UNIVERSIDAD
DE GRANADA

ETSIIT, Facultad de Ciencias

Doble Grado en Ing. Informática y Matemáticas

TRABAJO DE FIN DE GRADO

Visualización de Superficies 3D

Presentado por:
Daniel Zufrí Quesada

Tutor:
Carlos Ureña Almagro
Departamento de Lenguajes y Sistemas Informáticos

Pedro A. García Sánchez
Departamento de Álgebra

Curso académico 2022-2023

Índice general

1. Función Distancia con Signo **1**
- 1.1. Operaciones sobre SDF . . . 2
- 1.2. Renderizado . . . 8
  - 1.2.1. Creación del lienzo . . . 8
  - 1.2.2. Raymarching y Spheretracing . . . 11
  - 1.2.3. Iluminación . . . 14
- 1.3. Aproximación de implícitas . . . 17

2. Implicitación **19**
- 2.1. Polinomios en varias variables . . . 19
- 2.2. Bases de Groebner . . . 20
- 2.3. Implicitación Polinomial . . . 24

3. Desarrollo de la aplicación **25**
- 3.1. Librería de polinomios multivariable . . . 25
  - 3.1.1. Clase `Monomial` . . . 25
  - 3.1.2. Clase `Polynomial` . . . 25
  - 3.1.3. Clase `Ideal` . . . 25

A. Resultados de operaciones sobre SDF **27**

Bibliografía **29**

1. Función Distancia con Signo

Tenemos como objetivo representar superficies en $\mathbb{R}^3$. En general, estas pueden estar definidas de multitud de formas, siendo una de las más usuales a través de una función implícita. Nosotros nos centraremos en un tipo especial de funciones implícitas, que introducimos a continuación.

Definición 1.1. Sea $\Omega \subseteq \mathbb{R}^3$. Una **función distancia** es aquella que a cada punto de $\mathbb{R}^3$ le asigna su menor distancia a la frontera de Ω:

$$\begin{aligned} d_\Omega \colon \mathbb{R}^3 &\to \mathbb{R}_0^+. \\ x &\mapsto \inf\{\|x-y\|) : y \in \delta\Omega\}. \end{aligned}$$

Cuando Ω sea cerrado, podremos usar el mínimo en lugar del ínfimo.

Definición 1.2 (SDF)**.** Sea $\Omega \subseteq \mathbb{R}^3$. Una **función distancia con signo** es una función de la forma:

$$\begin{aligned} \phi_\Omega \colon \mathbb{R}^3 &\to \mathbb{R}. \\ x &\mapsto \begin{cases} d_\Omega(x), & x \in \mathbb{R}^3 \setminus \mathring{\Omega}. \\ -d_\Omega(x), & x \in \mathring{\Omega}. \end{cases} \end{aligned}$$

En general, nos referiremos a esta función por sus siglas en inglés SDF (*Signed Distance Function*), y la denotaremos simplemente ϕ siempre que no haya confusión.

Definición 1.3. Dada una función $\phi \colon \mathbb{R}^3 \to \mathbb{R}$ y $k \in \mathbb{R}$, llamamos **isosuperficie de ϕ con valor k** al conjunto:

$$S_{\phi,k} = \{(x,y,z) : \phi(x,y,z) = k\}.$$

Sin pérdida de generalidad podemos suponer $k = 0$, pues de no ser el caso, tomamos la función $\phi'(x,y,z) = \phi(x,y,z) - k$ y tenemos que $S_{\phi',0} = S_{\phi,k}$. Por tanto, la denotaremos como S_ϕ.

Nuestra intención será entonces construir una escena definida como la isosuperficie generada por un SDF. A partir de ahora, tomaremos $p = (x,y,z) \in \mathbb{R}^3$.

Ejemplo 1.1. Ejemplos simples de SDF ϕ en p para diferentes conjuntos Ω son:

- **Esfera de radio r centrada en el origen.**

$$\Omega = \{x \in \mathbb{R}^3 : \|x\| = r\}, \quad \phi(p) = \|p\| - r.$$

- **Plano con vector normal unitario $n = (a,b,c)$ y pasando por el origen**.

$$\Omega = \{p \in \mathbb{R}^3 : ax + by + cz = 0\}, \quad \phi(p) = p \cdot n.$$

- **Toro de radios R y r, con $R > r$**:

$$\Omega = \left\{ p \in \mathbb{R}^3 : \left(R - \sqrt{x^2 + y^2} \right)^2 + z^2 = r^2 \right\}, \quad \phi(p) = \left\| (\|(x, 0, z)\| - R, y) \right\| - r.$$

1.1. Operaciones sobre SDF

Si bien estas primitivas son fáciles de generar, también son muy simples y nos serán insuficientes si queremos construir escenas más complejas. Una de las ventajas de los SDF es que se pueden generar nuevas formas a partir de primitivas de forma muy sencilla. Para ello, una de las técnicas más útiles es la geometría de sólidos constructiva, que utiliza operaciones booleanas para combinar múltiples primitivas. Por la naturaleza de los SDF, estas operaciones se implementan fácilmente usando las funciones máx y mín.

Definición 1.4 (Operaciones Booleanas)**.** Sean A y B isosuperficies generadas por ϕ y γ respectivamente. Definimos los SDF para las siguientes operaciones.

- **Unión:** $\mathrm{sdf}_{A \cup B}(p) = \mathrm{mín}(\phi(p), \gamma(p))$,
- **Intersección:** $\mathrm{sdf}_{A \cap B}(p) = \mathrm{máx}(\phi(p), \gamma(p))$,
- **Diferencia:** $\mathrm{sdf}_{A \setminus B}(p) = \mathrm{máx}(\phi(p), -\gamma(p))$.

Observación 1.1. Solo en el caso de la unión se obtiene un SDF según lo establecido en la Def. 1.2, ya que al aplicar máx en el interior de la superficie (donde $\phi(p) < 0$), el resultado puede ser solo una cota inferior de la distancia. En nuestro caso solo estamos interesados en visualizar la frontera de las superficies así que podemos obviar este problema, con la salvedad de que el algoritmo de *raymarching* requiera de más iteraciones.

Un problema de usar estas transformaciones es que produce discontinuidades en la derivada del SDF resultante. Trataremos de evitar esta situación, además de por motivos analíticos, por motivos visuales, ya que esto produce bordes muy acusados en la intersección de ambas superficies. Existen muchas formas de combinar SDF de forma más natural. Usaremos una de las más extendidas, usada por programas de modelado 3D como Blender [1] o videojuegos como Dreams [2], y que ha sido estudiada por Íñigo Quílez en su web [3].

Observación 1.2. Para mayor claridad del razonamiento, en las figuras se representarán funciones de variable real, a pesar de que nosotros trabajamos en $\mathbb{R}^3$.

Empezamos trabajando con la unión. La idea es, dadas ϕ y γ, añadir una corrección para cada punto a la función mín original para que cumpla ciertos requisitos. Por comodidad, definiremos

$$\begin{aligned} \mathrm{mín}_{\phi,\gamma} \colon \mathbb{R}^3 &\to \mathbb{R}, \\ p &\mapsto \mathrm{mín}(\phi(p), \gamma(p)), \end{aligned}$$

y siguiendo un abuso de notación, escribiremos $\mathrm{mín}(p)$ cuando no haya lugar a dudas.

