Trabajo de fin de grado

Funciones de distancia con signo

Lukas Häring García

UNIVERSIDAD
DE GRANADA

Índice General

Introducción

El rápido incremento en potencia de la unidad de procesamiento gráfico (*GPU*) ha permitido utilizar técnicas de renderizado propuestas en los años noventa. Presentaremos el lenguaje *GLSL* del que hace uso la tecnología web y que es utilizado durante todo el desarrollo del proyecto.

Presentaremos las *funciones de distancia con signo*, una serie de funciones del que hace uso la técnica de trazado *spheremarching* [Hart, 1996], presentada por John C. Hart en 1996 y del que centraremos nuestro trabajo.

Haremos uso del modelo empírico de *iluminación de Phong*, presentado por Thuong Phong en 1975, que es indispensable para dar realismo y sensación tridimensional a una escena, junto a los materiales.

Lenguaje GLSL

Tipos

Mantiene una sintaxis similar a C [John Kessenich, 2020], encontramos los siguientes tipos más importantes.

- **int**. Entero con signo.
- **float**. Número real, con precisión de 32 bits.
- **bool**. Ocupa un byte, *true* o *false*.
- **vecN**. Vector matemático, *N*-úpla de floats.
  Definidos: vec2, vec3, vec4.
- **matN**. Matriz cuadrada de dimension *N*.
  Encontramos: mat2, mat3, mat4.
- **matNxM**. Matriz de dimensiones $N \times M$.
  Encontramos: mat2x2, mat2x3, mat2x4, mat3x2, mat3x3, mat3x4, mat4x2, mat4x3, mat4x4.

Vectores

El tipo vector, vecN, definido por una t-úpla: $(x, y[, z[, w]])$ ó $(r, g[, b[, a]])$. Utilizaremos el operador «.» para acceder y copiar estas componentes.

Constructores

- vecN(float,⋯, float)
- vecN(vecM, float)
- vecN(float, vecM)
- vecN(vecP, vecQ)

Funciones

- length(vecN vector)
- distance(vecN p1, vecN p2)
- normalize(vecN vector)
- dot(vecN v1, vecN v2)
- cross(vecN v1, vecN v2)

Matrices

Las matrices *matNxM* y *matN*, formadas por $N \times M$ y N^2 componentes flotantes, 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*.

Constructores

- matNxM(float, $\cdots$, float)
- matNxM(float, $\cdots$, float)
- matN(vecN,$\cdots$, vecN)
- matNxM(vecM,$\cdots$, vecM)
- matN(matM)

Funciones

- transpose(mat matrix)
- matrix1 * matrix2
- determinant(matN matrix)

Operadores matemáticos

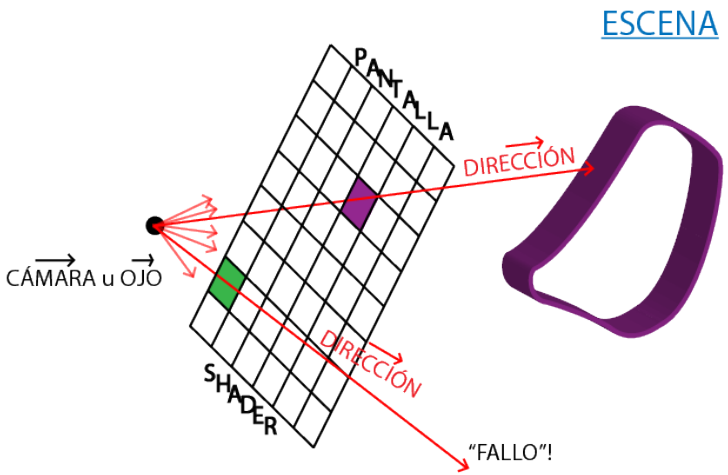
Agrupamos *float* y *vecN* con el nombre de *genType* para reunir los tipos de argumentos. Cuando utilizamos un operador sobre el tipo *vecN*, este se aplicará sobre cada una de sus componentes.

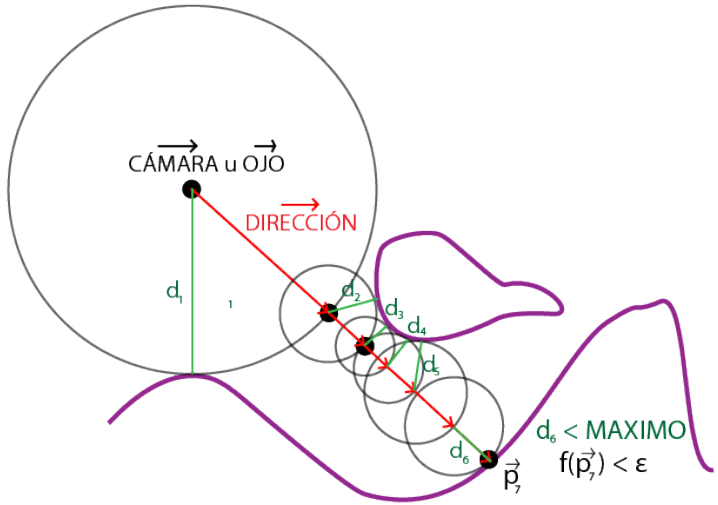
- radians(genType var)
- sin(genType var)
- tan(genType var)
- asin(genType var)
- atan(genType var)
- pow(genType a, genType b)
- exp(genType var)
- sqrt(genType var)
- sqrt(genType var)
- abs(genType a)
- sign(genType a)
- min(genType a, genType b)
- max(genType a, genType b)
- mix(
  genType a,
  genType b,
  (genType ó float ó bool) h
)

