. RESOLUCIÓN DE ARTEFACTOS

---

En el capítulo anterior hemos visto dos tipos de *funciones de distancia con signo* exactas e inexactas. Las funciones exactas devuelven la escena de manera correcta, ignorando imperfecciones por operaciones de coma flotante. Esto es debido a que la métrica es la euclídea y el *Marcher* siempre traza una esfera con ningún punto en el interior, de ahí su nombre, *Spheremarching*. Cuando tratamos de funciones inexactas, las esferas trazadas también se deforman, por lo que las distancias también lo hacen y por tanto, pueden contener puntos.

Encontramos dos problemas cuando tratamos de *funciones de distancia con signo inexactas*, en el *Marcher* puede sobreestimar la distancia o subestimar.

### 6.1 SOBREESTIMACIÓN DE LA DISTANCIA

Cuando tratamos *funciones de distancia con signo no exactas*, utilizaremos el término «sobreestimar» cuando se supera la distancia mínima real a la superficie, que es equivalente a que la bola contenga al menos un punto. Encontramos dos formas de estimar: El rayo trazado se encuentra dentro de la superficie o que el rayo atraviese una superficie.

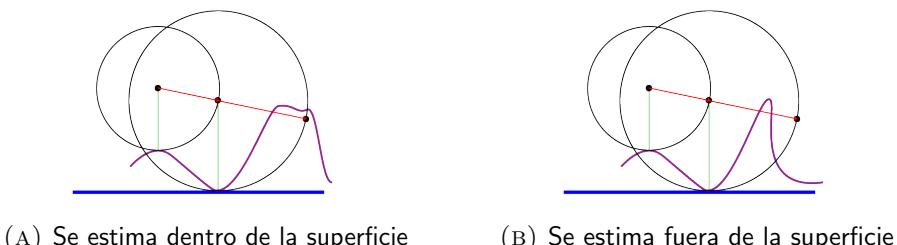


FIGURA 6.1: Dos formas de sobreestimar una superficie

Es difícil visualizar las circunferencias deformadas, por lo que se ha realizado un esquema de las dos situaciones que podemos encontrar.

### 6.1.1 Sobreestimación dentro de la superficie

Tenemos definido estar sobre una superficie cuando  $d_{n+1} < \epsilon$ , esta restricción había sido elegida ya que  $d_{n+1}$  siempre es positiva cuando trazamos desde fuera de la superficie. Cuando se sobreestima en el interior de la superficie, la siguiente distancia  $d_{n+1}$  será negativa. Debido a la condición de parada anterior, estos puntos del interior son considerados superficie y el algoritmo terminaría. Para solucionarlo, diremos que estamos sobre una superficie, si y solo si,  $|d_{n+1}| < \epsilon$ , este cambio forzará al *Marcher* a que, en caso de estar en el interior de la superficie, el rayo deba salir de la superficie. Veamos el cambio en el código del algoritmo:

```

1 float SphereMarching(
2     vec3 ojo,
3     vec3 direccion
4 ){
5     float distancia = 0.0;
6     for(int i = 0; i < PASOS; ++i){
7         vec3 p = ojo + direccion * distancia;
8         // d_n+1
9         float radio = escena_sdf(p);
10        // Estamos sobre la superficie cuando
11        // aunque sobreestimemos, no estemos
12        // <sobre> la superficie.
13        if(abs(radio) < EPSILON){
14            return distancia;
15        }
16        // Cuando d_n+1 sea negativo,
17        // intentará escapar del interior
18        // hacia el exterior.
19        distancia += radio;
20        if(distancia >= MAXIMO) break;
21    }
22    return MAXIMO;
23 }
```

Veamos el efecto que implica este cambio sobre la deformación vista en Figura ?? Operador de deformación no exacta.