Llamaremos a la mencionada corrección $\omega \colon \mathbb{R}^3 \times \mathbb{R}_0^+ \to \mathbb{R}$, de forma que la versión suavizada de la función mín original será

$$\begin{aligned} \mathrm{smín}_{\phi,\gamma} \colon \mathbb{R}^3 \times \mathbb{R}_0^+ &\to \mathbb{R}, \\ (p, k) &\mapsto \mathrm{mín}(p) - \omega(p, k), \end{aligned}$$

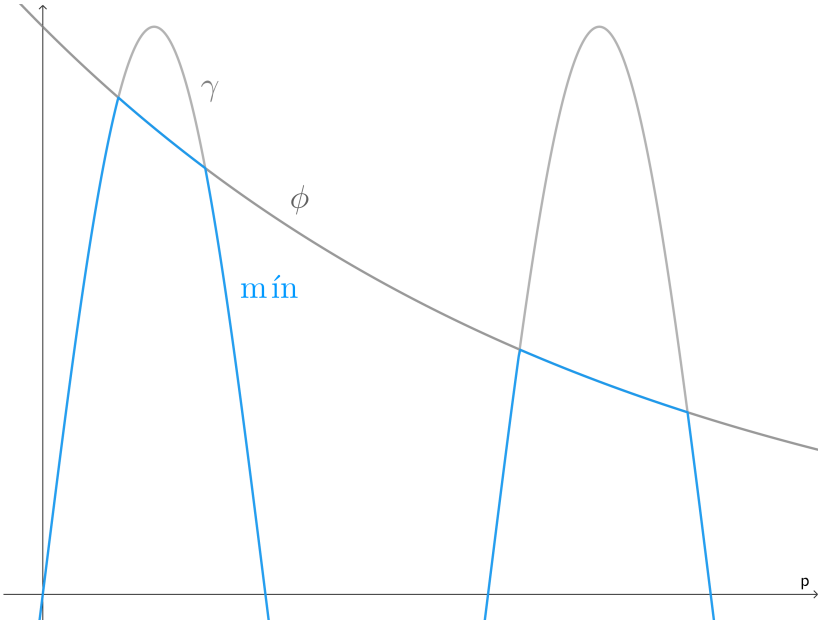

Figura 1.1.: Gráfica de mín: $\mathbb{R} \to \mathbb{R}$

donde la variable $k \in \mathbb{R}_0^+$ controlará la intensidad del suavizado. Siempre que no haya confusión, denotaremos $\text{smín}_{\phi,\gamma} = \text{smín}$.

Como no queremos que este cambio afecte al algoritmo de *raymarching*, debemos asegurar que se cumpla $\text{mín}(p) \geq \text{smín}(p, k)$, esto es,

$$\omega(p) \geq 0,\ \forall p \in \mathbb{R}^3,\ \forall k \in \mathbb{R}_0^+.$$

Si estudiamos como se comporta la versión real de mín en la Figura 1.1, vemos que los puntos de conflicto se encuentran cerca de las intersecciones de ϕ y γ, es decir, cuando ϕ y γ están arbitrariamente cerca. En el resto de puntos no queremos modificar la función original, luego estudiamos el comportamiento de smín en el conjunto de entornos de las intersecciones

$$B_k = \{p \in \mathbb{R}^3 : |\phi(p) - \gamma(p)| \leq k\},$$

siendo $\omega(p) = 0$ cuando $p \notin B_k$.

Para asegurar que smín sea continua en la frontera de B_k, imponemos la condición

$$\omega(p) = 0,\ \forall p \in \delta B_k.$$

Por otro lado, es lógico que ω tenga su mayor influencia justo en las intersecciones, luego imponemos también

$$\omega(c) = s,\ \text{donde } c \in I \equiv \{p \in \mathbb{R}^3 : \phi(p) = \gamma(p)\},\ s \in \mathbb{R}.$$

El valor s es el que deberemos ajustar para que smín cumpla nuestros requisitos. Fijado

un $p \in B_k$, ya tenemos una primera aproximación para ω :

$$\omega(p,k) = s\left(1 - \frac{|\phi(p) - \gamma(p)|}{k}\right)^n = \begin{cases} s\left(1 - \frac{\phi(p)-\gamma(p)}{k}\right)^n, & \phi(p) > \gamma(p), \\ s\left(1 + \frac{\phi(p)-\gamma(p)}{k}\right)^n, & \phi(p) \leq \gamma(p) \end{cases} \quad , s \in \mathbb{R},\ n \in \mathbb{N},$$

donde hemos añadido el parámetro n para añadir más control sobre el resultado final.

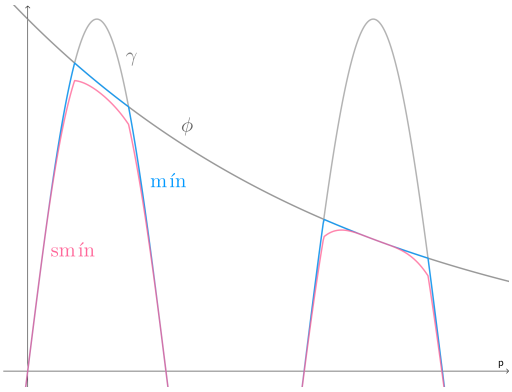

Figura 1.2.: $k = 0.6$

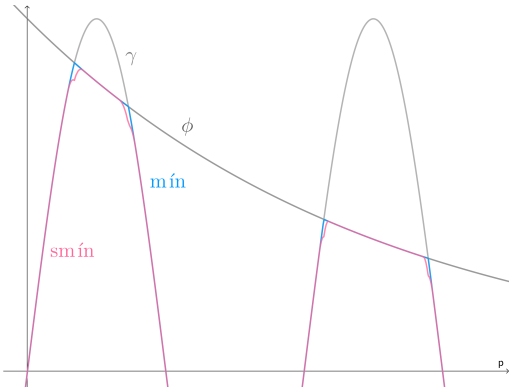

Figura 1.3.: $k = 0.1$

Figura 1.4.: Primera aproximación de $smin(p,k)$ con $s = 0.05$ y $n = 2$

Es evidente que ω está bien definida, pues:

$$\phi(p) = \gamma(p) \implies \frac{\phi(p) - \gamma(p)}{k} = 0 \implies \omega(p) = s$$

Ya podemos pasar a solucionar el problema de continuidad de la derivada. Esta tiene la forma:

$$\begin{aligned} \text{smín}'(p,k) &= (\text{mín}(\phi(p), \gamma(p)))' - \omega(p,k)' \\ &= \begin{cases} \gamma'(p) + sn\left(1 - \frac{\phi(p)-\gamma(p)}{k}\right)^{n-1}\left(\frac{\phi'(p)-\gamma'(p)}{k}\right), & \phi(p) > \gamma(p), \\ \phi'(p) - sn\left(1 + \frac{\phi(p)-\gamma(p)}{k}\right)^{n-1}\left(\frac{\phi'(p)-\gamma'(p)}{k}\right), & \phi(p) \leq \gamma(p). \end{cases} \end{aligned}$$

Veamos cuándo está bien definida:

$$\phi' - sn\left(1 + \frac{\phi - \gamma}{k}\right)^{n-1}\left(\frac{\phi' - \gamma'}{k}\right) = \gamma' + sn\left(1 - \frac{\phi - \gamma}{k}\right)^{n-1}\left(\frac{\phi' - \gamma'}{k}\right)$$

Evaluando en $c \in I$:

$$\cancel{\phi'(c) - \gamma'(c)} = 2sn\left(1 + \frac{\phi(c) - \gamma(c)}{k}\right)^{n-1}\left(\frac{\cancel{\phi'(c) - \gamma'(c)}}{k}\right);$$

$$s = \frac{k}{2n}\left(1 - \frac{\cancel{\phi(c) - \gamma(c)}}{k}^{0}\right);$$

$$s = \frac{k}{2n}.$$

Hemos llegado a la expresión final

$$\omega(p,k) = \begin{cases} \frac{k}{2n}\left(1 - \frac{|\phi(p)-\gamma(p)|}{k}\right)^n, & |\phi(p) - \gamma(p)| \leq k, \\ 0, & \text{otro caso .} \end{cases}$$

$$= \frac{\max\left(k - |\phi(p) - \gamma(p)|, 0\right)^n}{2n \cdot k^{n-1}},\ s \in \mathbb{R},\ n \in \mathbb{N}.$$

Podemos observar los resultados en la Figura 1.7.