Spheremarcher

Spheremarcher

Un *fragment shader* es aplicado, que es procesado por una *hebra* de la *GPU*. «Lanzararemos un rayo», de manera numérica, para cada píxel y se aproxima la intersección, de manera iterativa, desde el ojo en la dirección del píxel dirección.





Spheremarcher 2

Como se ha comentado anteriormente, hacemos uso de las *funciones de distancia con signo* las cuales codifican la escena, $f : \mathbb{R}^3 \longrightarrow \mathbb{R}$. Definimos la posición del «rayo» en la iteración n-ésima, como:

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

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

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

Definition

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

Definition

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

Condiciones de parada

Al tratarse de un método numérico, vamos a definir las condiciones de parada:

1. **Primera condición**. Utilizaremos una variable de control, ϵ 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, $n \geq PASOS$. *PASOS* es una constante fijada.

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

Modelo de iluminación

Normal de una isosuperficie

Para un modelo de iluminación es indispensable el cálculo de la normal de una isosuperficie, por ello, vamos a presentar el siguiente teorema:

Theorem

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:

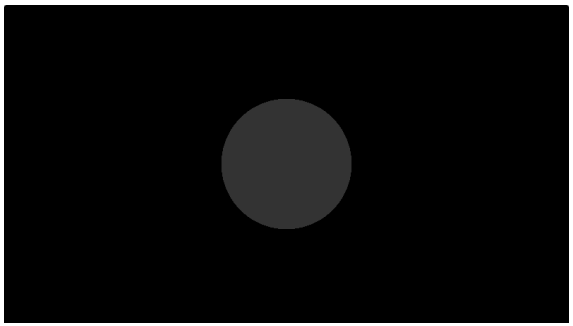
$$\vec{n} = norm(\nabla f(x, y, z)) \approx norm \left(\left\langle \begin{array}{c} \frac{f(x + 0.001, y, z) - f(x, y, z)}{0.001} \\ \frac{f(x, y + 0.001, z) - f(x, y, z)}{0.001} \\ \frac{f(x, y, z + 0.001) - f(x, y, z)}{0.001} \end{array} \right\rangle \right)$$

Intensidad lumínica

Para cada luz $\vec{l_i} \in L$, definimos el vector director de la luz hasta el punto $\vec{p}$ como $\vec{d_i} = \text{norm}(\vec{l_i} - \vec{p})$, la intensidad es un factor multiplicativo.

Intensidad ambiental

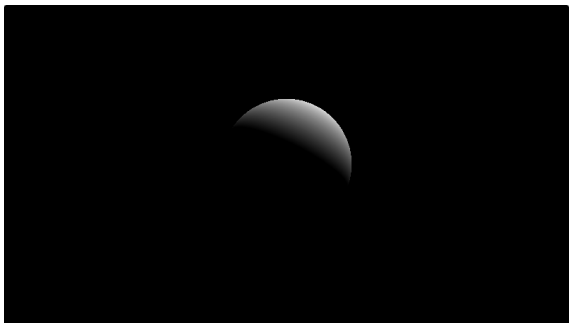
Intensidad mínima sobre la isosuperficie.



$$I_a \in [0, 1]$$

Intensidad difusa

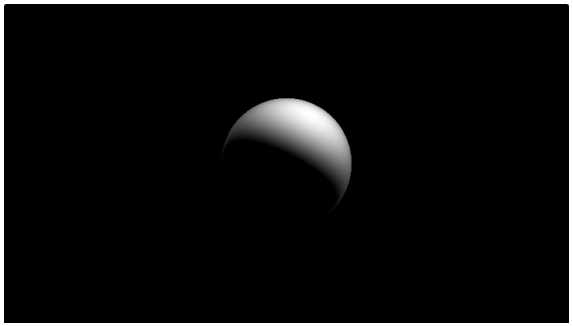
Intensidad por la luz refractada por la superficie.



$$I_d = \sum_{\vec{l_i} \in L} \vec{n} \cdot \vec{d_i}$$

Intensidad especular

Intensidad por la incidencia en el ojo de la luz reflectada.