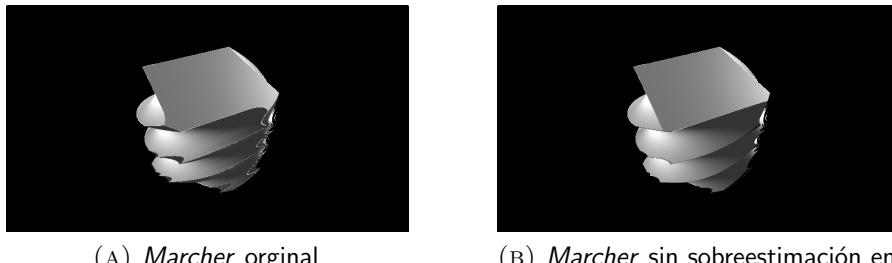


FIGURA 6.2: Comparativa de los cambios realizados en el *Marcher*. A la izquierda, el *Marcher* original, a la derecha, el *Marcher* con  $|d_{n+1}| < \epsilon$

Enlace del ejemplo:<https://www.shadertoy.com/view/ttsBDs>

## 6.2 SOBREESTIMACIÓN FUERA DE LA SUPERFICIE

Este segundo caso ocurre cuando la deformación aplicada hace que el rayo atraviese la superficie, como ocurre en el ejemplo (B). Esta sobreestimación afecta considerablemente a la eficiencia del *Marcher*. La solución es escalar en todo momento la bola trazada, obligando a que la bola deformada no pueda contener ningún punto, es decir, sobreestimar. Este cambio nos obligará a utilizar un mayor número de iteraciones, de forma proporcional, para alcanzar la superficie.

$$d'_n = d_{n-1} + f(\vec{p}_{n-1}) \cdot k \leq d_n$$

Donde  $f$  es nuestra escena;  $k \in [0, 1]$  es un factor de escalado de la bola y proporcional al nuevo número de iteraciones. Además, puede ayudar, en ciertas situaciones, a resolver el problema de la *sobreestimación dentro de la superficie*.

```
1 #define FACTOR_SOBREESTIMACION k
2 float SphereMarching(
3     in vec3 ojo,
4     in vec3 direccion,
5     float distancia_plano
6 ){
7     float distancia = 0.0;
8     for(int i = 0; i < PASOS; ++i){
9         vec3 rayo = ojo + direccion *
10            distancia;
11         // Escalamos el radio de la esfera
12         float radio = escena_sdf(rayo) *
13             FACTOR_SOBREESTIMACION;
14         if(abs(radio) < EPSILON){
15             return distancia;
16         }
17     }
18 }
```

```

14      }
15      distancia += radio;
16      if(distancia > distancia_plano)break;
17  }
18  return distancia_plano;
19 }
```

Efecto de distintos factores  $k$  al trazado de la escena por el *Marcher*.

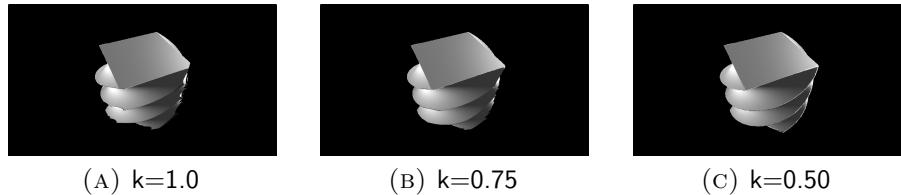


FIGURA 6.3:  $k \in \{1, 0.75, 0.5\}$  respectivamente sobre el *Marcher*

Enlace del ejemplo: <https://www.shadertoy.com/view/3tBBRR>

### 6.2.1 Subestimación de la distancia

Esta estimación puede ocurrir tanto en *funciones de distancia con signo exactas e inexactas*. Ocurre cuando el *Marcher* ha finalizado pero la distancia recorrida por el rayo es inferior a la del plano trasero, es decir, el rayo sigue estando presente en la escena. Por ejemplo, puede ocurrir cuando el rayo pasa de manera paralela, muy cerca a una superficie. Pero, en realidad, no está «sobre» ella, es decir,  $f(\vec{rayo}) \geq \epsilon$ . La siguiente imagen ilustra este problema:

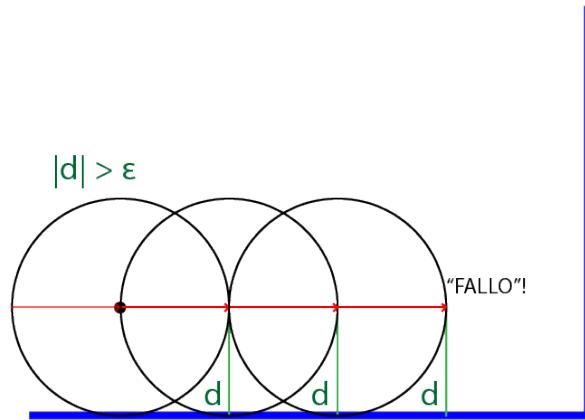


FIGURA 6.4: Ejemplo de subestimación de una superficie

Estos puntos serán tratados como «*fallos*» y por tanto, como píxeles de fondo. Utilizar un factor de sobreestimación  $k > 1$  no

sería la solución, ya que, crearía *artefactos*. La principal solución es incrementar el número de iteraciones del algoritmo, es decir, incrementar el número de «*PASOS*», provocando un mayor gasto computacional.



## 7. MATERIALES

---

En este último capítulo, vamos a ver como dar color y texturas a los elementos de nuestra escena. Identificaremos cada elemento asignando un entero positivo  $id \in \mathbb{N}$  que será devuelto junto con la distancia a este objeto, es decir, vamos a devolver un  $vec2$  cuya componente «x» será la distancia y cuya componente «y», el identificador  $id$ . Asignaremos la constante  $id = -1$  cuando no se ha trazado ningún objeto. Primero, vamos a modificar el *Marcher* para que este pueda devolver ambos valores:

```
1 // Devolvemos dos elementos, distancia e id.
2 vec2 SphereMarching(vec3 ojo, vec3 direccion)
3 {
4     float distancia = 0.0;
5     // Realizamos PASOS iteraciones de
6     // marching.
7     for(int i = 0; i < PASOS; ++i){
8         vec3 p = ojo + direccion * distancia;
9         // La escena devuelve el radio de la
10        // bola y el id del elemento
11        vec2 info = escena_sdf(p);
12        // Factor para la sobreestimación.
13        info.x *= 1.0;
14        // info.x contiene la distancia
15        if(abs(info.x) < EPSILON){
16            // info.y contiene el id de un
17            // elemento de la escena.
18            // Devolvemos la distancia
19            // acumulada (o distancia del ojo
20            // a la superficie) y el id.
21            return vec2(distancia, info.y);
22        }
23        // incrementamos la distancia
24        distancia += info.x;
25        if(distancia >= MAXIMO) break;
26    }
27    // Devolvemos un id desconocido.
28    return vec2(MAXIMO, -1);
29 }
```

El vector devuelto con nombre *info*, toma los valores directamente de la escena, por lo que vamos a modificar el esquema de nuestra función «*escena\_sdf*», este ahora devolverá la distancia más cercana a un objeto y su identificador. El esquema será el siguiente:

```

1 vec2 escena_sdf(vec3 p){
2     // Identificador inicial y la distancia máxima.
3     float id = -1.0;
4     float min_dist = MAXIMO;
5
6     // El esquema es el siguiente para cada figura de nuestra escena.
7     // 1. Creamos nuestra primera figura.
8     float sdf_0 = ....;
9     // 2. Comprobamos que esta figura es la más cercana encontrada hasta el momento.
10    if(sdf_0 < min_dist){
11        // 2.1 En caso afirmativo, actualizamos los valores.
12        // Asignamos el id de esta figura.
13        id = 0. ;
14        // Actualizamos la distancia mínima como la distancia a esa figura.
15        min_dist = sdf_0;
16    }
17
18    // Repetimos este esquema para cada elemento de la escena,
19    float sdf_1 = ...;
20    if(sdf_1 < min_dist){
21        id = 1. ;
22        min_dist = sdf_1;
23    }
24    ...
25    // Finalmente, devolvemos la distancia mínima y el objeto que la devuelve.
26    return vec2(min_dist, id);
27 }
```

Al devolver ahora dos componentes, debemos modificar todas las funciones que hacían uso de esta función, donde encontramos «Normal» y «ModeloIluminacion». Utilizamos el identificador devuelto para asignar un material, en código, recibe el nombre de «obtenerMaterial». El material será multiplicado por la intensidad.