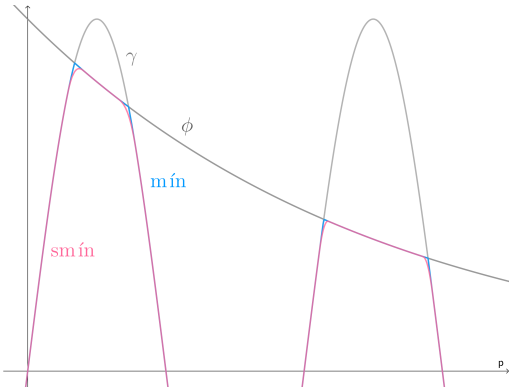

Figura 1.5.: $k = 0.1,\ n = 2$

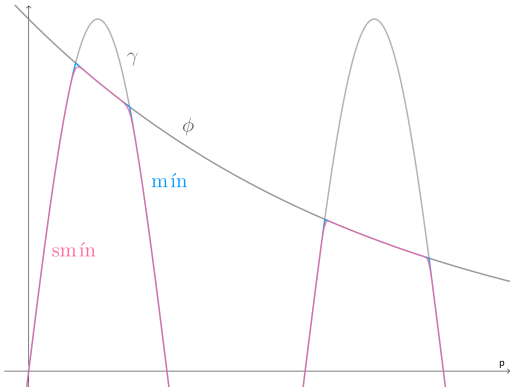

Figura 1.6.: $k = 0.1,\ n = 3$

Figura 1.7.: Resultado final de $smin(p,k)$

Finalmente, para obtener una versión suavizada del máximo, es fácil comprobar que

$$\begin{aligned} \text{smáx}_{\phi,\gamma} \colon \mathbb{R}^3 \times \mathbb{R}_0^+ &\to \mathbb{R}, \\ (p,k) &\mapsto -\text{smín}_{-\phi,-\gamma}(p,k). \end{aligned}$$

Recogemos los resultados obtenidos a continuación.

Definición 1.5 (Operaciones Booleanas Suavizadas)**.** Sean A y B isosuperficies generadas por ϕ y γ respectivamente. Definimos los SDF para las operaciones booleanas suavizadas como sigue.

- **Unión suavizada:** $\text{sdf}_{unionS}(p) = \text{mín}(\phi(p), \gamma(p)) - \frac{\text{máx}(k - |\phi(p)-\gamma(p)|, 0)^n}{2n \cdot k^{n-1}}$,

- **Intersección suavizada:** $\text{sdf}_{interS}(p) = -\text{mín}(-\phi(p), -\gamma(p)) + \frac{\text{máx}(k-|\phi(p)-\gamma(p)|,0)^n}{2n\cdot k^{n-1}}$,
- **Diferencia suavizada:** $\text{sdf}_{difS}(p) = -\text{mín}(-\phi(p), \gamma(p)) + \frac{\text{máx}(k-|\phi(p)+\gamma(p)|,0)^n}{2n\cdot k^{n-1}}$,

donde $k \in \mathbb{R}_0^+$ controla la influencia del suavizado.

Observamos que los operadores definidos en la Def. 1.4 no son más que un caso particular de estos últimos cuando $k \to 0$.

Pasamos ahora a estudiar otro tipo de operaciones, que nos resultarán útiles para generar nuevas formas a partir de un único SDF. Todas ellas se basarán en, dada ϕ, aplicar una transformación $t : \mathbb{R}^3 \to \mathbb{R}^3$ a cada punto de la isosuperficie S_ϕ para obtener una nueva S_γ. Si queremos saber si un punto $q \in \mathbb{R}^3$ está en S_γ, tenemos que comprobar si su preimagen por la transformación pertenece a S_ϕ. Por tanto, bastará evaluar el SDF original en $t^{-1}(p)$:

$$\gamma(p) = \phi(t^{-1}(p)).$$

Esto funciona bien para transformaciones como las traslaciones o rotaciones, ya que mantienen las distancias. Sin embargo, este no es el caso del escalado, ya que si tomamos $l(p) = sp$ con $s \in \mathbb{R}_0^+$:

$$\|p - p'\| = d \implies \|l(p) - l(p')\| = \|sp - sp'\| = s\|p - p'\| = s \cdot d, \text{ donde } p, p' \in S_\phi.$$

Como las distancias se escalan, deberemos hacer lo propio con el nuevo SDF, aplicándole el mismo factor de escalado s como muestra la Def. 1.6.

Definición 1.6 (Operaciones afines)**.** Sea A una isosuperficie. Definimos los SDF para las siguientes operaciones.

- **Traslación de vector v:** $\text{sdf}_{traslacion}(p) = \text{sdf}_A(p - v)$,
- **Escalado uniforme de dimensiones s:** $\text{sdf}_{escalado}(p) = \text{sdf}_A(p/s) \cdot s$,
- **Rotaciones de ángulo α sobre los ejes x, y, z:**

$$\text{sdf}_{rotX}(p) = \text{sdf}_A(R_x^{-1}(\alpha) \cdot p^t),\ R_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) \\ 0 & \sin(\alpha) & \cos(\alpha) \end{pmatrix},$$

$$\text{sdf}_{rotY}(p) = \text{sdf}_A(R_y^{-1}(\alpha) \cdot p^t),\ R_y(\alpha) = \begin{pmatrix} \cos(\alpha) & 0 & \sin(\alpha) \\ 0 & 1 & 0 \\ -\sin(\alpha) & 0 & \cos(\alpha) \end{pmatrix},$$

$$\text{sdf}_{rotZ}(p) = \text{sdf}_A(R_z^{-1}(\alpha) \cdot p^t),\ R_z(\alpha) = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Siguiendo el mismo razonamiento, podemos definir operaciones que modifiquen la geometría de la superficie.

Definición 1.7 (Operaciones Deformantes)**.** Sea A una isosuperficie. Definimos los SDF para las siguientes operaciones.

- **Torsión:** $\mathrm{sdf}_{torsion}(p) = \mathrm{sdf}_A(p')$, con $p' = R_z(ky) \cdot (x, z, y)^t$,
- **Plegado:** $\mathrm{sdf}_{plegado} = \mathrm{sdf}_A(p')$, con $p' = R_z(kx) \cdot p^t$,
- **Redondeo:** $\mathrm{sdf}_{redondeo}(p) = \mathrm{sdf}_A(p) - k$,
- **Desplazamiento:** $\mathrm{sdf}_{desplazamiento}(p) = \mathrm{sdf}_A(\delta(p))$.

donde

- $k \in \mathbb{R}_0^+$ controla la intensidad de la deformación,
- $\delta\colon \mathbb{R}^3 \to \mathbb{R}^3$ es un patrón de desplazamiento,
- $R_z(\alpha) \in \mathcal{M}_3(\mathbb{R})$ es la matriz de rotación de ángulo α sobre el eje z dada en la Def. 1.6.

Introducimos por último operaciones que permiten generar copias de otras superficies.

Definición 1.8 (Operadores de Posicionamiento)**.** Sea A una isosuperficie. Definimos los SDF para las siguientes operaciones.