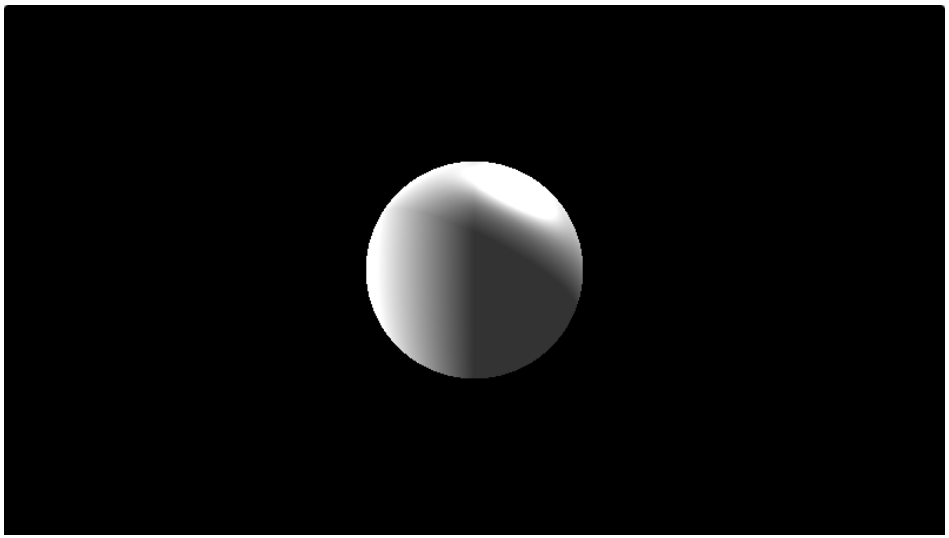
$$I_e = \sum_{\vec{l_i} \in L} o\vec{j}o \cdot \left(\vec{d_i} \veebar \vec{n}\right)$$

Modelo de Iluminación de Phong

Presentado por Thuong Phong en 1975 como un modelo empírico[Phong, 1975], resultado de las sumas de las intensidades anteriores, además, utiliza un *homeomorfismo* como factor de brillo para la intensidad especular:

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

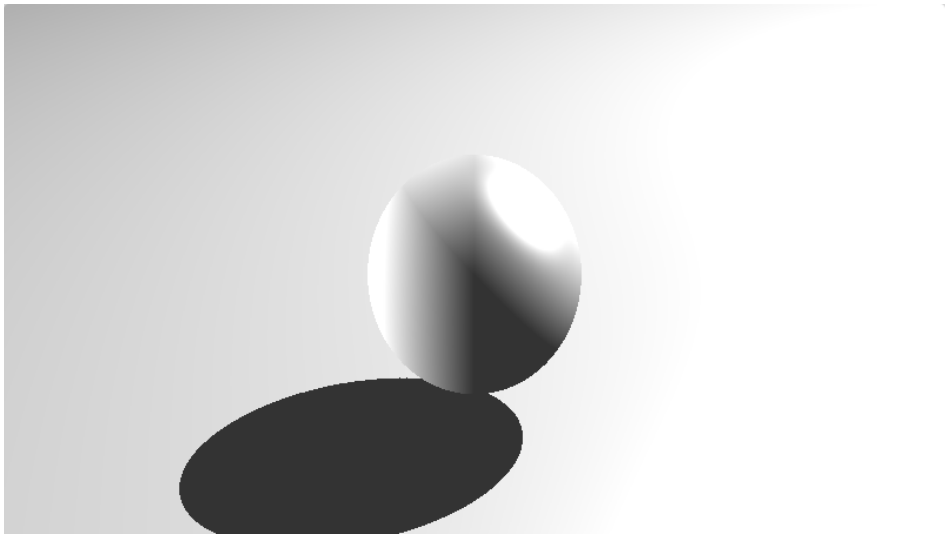
$$I_{Phong} = I_a + \sum_{\vec{l_i} \in L} \underbrace{\vec{n} \cdot_{[0,1]} (\vec{l_i} - \vec{p})}_{\text{Intensidad Difusa}} + \underbrace{h_k \left(\vec{ojo} \cdot_{[0,1]} \left(\left(\vec{l_i} - \vec{p} \right) \veebar \vec{n} \right) \right)}_{\text{Intensidad Especular}}$$



Umbra

Dado un punto punto $\vec{p}$ sobre la superficie, lanzaremos otro rayo hacia la luz para ver si este es ocluido, 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, por ello, separaremos el punto $\vec{p}$ de la superficie haciendo uso de la normal de la superficie y un factor de empuje $k \in \mathbb{R}_0^+$.