$$\text{color}_{rgb}(id) = \text{obtenerMaterial}(id) \cdot I_{Phong}$$

Pongamos como ejemplo: una sección de un toro y una esfera, aplicando el esquema de código utilizado antes:

```

1 vec2 escena_sdf(vec3 p){
2     // ID inicial y distancia máxima.
3     float id = -1.0;
4     float min_dist = MAXIMO;
5     // Toro rotado ri = 0.3 y r = 0.05.
6         // seccionado por un plano n=-z.
7     float sdf_0 = max(
8         SDFToro(rotYZ(p, PI / 4.), .3, .05),
9         SDFPlano(p, vec3(0., 0., -1.)))
10    );
11    // Comprobamos que sea la mas cercana.
12    if(sdf_0 < min_dist){
13        // Identificador del toro
14        id = 0.;
15        min_dist = sdf_0;
16    }
17    // Esfera de radio 0.2
18    float sdf_1 = SDFEsfera(p, 0.2);
19    // Comprobamos que sea la mas cercana.
20    if(sdf_1 < min_dist){
21        // Identificador de la esfera.
22        id = 1.;
23        min_dist = sdf_1;
24    }
25    // (distancia mínima, id)
26    return vec2(min_dist, id);
}

```

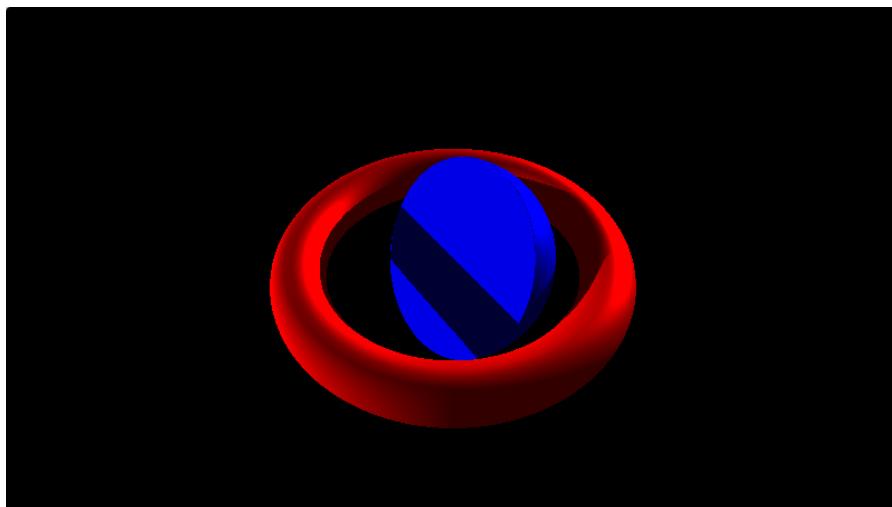


FIGURA 7.1: Materiales asignados a las distintas figuras.

Enlace del ejemplo:<https://www.shadertoy.com/view/w1BBRR>

En el ejemplo anterior, *obtenerMaterial*, hace uso únicamente del identificador para devolver el color deseado:

```

1 // Devuelve el color del material en el punto
   p
2 vec4 obtenerMaterial(vec3 p, float id){
3     // Identificador del toro seccionado
4     if(id == 0.){
5         return vec4(1., 0., 0., 1.);
6     }else if(id == 1.0){
7         return vec4(0., 0., 1., 1.);
8     }
9 }
```

Podemos utilizar el punto  $\vec{p}$  para crear una proyección hacia el sistema de coordenadas de alguna textura asignada en un canal y que podemos acceder mediante el operador «texture» de *GLSL*. Utilizaremos la textura creada en el ejemplo, 1.4 y calcularemos la proyección de la esfera sobre las coordenadas de la textura:

$$(u \ v) = 0.5 \cdot \left( \frac{\arctan\left(\frac{\vec{p}_x}{\vec{p}_z}\right)}{\pi} + 1 \quad \vec{p}_y + 1 \right)$$

```

1 // Devuelve el material en el punto p
2 vec4 obtenerMaterial(vec3 p, float id){
3     // Identificador del toro seccionado
4     if(id == 0.){
5         return vec4(1., 0., 0., 1.);
6     }else if(id == 1.0){
7         vec3 n = normalize(p);
8         vec2 uv = vec2(
9             atan(n.x, n.z) / (2.*PI) + 0.5,
10            n.y * 0.5 + 0.5
11        );
12        // Ejemplo : TFG 0-1 Homotopía
13        vec4 text0 = texture(iChannel0,uv);
14        vec4 text1 = texture(iChannel1,uv);
15        float mascara = min(texture(iChannel2
16            , uv*.25).r*2., 1.);
17        vec3 homotopia = mix(text0.rgb, text1
18            .rgb, h(mascara));
19        // Devolvemos la textura
20        return vec4(homotopia, 1.);
```

El resultado:

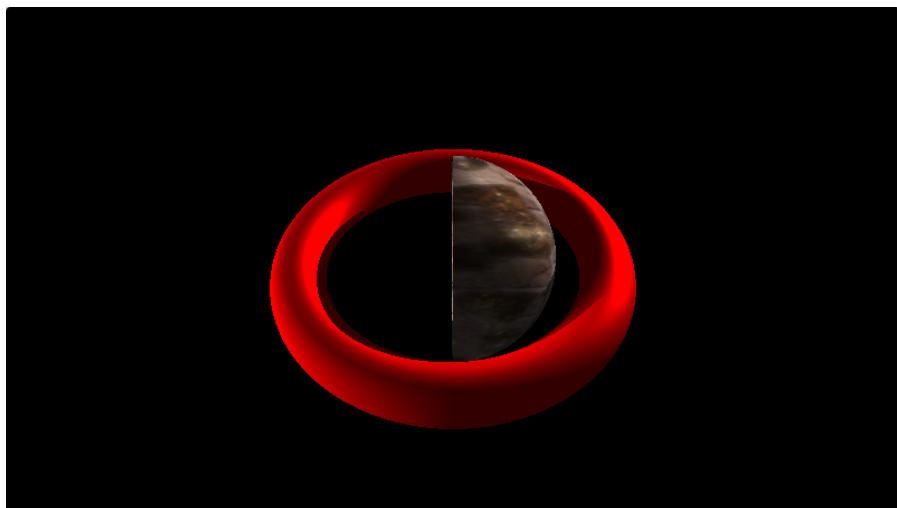


FIGURA 7.2: Textura asignada a la semiesfera



## 7. AGRADECIMIENTOS

---

Agradecer a mi tutor, Prof. Dr. Juan Carlos Torres Cantero, por la ayuda proporcionada en el desarrollo del proyecto y su corrección. Además, agradecer al profesor, Jesús García Miranda, por su constante apoyo durante estos cuatro años de carrera.

Agradecer a toda mi familia por confiar siempre en mi, ayudándome tanto sentimental como en lo económico, pudiéndo desarrollar unos estudios universitarios dignos que todos deberíamos recibir. Destacar a mis abuelos maternos, con los que he convivido desde los 9 años y a los que no podré agradecer lo suficiente.

Quiero agradecer a la persona que me inspiró a desarrollar este proyecto, Íñigo Quilez, divulgador del tema y uno de los creadores de la plataforma *Shadertoy*. Además de poner un granito de arena más a mi pasión por las matemáticas.

Finalmente, agradecer a mi amigo y compañero de la escuela, José Francisco Morales Garrido, que ha sido un pilar fundamental tanto emocional como personal, al que aprecio mucho.

## COLOFÓN

Este trabajo ha sido escrito en Granada en junio de 2020 y acabado en septiembre de 2020.

Ha sido compuesto utilizando L<sup>A</sup>T<sub>E</sub>X con el estilo proporcionado por el paquete **classicthesis**, desarrollado por André Miede e Ivo Pletikosić e inspirado en el del libro de Robert Bringhurst, «*The Elements of Typographic Style*». Puede obtenerse una copia de dicho paquete en

<https://bitbucket.org/amiede/classicthesis/>.

Las tipografías utilizadas han sido *EBGaramond* para el cuerpo, *Garamond Math* para las matemáticas, *Open Sans* para las leyendas y *Go Mono* para el código.