- **Simetrías sobre los ejes x, y, z:**

$$\mathrm{sdf}_{simX}(p) = \mathrm{sdf}_A(|x|, y, z), \quad \mathrm{sdf}_{simY}(p) = \mathrm{sdf}_A(x, |y|, z),$$
$$\mathrm{sdf}_{simZ}(p) = \mathrm{sdf}_A(x, y, |z|),$$

- **Simetrías sobre los planos xy, xz, yz:**

$$\mathrm{sdf}_{simXY}(p) = \mathrm{sdf}_A(|x|, |y|, z), \quad \mathrm{sdf}_{simXZ}(p) = \mathrm{sdf}_A(|x|, y, |z|),$$
$$\mathrm{sdf}_{simYZ}(p) = \mathrm{sdf}_A(x, |y|, |z|),$$

- **Repetición $l \in \mathbb{N}^3$ veces en los ejes x, y, z con separación $s \in \mathbb{R}$:**

$$\mathrm{sdf}_{rep}(p) = \mathrm{sdf}_A\left(p - s \cdot c\left(r\left(\frac{p}{s}\right), -l, l\right)\right),$$

- **Repetición infinita:**

$$\mathrm{sdf}_{repInf}(p) = \mathrm{sdf}_A\left(\left(p + \frac{l}{2} \mod l\right) - \frac{l}{2}\right),$$

donde

- $c\colon \mathbb{R} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, $c(x, a, b) = \mathrm{mín}(\mathrm{máx}(x, a), b)$ acota x en $[a, b]$,
- $r\colon \mathbb{R}^3 \to \mathbb{R}^3$ redondea las componentes de un vector a sus enteros más cercanos.

Observación 1.3. Hay casos en los que los SDF definidos en la Def. 1.8 podrían no ser exactos, al igual que ocurría con la intersección y la diferencia en la Def. 1.4:

- Para las simetrías, cuando el objeto interseca el plano de simetría,
- Para las repeticiones, cuando las dimensiones del objeto sean mayores o iguales a $l/2$.

A la vista de todas esta operaciones a nuestra disposición, es evidente el potencial que tienen los SDF, tanto en la generación y modificación de figuras como en su eficiencia, ya que podemos visualizar miles de objetos al precio de uno utilizando las simetrías o repeticiones.

En el Apéndice A se muestra el resultado de aplicar las operaciones vistas a diferentes primitivas.

1.2. Renderizado

Una vez definida la escena a partir de un SDF, necesitamos una forma para visualizarla, para lo que utilizaremos la API de OpenGL [4] y aplicaremos la técnica de *raymarcing*.

1.2.1. Creación del lienzo

Si bien se puede hacer *raymarching* directamente sobre una escena 3D, nuestra escena constará únicamente de un plano formado por cuatro vértices y dos triángulos, que usaremos como lienzo (o *canvas*) para dibujar sobre él. Para ello, necesitaremos trabajar sobre diferentes espacios de coordenadas que nos proporciona OpenGL, los cuales pasamos a enumerar.

- **Coordenadas locales o de objeto:** distancias relativas al origen del objeto,
- **Coordenadas globales o de mundo:** distancias relativas a un origen común para todos los objetos,
- **Coordenadas de cámara:** distancias relativas a un sistema de referencia posicionado y alineado con la cámara,
- **Coordenadas de recortado:** distancias normalizadas en el rango $[-1, 1]^2$ relativas a un sistema asociado al rectángulo que forma la imagen en pantalla.

Para crear el lienzo, debemos declarar sus vértices y cómo estos se unen formando triángulos. Si hacemos uso de `GL_TRIANGLES` bastará con definir los vértices en sentido antihorario, pero hay que tener en cuenta que tendremos que repetir dos vértices, ya que se irán formando los triángulos en grupos de tres vértices. Una alternativa para no repetir vértices sería utilizar tablas de vértices e índices, pero en nuestro caso no merece la pena al tener únicamente seis vértices. Un ejemplo de definición de vértices formando un lienzo rectangular podría ser el que se muestra en la Figura 1.8.

Este lienzo, como toda geometría, tendrá asignado dos *shaders* o procesadores, programas escritos en GLSL (lenguaje parecido a C) y que se ejecutan en la GPU. Estos programas son independientes entre sí, y la única forma en la que pueden comunicarse entre ellos es mediante el paso de atributos de entrada y salida con las palabras clave `in` y `out` respectivamente. Hay dos tipos de *shaders*: de vértices (*vertex shader*) y de fragmento o píxel (*fragment shader*), cada uno con atributos específicos de entrada y salida.

En el *vertex shader* utlizaremos

```
glBegin(GL_TRIANGLES);
    glColor3f(1.0f, 1.0f, 1.0f);

    // Triangulo inferior
    glVertex3f(-2.0f, -1.0f, 0.0f);
    glVertex3f(-2.0f, 1.0f, 0.0f);
    glVertex3f(2.0f, 1.0f, 0.0f);

    // Triangulo superior
    glVertex3f(-2.0f, -1.0f, 0.0f);
    glVertex3f(2.0f, 1.0f, 0.0f);
    glVertex3f(2.0f, -1.0f, 0.0f);
glEnd();
```

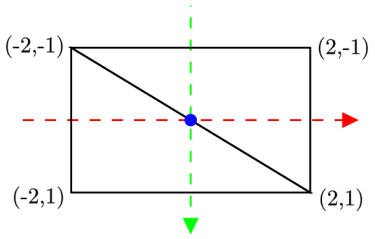

Figura 1.8.: Construcción del lienzo

- **`in vec4 gl_Vertex`**: contiene las coordenadas locales del vértice actual y es pasado autómaticamente por la aplicación,
- **`out vec4 gl_Position`**: posición transformada del vértice actual. La cuarta componente es la componente homogénea, que es necesaria para realizar el cambio a coordenadas recortadas.

En el *fragment shader* utilizaremos

- **`in vec4 gl_FragCoord`**: coordenadas de dispositivo para el píxel actual en el *fragment shader*. Al ser un atributo de entrada del *fragment shader*, está interpolada en cada vértice. La cuarta componente es la inversa de la componente homogénea de `gl_Position`, y se utiliza en el cálculo de la profundidad de los píxeles y en las operaciones de corrección de perspectiva,
- **`out vec4 gl_FragColor`**: terna RGBA para asignar el color del píxel actual en el *fragment shader*.

Por último, en caso de que queramos pasar nuestros propios atributos desde otro programa, deberemos hacerlo a través de un `uniform`.

En primer lugar se ejecuta el **procesador de vértices o *vertex shader*** para cada vértice de la geometría. Su objetivo principal es el de realizar transformaciones de coordenadas, y adicionalmente pasar atributos al *fragment shader*. Dada la posición del vértice actual, que se nos proporciona a través del atributo `gl_Vertex`, para cambiar de un sistema de coordenadas a otro se utilizan matrices de transformación [5] [6]. En particular, usaremos las que siguen.

- **Matriz de modelo M:** define la posición, orientación y escala del objeto en la escena. Se utiliza para pasar del coordenadas locales a coordenadas de mundo. En nuestro caso, si creamos el plano centrado en el origen, podemos simplemente tomar

$$M = Id_4,$$

- **Matriz de vista V:** define la posición y orientación de cada punto respecto a la cámara de la escena. Se utiliza para pasar de coordenadas de mundo a coordenadas de vista.