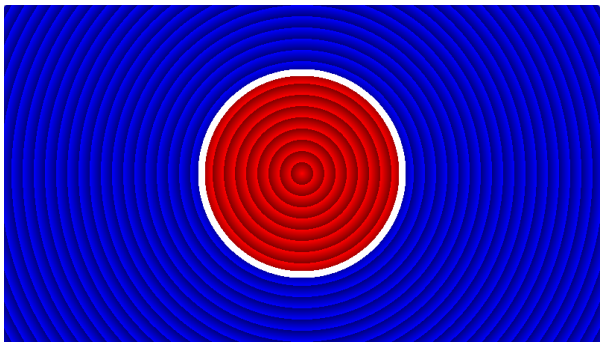
$$\vec{p'} = \vec{p} + \vec{n} \cdot k$$



Funciones de distancia con signo (FDS)

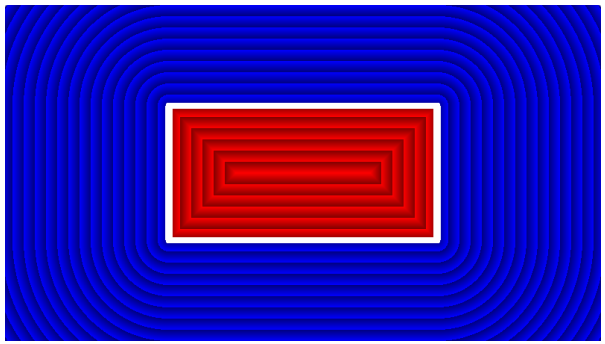
Primitivas sobre $\mathbb{R}^2$

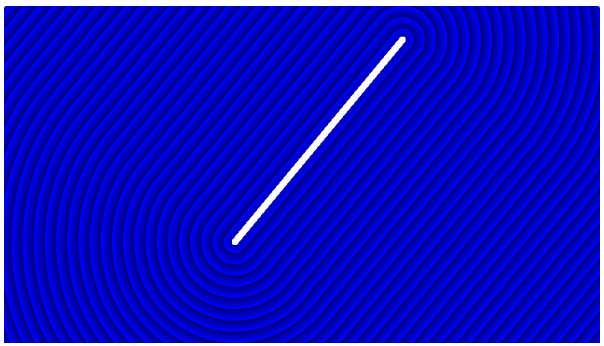
Funciones de distancia con signo de [Quilez, 2018] y [Quilez, 2011],



```
float SDFCircunsferencia(vec2 p, float r){
    return length(p) - r;
}
```

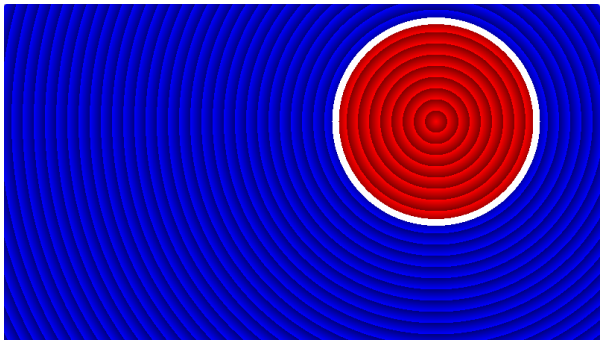
```
float SDFRectangulo(vec2 p, vec2 s){
        vec2 a = abs(p) - s;
        float extr = length(max(a, 0.0));
        float intr = min(max(a.x, a.y), 0.0);
        return extr + intr;
}
```





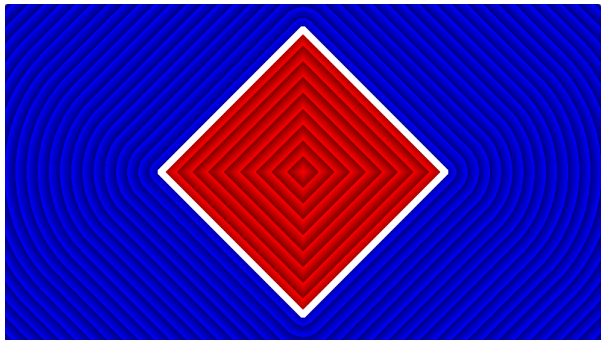
```
vec2 proy01(in vec2 a, in vec2 b){
    return b * clamp(dot(b, a) / dot(b, b), 0., 1.);
}
float SDFSegmento(vec2 p, vec2 a, vec2 b){
    vec2 v = p - a;
    vec2 w = b - a;
    return length(v -  proy01(v, w));
}
```

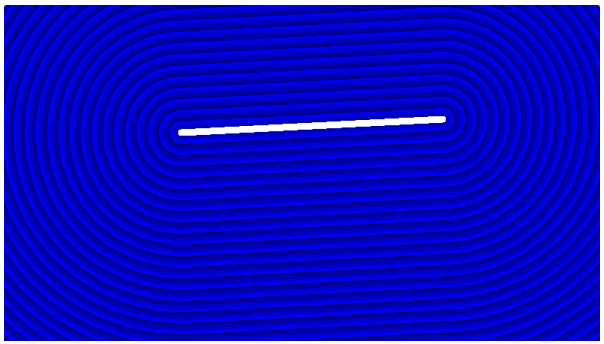
Operadores sobre $\mathbb{R}^2$



```
float escena_sdf(vec2 p){
    vec2 pt = p - vec2(0.1, 0.2);
    return SDFCircunsferencia(pt, 0.3);
}
```

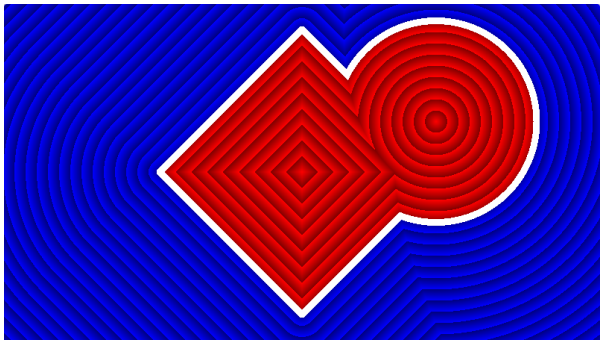
```
#define PI 3.1415
mat2 rot(float a){
    return mat2(+cos(a), -sin(a), +sin(a), +cos(a));
}
float escena_sdf(vec2 p){
    vec2 pr = p * rot(45. * PI / 180.);
    return SDFRectangulo(pr, vec2(0.3));
}
```





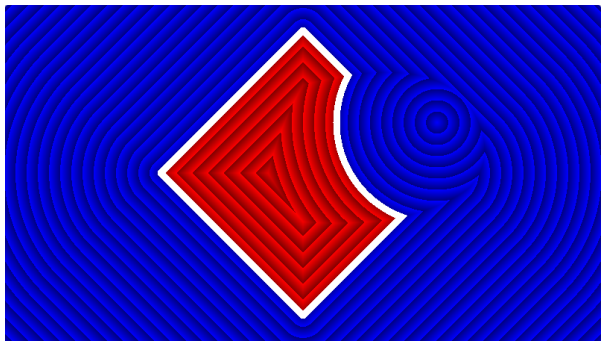
```
vec2 simetria(vec2 p, vec2 a, vec2 b){
    return a + simetria(b - a, p - a);
}
float escena_sdf(vec2 p){
    vec2 a = vec2(0.2, 0.2), b = vec2(0.0, 0.1);
    vec2 ps = simetria(p, a, b);
    return SDFSegmento(ps, vec2(-0.2, -0.2), vec2(0.3, 0.4));
}
```

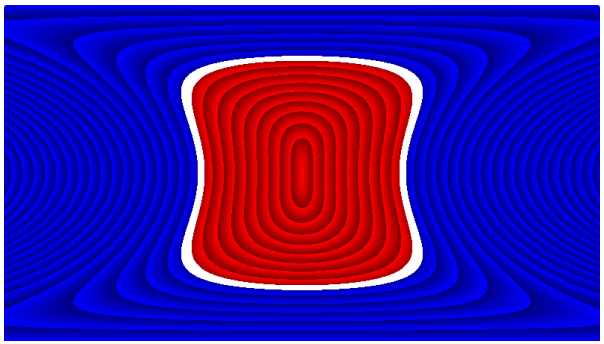
Operadores sobre $\mathbb{R}^2$



```
float escena_sdf(vec2 p){
    vec2 pr = p * rot(PI / 180. * 45.0);
    vec2 pt = p - vec2(0.4, 0.15);
    return min(
        SDFRectangulo(pr, vec2(0.3)), // f
        SDFCircunsferencia(pt, 0.3)   // g
    );
}
```

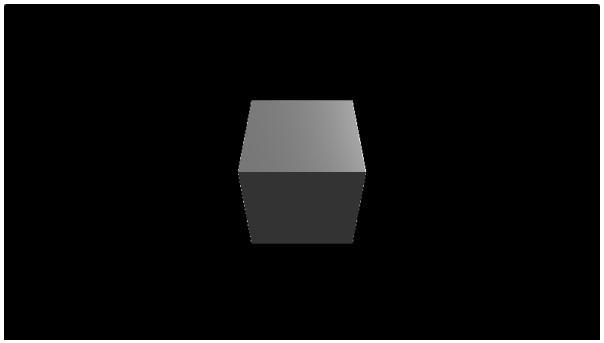
```
float escena_sdf(vec2 p){
    vec2 pr = p * rot(PI / 180. * 45.0);
    vec2 pt = p - vec2(0.4, 0.15);
    return max(
        SDFRectangulo(pr, vec2(0.3)),
       -SDFCircunsferencia(pt, 0.3)
    );
}
```





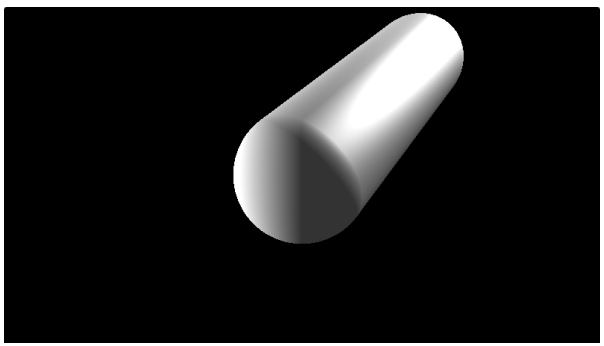
```
float sdf(vec2 p){
    vec2 pn = vec2(
        p.x * cos(p.y * PI),
        p.y * sin(p.y * PI)
    );
    return SDFCircunferencia(pn, 0.1);
}
```