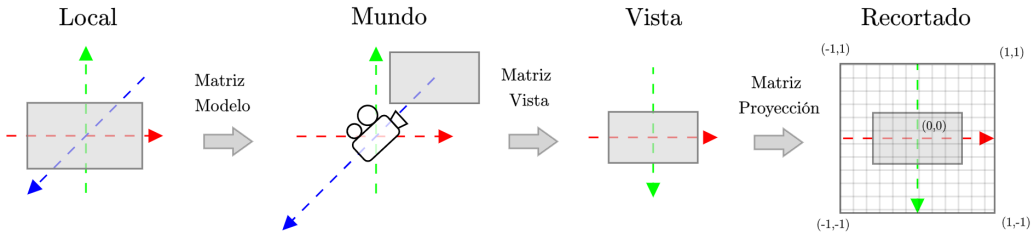

Figura 1.9.: Coordenadas locales a recortadas

Lo que ocurre en realidad es que la cámara está fija en el origen, y es el resto de la escena es la que se mueve respecto a ella. Por tanto, esta matriz contiene la posición y orientación inversa de la cámara. En nuestro caso, si queremos desplazar la cámara una unidad en el eje Z, la matriz de vista tendrá la forma

$$V = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

- **Matriz de proyección:** define cómo la escena se proyecta en la pantalla, incluyendo el campo de visión, aspecto y planos cercano y lejano. Se utiliza para pasar de coordenadas de vista a coordenadas recortadas. OpenGL nos proporciona una función para calcularla:

```
glm::mat4 projectionMatrix = glm::perspective(
    glm::radians(FoV), // campo vertical de vision
    4.0f / 3.0f,       // aspecto
    0.1f,              // plano de corte cercano
    100.0f             // plano de corte lejano
);
```

Con esto, ya podemos escribir nuestro *vertex shader*:

Procesador de vértices

```
#version 330
uniform mat4 projection;
uniform mat4 view;
uniform mat4 model;

void main()
{
    gl_Position = projection * view * model * gl_Vertex;
}
```

Tras realizar estas transformaciones, las coordenadas de recortado se transforman a coordenadas de dispositivo, que están centradas en la esquina inferior izquierda de la pantalla y

toman valor en el rango $[0, r_x] \times [0, r_y]$, donde $r = (r_x, r_y)$ es la resolución de la pantalla.

Ahora le toca el turno al **procesador de fragmentos o *fragment shader***. Este se ejecuta para cada píxel de la pantalla, y su objetivo es asignar a la variable `gl_FragColor` el color que el píxel tendrá como una terna RGBA. Será aquí donde hagamos todos los cálculos necesarios pare renderizar la superficie con *raymarching*. Para ello, necesitaremos un sistema de coordenadas dentro del propio lienzo, que generaremos haciendo uso de `gl_FragCoord` y la resolución del lienzo, atributo que pasaremos nosotros al *shader* a través de un `uniform`, que llamaremos `u_resolution`.

Para obtener estas coordenadas, primero desplazamos el origen que nos proporciona `gl_FragCoord` al centro de la pantalla, y posteriormente normalizamos respecto a alguno de los ejes. Hacemos esto porque si intentamos normalizar sobre ambos ejes, obtendremos coordenadas en el rango $[-0.5, 0.5]^2$, lo que hará que en un lienzo que no sea cuadrado, la imagen se vea estirada en la dirección del eje más largo. Nosotros normalizaremos respecto al eje vertical, ya que en nuestro caso será siempre el menor. Esto nos dará como resultado unas coordenadas con valores en $[-0.5 \cdot aspect, 0.5 \cdot aspect] \times [-0.5, 0.5]$, donde $aspect$ es el ratio de aspecto del lienzo:

$$uv = \frac{gl\_FragCoord.xy - 0.5 \cdot u\_resolution.xy}{u\_resolution.y},$$

Hemos denotado a las coordenadas obtenidas como `uv`, haciendo referencia a la similitud que tienen con el uso que se le da a las coordenadas de textura habituales. Podemos ver la diferencia entre ambos sistemas de coordenadas si usamos `uv` como los canales rojo y verde de `gl_FragColor`, tal y como se muestra en la Figura 1.14.

Ya tenemos nuestro *fragment shader* preparado para aplicar el algoritmo de *raymarching*:

Procesador de fragmentos

```
#version 330

void main()
{
    vec2 uv=(gl_FragCoord.xy-.5*u_resolution.xy)/u_resolution.y;
    vec3 color = raymarching(uv);
    gl_FragColor = vec4(color,1.0);
}
```

1.2.2. Raymarching y Spheretracing

A partir de ahora, pensamos en que nuestra escena no es la de OpenGL, sino aquella que queremos dibujar usando *raymarching* dado un SDF ϕ. Podemos pensar en esta escena como $\mathbb{R}^3$ con su base usual $B_u = \{e_1, e_2, e_3\} = \{(1,0,0), (0,1,0), (0,0,1)\}$, donde colocamos los siguientes elementos.

- **Plano de visión:** rejilla en el plano XY y centrada en el origen, donde cada uno de sus cuadrados corresponde a un píxel del lienzo,
- **Punto de la cámara c_o:** punto del espacio desde donde se observa la escena,

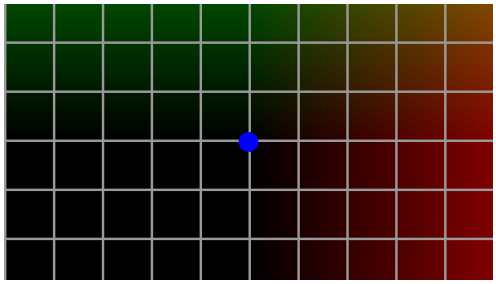

Figura 1.10.: Eje X

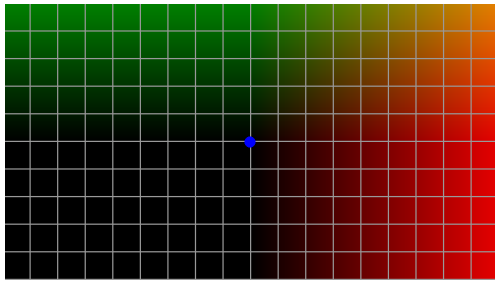

Figura 1.11.: Eje Y

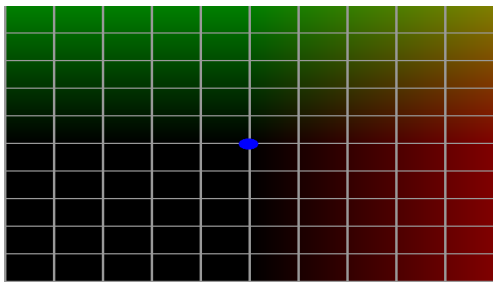

Figura 1.12.: Ejes X e Y

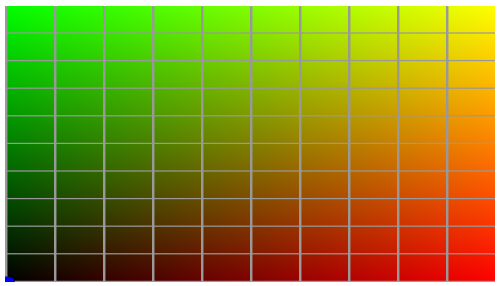

Figura 1.13.: Ninguno

Figura 1.14.: Normalización de coordenadas sobre distintos ejes

- **Punto de atención o *lookat point l*:** hacia que punto del espacio debe mirar la cámara. Tomaremos siempre $l = (0, 0, 0)$.

El método del *raymarching* consiste en trazar rayos a partir de c_o hacia el centro de cada uno de los cuadrados del plano de visión, de forma que si el rayo interseca con S_ϕ significa que ese píxel corresponde a un punto de la superficie, y será coloreado como tal.