Primitivas sobre $\mathbb{R}^3$



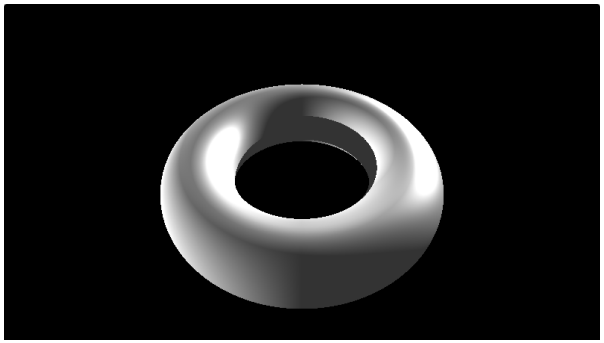
```
float SDFPrisma(vec3 p, vec3 s){
    vec3 pa = abs(p) - s;
    return length(max(pa, 0.)) +
    min(max(max(pa.x, pa.y), pa.z), 0.);
}
```

```
float SDFPlano(vec3 p, vec3 n){
   return dot(p, n);
}
```



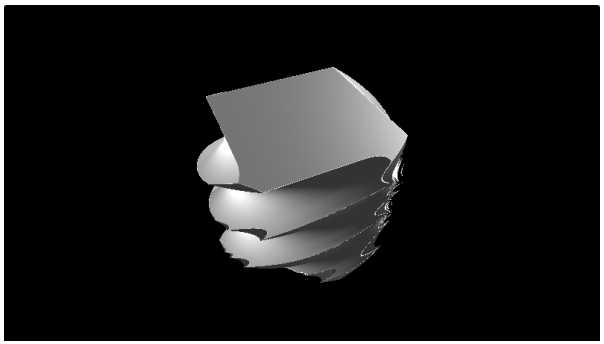
```
float SDFSegmento(vec3 p, vec3 a, vec3 b){
    vec3 v = p - a;
    vec3 w = b - a;
    return length(v -  proy01(v, w));
}
float SDFCapsula(vec3 p, vec3 a, vec3 b, float k){
    return SDFSegmento(p, a, b) - k;
}
```

Operadores sobre $\mathbb{R}^3$



```
float SDFToro(vec3 p, float rx, float r){
    vec2 rev = vec2(length(p.xz), p.y);
    vec2 pt = rev - vec2(rx, 0.);
    return SDFCircunferencia(pt, r);
}
```

```
float SDFCilindro(vec3 p, float r){
    vec3 n = normalize(vec3(1, 0, 0));
    vec2 proy = proyPlano(p, n).yz;
    return SDFCircunferencia(proy, r);
}
```

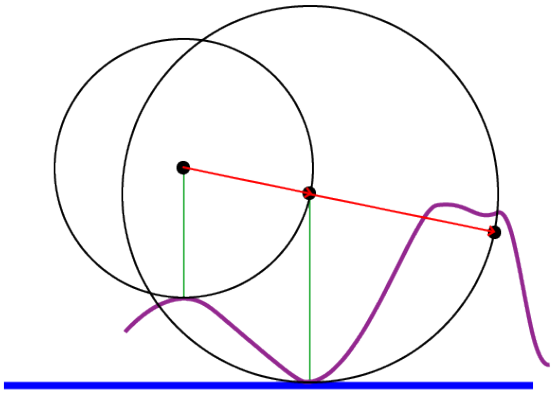


```
float escena_sdf(vec3 p){
    vec2 ry = p.yz * rot(PI / 4.0);
    p = vec3(p.x, ry.x, ry.y);
    float a = p.y * 10.0;
    p = vec3(
        +p.x * cos(a) + p.z * sin(a),
        +p.y,
        -p.x * sin(a) + p.z * cos(a)
    );
    p.xz * rot(a)
    return SDFPrisma(p, vec3(0.2));
}
```

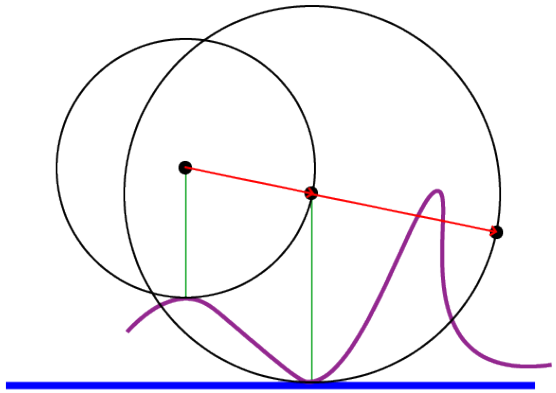
Resolución de artefactos

Sobreestimación

Cuando tratamos *funciones de distancia con signo no exactas*, utilizaremos el término «sobreestimar» cuando se supera la distancia mínima real a la superficie a lo que llamaremos «artefacto». El rayo trazado se encuentra dentro de la superficie o el rayo atraviese una superficie.