Cada uno de estos rayos estará definido por un origen r_o y una dirección r_d. El origen será siempre la posición de la cámara c_o, pero la dirección requiere más trabajo. En el escenario descrito en la Figura 1.15, dado que en todo momento conocemos las coordenadas de cada punto de la rejilla a través de $uv = (u, v)$, es claro que podemos tomar $r_d = (u, v, -c_o)$. Aquí c_o actúa como un control del campo de visión, de forma que cuanto menor sea su valor, menor se verán los objetos dibujados. Lo fijaremos a un valor de 1. Sin embargo este escenario es el más sencillo posible, y si queremos poder mover la cámara tendremos que poder trabajar con una orientación arbitraria suya. Para ello deberemos construir un marco cartesiano relativo a ella, esto es, una base $B = \{f_1, f_2, f_3\}$ de $\mathbb{R}^3$ alineada con la cámara. Esta base deberá ser ortonormal y tener la misma orientación que la base usual.
Obtenemos primero vectores que nos resultarán útiles para generar esta base.

- **Vector director c_d:** indica la dirección hacia la que mirará la cámara. luego vendrá dado por $c_d = l - c_o$,
- ***Right vector* c_r** : es el análogo a e_x en la base usual, luego lo podemos obtener como $c_r = (0, 1, 0) \times c_d$,

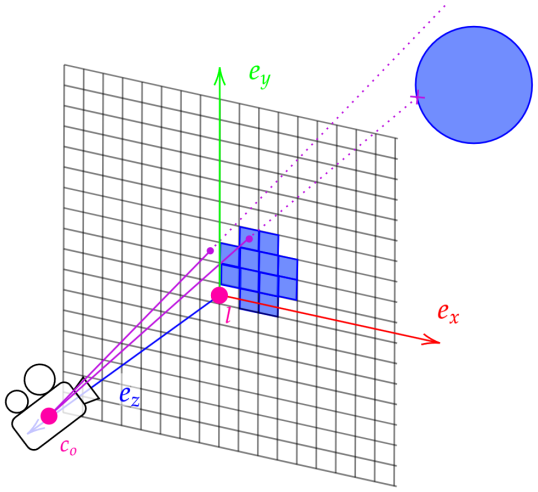

Figura 1.15.: Trazado de rayos a través del plano de visión

- ***Up vector*** c_u: dirección en la que el observador ve proyectada en vertical la escena. Podemos obtenerlo como $c_u = c_d \times c_r$.

A partir de estos vectores podemos obtener $\{f_1, f_2, f_3\}$ normalizándolos y teniendo en cuenta que el plano de visión y la cámara estarán orientados de forma opuesta:

$$f_1 = -\frac{c_r}{\|c_r\|} = -\frac{(0,1,0) \times c_d}{\|l - c_o\|}, \quad f_2 = -\frac{c_d}{\|c_d\|} = -\frac{l - c_o}{\|l - c_o\|}, \quad f_3 = \frac{c_u}{\|c_u\|} = f_2 \times f_1.$$

Ahora solo queda transformar el vector director $(u, v, -1)$ a la base que acabamos de obtener. La matriz de cambio de base serán las coordenadas por columnas de $\{f_1, f_2, f_3\}$ escritas en función de $\{e_1, e_2, e_3\}$. Al ser esta la base usual, obtenemos que la matriz de cambio de base consiste en escribir por columnas $\{f_1, f_2, f_3\}$, de forma que

$$rayo = (u, v, -1)_B = (-c_r|c_u| - c_d) \cdot \begin{pmatrix} u \\ v \\ -1 \end{pmatrix}$$

Ahora que ya tenemos toda la información del rayo, falta comprobar si este interseca con S_ϕ. Para esto se utiliza un método iterativo: a partir de c_o, en cada iteración avanzamos en la dirección del rayo una distancia fija δ. Evaluamos entonces nuestro SDF en la posición actual, de forma que si obtenemos un valor muy cercano a 0 significará que hemos llegado a la isosuperficie. De lo contrario, repetimos el proceso hasta encontrar una intersección o superar un número máximo de iteraciones, en cuyo caso concluiremos que no hay intersección. La Figura 1.18 ilustra este procedimiento, donde `DibujarSuperficie()` y `DibujarFondo()` devuelven ternas RGBA que serán asignadas al píxel actual dependiendo de si hay intersección o no.

Una desventaja de esta técnica es que puede ser bastante lenta, ya que cuanto más alejados estén los puntos de S_ϕ del observador, mayor es el número de iteraciones necesarias para encontrar la intersección en caso de que la haya. En el peor de los casos en el que tal intersección no exista, se habrá realizado el número máximo de iteraciones, que deberá ser bastante

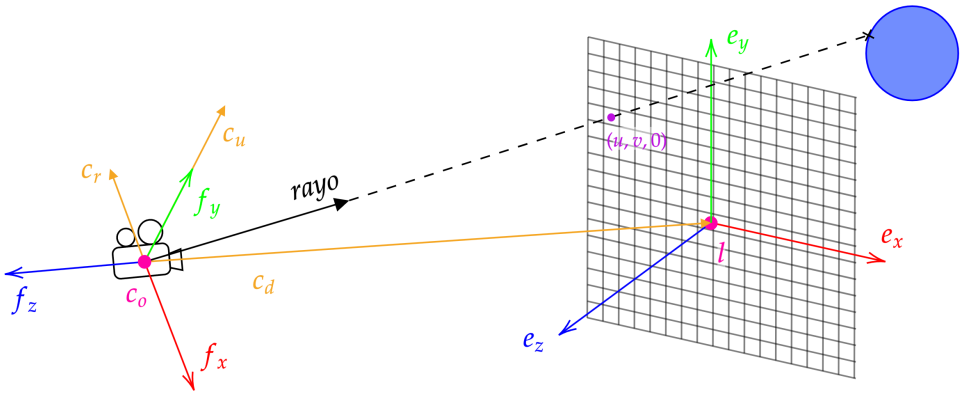

Figura 1.16.: Obtención de la dirección del rayo

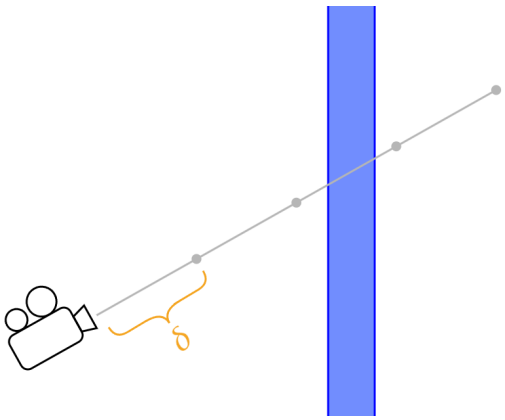

Figura 1.17.: Pérdida de intersección en *raymarching* para valores elevados de δ

alto, pues si no queremos perder ninguna intersección como ocurre en la Figura 1.17, el valor de incremento δ tenddrá que ser pequeño.

La solución a este problema es el uso de *spheretracing*, que reduce drásticamente el número de iteraciones y por tanto evaluaciones de ϕ, necesarias para detectar la intersección. Su funcionamiento es similar al *raymarching*, con la diferencia de que el incremento en la posición del rayo no es fija, sino que es la máxima que podemos tomar en cada momento asegurándonos de no perder información. Esta distancia será la mínima del punto actual del rayo a S_ϕ, que no es más que evaluar ϕ en dicho punto.

El algoritmo que usaremos será por tanto el descrito en la Figura 1.19.