1) Estimación dentro de la superficie.



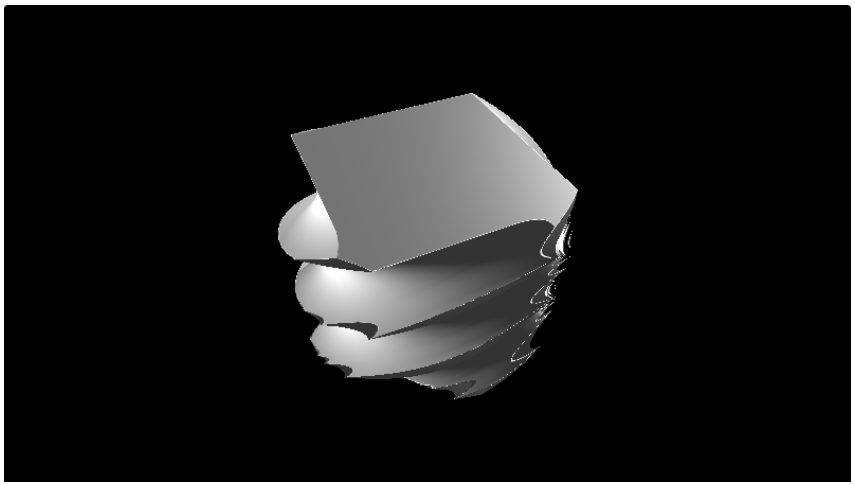
2) Estimación fuera de la superficie.

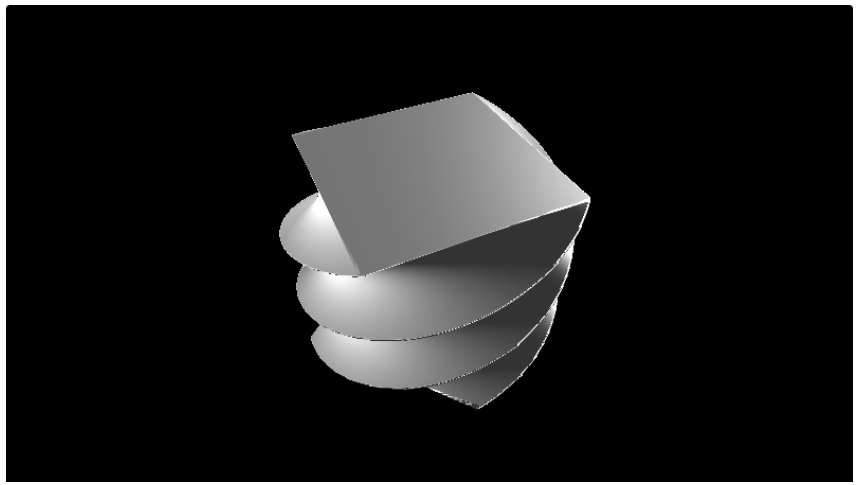
Solución para la sobreestimación

Vamos a modificar la condición de parada impuesta que definía «estar sobre la isosuperficie». Ahora, diremos que estamos sobre una superficie, si y solo si, $|f(\vec{rayo}_n)| < \epsilon$, así, si nos encontramos dentro, deberá salir del objeto, gracias al signo de la distancia.

1. Incrementar el número de iteraciones para nuestro algoritmo.
2. Escalar $k \in [0, 1]$ el radio de la bola, es decir, la distancia más corta a la superficie.

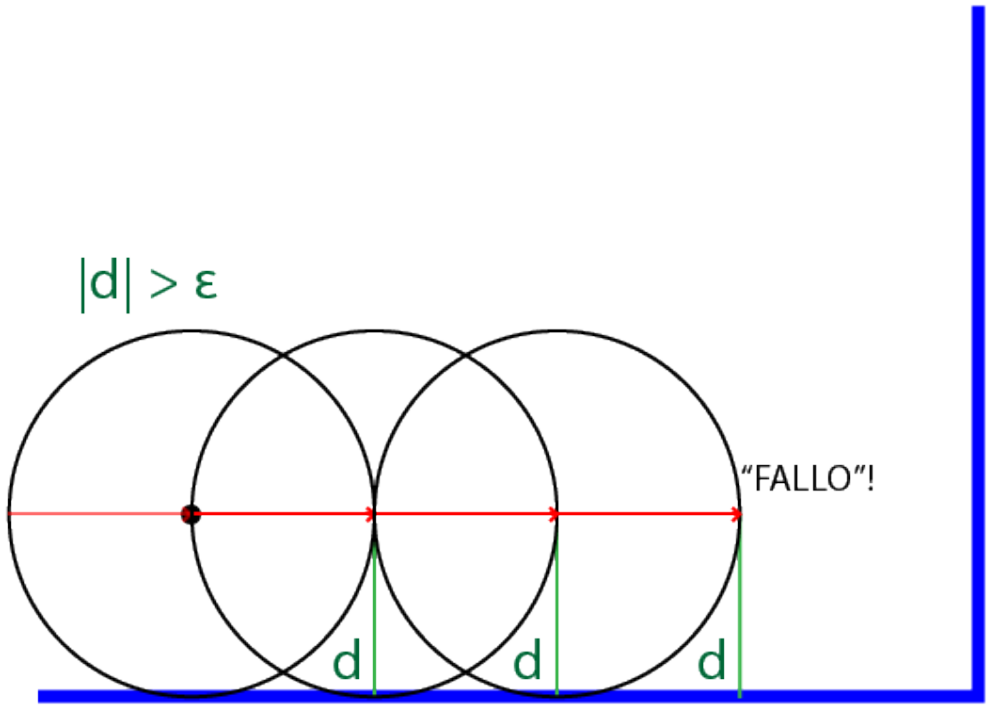
$$d'_n = d_{n-1} + f(\vec{p}_{n-1}) \cdot k \leq d_n$$