1.2.3. Iluminación

Ya sabemos qué píxeles pertenecen a la superficie, pero no de qué color deben dibujarse. Para ello utilizaremos el modelo de reflexión de Phong. En él trabajaremos con los siguientes

Algorithm 1: Raymarching

Data: origen p, dirección v
$d \leftarrow 0$ `// distancia total`
for $i \in MAX\_ITERACIONES$ **do**
  rayo $\leftarrow p + d \cdot v$
  sdf $\leftarrow \phi$(rayo)
  if $sdf < \varepsilon$ **then**
    return *DibujarSuperficie(rayo)*
  end
  $d \leftarrow d + \delta$;
  if $d > MAX\_DISTANCIA$ **then**
    return *DibujarFondo()*
  end
end

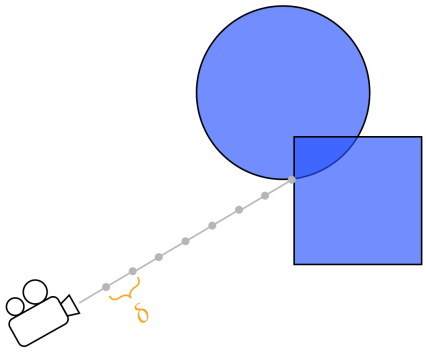

Figura 1.18.: Algoritmo de *raymarching*

Algorithm 2: Spheretracing

Data: origen p, dirección v
$d \leftarrow 0$ `// distancia actual`
for $i \in MAX\_ITERACIONES$ **do**
  rayo $\leftarrow p + d \cdot v$
  sdf $\leftarrow \phi$(rayo)
  if $sdf < \varepsilon$ **then**
    return *DibujarSuperficie(rayo)*;
  end
  $d \leftarrow$ d + sdf
  if $d > MAX\_DISTANCIA$ **then**
    return *DibujarFondo()*;
  end
end

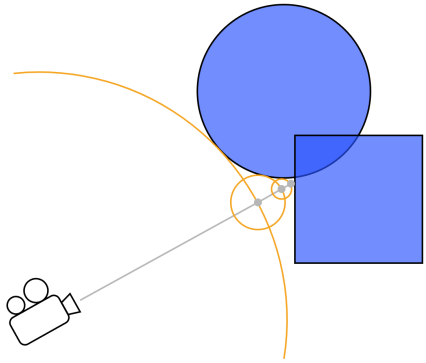

Figura 1.19.: Algoritmo de *spheretracing*

parámetros:

- Luz ambiente i_a: vector RGB suma de las contribuciones de todas las fuentes de luz de la escena. Contiene un valor por defecto que será el color del fondo.
- Lista de fuentes de luz $(l_1, \ldots, l_m, \ldots, l_n)$.
- Materiales asignados a cada objeto definidos por las constantes:
  - Reflexión especular k_s: modela cómo se refleja la luz en objetos brillantes. Depende de tanto de la posición de las fuentes de luz como de la del observador.
  - Reflexión difusa k_d: modela cómo se refleja la luz en objetos mates. Depende de la posición de las fuentes de luz, pero no de la posición del observador.
  - Reflexión ambiental k_a: representa la proporción de luz del ambiente que refleja el objeto.

  - Coeficiente de brillo α: valor real positivo que permite variar el tamaño de las zonas brillantes (a mayor valor, menor tamaño y más pulida o especular).
- Los vectores normalizados definidos para cada punto de la superficie p:
  - L_m: vector director desde p a cada fuente de luz l_m.
  - N vector normal a la superficie en p.
  - R_m dirección del rayo de luz reflejado especularmente desde la fuente l_m en el punto p.
  - V: dirección de p a la posición del observador.

TODO: diagrama vectores
Con estas variables, el color en p vendrá dado por la fórmula:

$$I_p = k_a i_a + \sum_{m=0}^{n} \left(k_d \left(L_m \cdot N \right) + k_s \left(R_m \cdot V \right)^{\alpha} \right)$$

De los vectores anteriores, L_m y V se calculan trivialmente, y R_m se puede obtener como

$$R_m = 2(L_m \cdot N)N - L_m$$

Veamos como obtener N.

1.2.3.1. Cálculo del vector normal

Definición 1.9. El **gradiente** de una función implícita $\phi\colon \mathbb{R}^3 \to \mathbb{R}$ es $\nabla\phi = \left(\frac{\partial\phi}{\partial x}, \frac{\partial\phi}{\partial y}, \frac{\partial\phi}{\partial z}\right)$.

Proposición 1.1. *$\nabla\phi$ es perpendicular a S_ϕ.*

Demostración. Sea $s \in S_\phi$ arbitrario. Tomamos una curva parametrizada:

$$\begin{aligned} \alpha\colon [0,1] &\to S_\phi \\ t &\mapsto (x(t), y(t), z(t)) \end{aligned}$$

cumpliendo $\alpha(t_0) = s$. Veamos que $\nabla\phi(s) \perp \alpha$:

$$\begin{aligned} \alpha(t) \subset S_\phi &\implies \phi(\alpha(t)) = 0 \\ &\implies \nabla\phi(\alpha(t)) = \frac{\partial F}{\partial x}\frac{dx}{dt} + \frac{\partial F}{\partial y}\frac{dy}{dt} + \frac{\partial F}{\partial z}\frac{dz}{dt} = 0 \\ &\implies \langle \nabla\phi(x,y,z), \alpha'(t) \rangle = 0 \implies \langle \nabla\phi(x,y,z), \alpha'(t_0) \rangle = 0 \end{aligned}$$

Por tanto, $\nabla\phi(s)$ es perpendicular al vector tangente de α en s, que a su vez está contenida en el plano tangente de S_ϕ en s. Por tanto $\nabla\phi(s) \perp S_\phi$. □

Hemos visto que calcular N equivale a calcular $\nabla\phi$. Sin embargo, dado que la superficie viene dada por un SDF que podría ser no derivable, no podemos hacerlo de forma analítica. De ser derivable, podría ser una buena opción calcular el gradiente una única vez al momento de definir la ϕ, tras lo cual para obtener N solo habría que realizar evaluaciones de dicho gradiente. Sin embargo, por su sencillez y popularidad, nos decantaremos por un método numérico que únicamente hará uso de evaluaciones de ϕ.

Definición 1.10. Dada $\phi : \mathbb{R}^3 \to \mathbb{R}$ diferenciable, $p = (x, y, z)$, $v \in \mathbb{R}^3$, definimos la **derivada direccional** en p con dirección v a:

$$\nabla_v f(p) = \nabla f(p) \cdot v = \frac{\partial f(p)}{\partial x} v_x + \frac{\partial f(p)}{\partial y} v_y + \frac{\partial f(p)}{\partial z} v_z$$

Para el cálculo de cada parcial de f podemos utilizar la definición de derivada. Por ejemplo, para la primera componente:

$$\frac{\partial f(p)}{\partial x} v_x = \lim_{h \to 0} \frac{f(p + (h, 0, 0)) - f(p)}{h} v_x$$

Procedemos a calcular N aplicaremos el **método del tetraedro**, el cual se basa en evaluar ϕ en la dirección de los vértices de un tetraedro:

$$k_0 = (1, -1, -1) \quad k_1 = (-1, -1, 1) \quad k_2 = (-1, 1, -1) \quad k_3 = (1, 1, 1)$$

Proposición 1.2 (Método del tetraedro)**.** *Dado $p \in S_\phi$, su vector normal N se obtiene normalizando el vector*

$$\hat{N} = \sum_{i=0}^{3} k_i \cdot f(p + h k_i)$$

Demostración. Por la proposición Proposición 1.1, basta comprobar que $\hat{N}$ es colineal a $\nabla\phi(p)$.