Subestimación

Este tipo de estimación puede ocurrir tanto en *funciones de distancia con signo exactas como inexactas*. Cuando el *Marcher* finaliza consumiendo todas las iteraciones disponibles, es decir, en la **Tercera condición**. Suele ocurrir cuando el rayo pasa de manera paralela, muy cerca a una superficie con $f(\vec{rayo}_n) \geq \epsilon$. La solución trivial es incrementar el número de iteraciones.



Materiales

Materiales

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 $id = 0$ cuando no se ha trazado ningún objeto. Esto hace que *f*, nuestra escena, esté definida como,

$$f : \mathbb{R}^3 \longrightarrow \mathbb{R} \times \mathbb{N}$$

El pixel resultado, será:

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

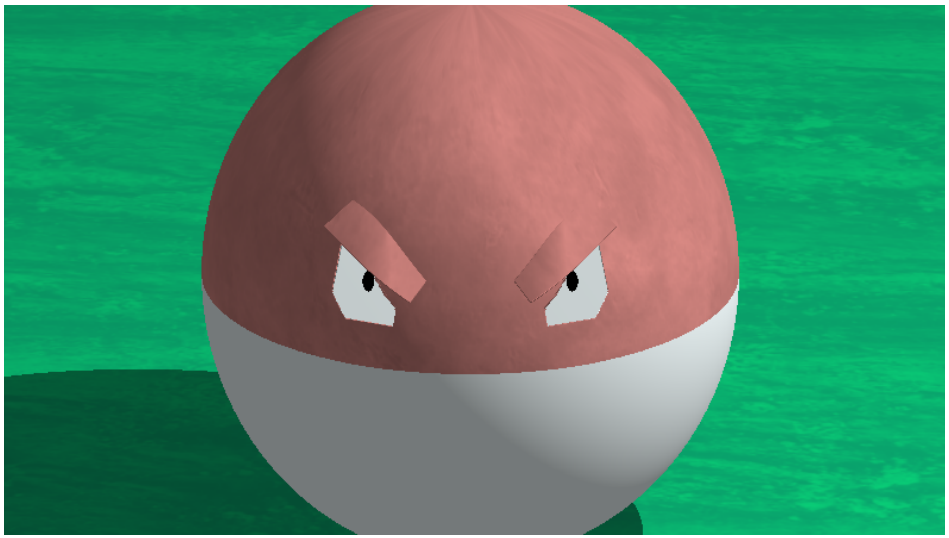
También podemos utilizar el punto $\vec{p}$ y el identificador *id* para calcular la proyección hacia el sistema de coordenadas de alguna textura y tener así una escena más rica.

Materiales

Utilizando la ecuación de la proyección cilíndrica:

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

Se ha creado la siguiente imagen:



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

Conclusiones

Conclusiones

- Se trata de una técnica novedosa y con un ámplio campo de estudio, que requiere de un elevado conocimiento matemático.
- El modelo de iluminación es esencial para la creación de escenas tridimensionales.
- Utilizar funciones de distancia con signo exactas, que ayudan a la convergencia del algoritmo.
- La sub/sobreestimación, requiere de un mayor ejercicio computacional.
- Los materiales dan una riqueza visual al ejercicio, en caso de texturización, utilizaremos proyecciones sobre las coordenadas (u, v).

References I

[Hart, 1996] Hart, J. C. (1996).
Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces.
The Visual Computer, 12(10):527–545.

[John Kessenich, 2020] John Kessenich, Dave Baldwin, R. R. (2020 (accessed September 02, 2020)).
The OpenGL© Shading Language.
https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.40.pdf.

[Phong, 1975] Phong, B. T. (1975).
Illumination for computer generated pictures.
Communications of the ACM, 18(6):311–317.

References II

[Quilez, 2011] Quilez, I. (2011).
distance functions.
https://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm.

[Quilez, 2018] Quilez, I. (2018).
2d distance functions.
https://www.iquilezles.org/www/articles/distfunctions2d/distfunctions2d.htm.

La plantilla utilizada: https://github.com/martinhelso/UiB