$$\hat{N} = \sum_{i=0}^{3} k_i \cdot f(p + h k_i) = \sum_{i=0}^{3} k_i \cdot f(p + h k_i) + k_i \cdot f(p) = \sum_{i=0}^{3} k_i \cdot f(p + h k_i - f(p)) = \sum_{i=0}^{3} k_i \nabla_{k_i} f(x)$$
$$= \sum_{i=0}^{3} k_i \cdot (k_i \cdot \nabla f(p)) = \sum_{i=0}^{3} (k_i \cdot k_i) \nabla f(p) = \sum_{i=0}^{3} \nabla f(p) = 4 \nabla f(p)$$

donde hemos usado que $\sum_{i=0}^{3} k_i = (0, 0, 0)$, $\sum_{i=0}^{3} k_i \cdot k_i = (1, 1, 1)$, y que el producto escalar es un operador lineal. □

1.3. Aproximación de implícitas

Empezábamos el capítulo diciendo que una de las representaciones más comunes de superficies es a través de implícitas, pero hasta ahora nos hemos centrado en estudiar un tipo particular de este tipo de ecuaciones. Si intentásemos aplicar los algoritmos anteriores a una función implícita cualquiera, observaríamos que el resultado tiene ciertos defectos, tales como deformaciones, o incluso no se visualiza. Vamos a introducir una técnica que dada una función implícita ϕ, nos permitirá obtener un SDF aproximado de S_ϕ. Esto nos será útil cuando no conozcamos o no podamos calcular explícitamente el SDF de una superficie pero sí su representación implícita.

Proposición 1.3. *Sea ϕ: $\mathbb{R}^3 \to \mathbb{R}$ una función implícita cualquiera y $p \in \mathbb{R}^3$. Entonces:*

$$sdf_{S_\phi}(p) \approx \frac{|\phi(p)|}{|\nabla\phi(p)|}$$

Demostración. Sea q al punto de S_ϕ más cercano a p y $v = \vec{pq} \perp S_\phi$ en q. Queremos calcular la distancia de p a S_ϕ, que es justamente v. Podemos calcular el desarrollo de Taylor de ϕ en p:

$$\phi(p+v) = \phi(p) + \nabla(p) \cdot (p+v-p) + \mathcal{O}(|p+v-p|^2) \implies \phi(p+v) = \phi(p) + \nabla(p) \cdot v + \mathcal{O}(|v|^2)$$

Suponemos ahora que p está cerca de S_ϕ, esto es, $|v| < \varepsilon$. Como $\phi(p+v) = \phi(q) = 0$, tenemos que:

$$\begin{aligned} 0 = |\phi(p+v)| &\approx |\phi(p) + \nabla(p) \cdot v| \leq |\phi(p)| - |\nabla(p) \cdot v| \\ &\leq |\phi(p)| - |\nabla(p)||v| \implies |v| \leq \frac{|\phi(p)|}{|\nabla(p)|} \end{aligned}$$

donde hemos usado la desigualdad triangular y las propiedades del producto escalar. □

Al igual que nos ocurría al calcular el vector normal, habrá ocasiones en las que el gradiente no pueda ser obtenido de forma analítica. Esta vez sin embargo no nos basta con conocer solo la dirección del gradiente, así que no podemos aplicar de nuevo el método del tetraedro. Usaremos un nuevo método en su lugar que usa la definición de derivada.

Proposición 1.4 (Método de las diferencias centrales)**.** *Sea $\phi\colon \mathbb{R}^3 \to \mathbb{R}$. Entonces:*

$$\frac{\partial \phi}{\partial x}(p) \approx \frac{\phi(p + (h,0,0)) - \phi(p - (h,0,0))}{2h}$$

$$\frac{\partial \phi}{\partial y}(p) \approx \frac{\phi(p + (0,h,0)) - \phi(p - (0,h,0))}{2h}$$

$$\frac{\partial \phi}{\partial z}(p) \approx \frac{\phi(p + (0,0,h)) - \phi(p - (0,0,h))}{2h}$$

donde $h \in \mathbb{R}^+$ es un valor arbitrariamente pequeño.

Para calcular el gradiente necesitaremos evaluar ϕ un total de seis veces, ya que hay que realizar dos evaluaciones por cada variable. Es por este motivo que

2. Implicitación

Ahora que sabemos representar las superficies generadas por una función implícita cualquiera, nos proponemos ser capaces de representar también superficies definidas paramétricamente. Para ello haremos uso de la teoría de bases de Groebner, pero antes debemos introducir una serie de definiciones y resultados previos que nos pongan en contexto.

2.1. Polinomios en varias variables

Fijamos A cuerpo y $X = \{x_1, \dots, x_n\}$ un conjunto de variables distintas.

Definición 2.1. Llamamos **monomio** en X al producto de la forma:

$$x_1^{\alpha_1} \cdots x_n^{\alpha_n} \quad , \ \alpha_i \in \mathbb{N}, \ \forall 0 \leq i \leq n$$

Lo denotaremos como X^α, y diremos que $\alpha \in \mathbb{N}^n$ es el **exponente** del monomio.

Definición 2.2. Definimos un **polinomio** en X con coeficientes en A a toda combinación lineal finita de monomios:

$$\sum_{\alpha \in \mathbb{N}^n} a_\alpha X^\alpha, \ a_\alpha \in A$$

Proposición 2.1. *El conjunto de polinomios, que denotaremos $A[X] = A[x_1, \dots, x_n]$, es un cuerpo con las operaciones:*

- *Suma heredada de A:* $(f+g)(\alpha) = f(\alpha) + f(\alpha), \ \forall f, g \in A[X]$.
- *Producto de convolución:* $(fg)(\alpha) = \sum_\alpha \sum_{\beta+\gamma=\alpha} f(\beta)g(\gamma), \ \forall f, g \in A[X]$.

Nos surge la pregunta de si hay alguna forma "natural" de ordenar los monomios que forman un polinomio, al igual que ocurre para polinomios de una sola variable. Primero tenemos que especificar qué es lo que consideramos "natural".

Definición 2.3. Un **orden admisible** es un orden total $\leq$ sobre $\mathbb{N}^n$ cumpliendo:

1. $(0, \dots, 0) \leq \alpha, \ \forall \alpha \in \mathbb{N}^n$.
2. $\alpha < \beta \implies \alpha + \gamma < \beta + \gamma, \ \forall \alpha, \beta, \gamma \in \mathbb{N}^n$.

Siempre supondremos que usamos un orden admisible, luego podemos ordenar los monomios que conforman un polinomio ordenando sus exponentes según cualquier dicho orden. Hay multitud de órdenes entre los que elegir. Algunos de los más importantes son:

Ejemplo 2.1. Definimos el **orden lexicográfico** $\leq_{\text{lex}}$ como:

$$\alpha \leq_{\text{lex}} \beta \iff \begin{cases} \alpha = \beta \\ \quad \text{ó} \\ \alpha_i < \beta_i, \text{ donde } i \text{ es el primer índice tal que } \alpha_i \neq \beta_i \end{cases}$$

Definición 2.4. Llamamos **grado** de $\alpha \in \mathbb{N}^n$ a $|\alpha| = \sum_{i=1}^{n} \alpha_1 + \cdots + \alpha_n$.

Ejemplo 2.2. Definimos el **orden lexicográfico graduado inverso** $\leq_{\text{degrevlex}}$ como:

$$\alpha \leq_{\text{degrevlex}} \beta \iff \begin{cases} |\alpha| < |\beta| \\ \quad \text{ó} \\ |\alpha| = |\beta| \text{ y } \alpha_i > \beta_i, \text{ donde } i \text{ es el último índice tal que } \alpha_i \neq \beta_i \end{cases}$$

Una vez obtenida la noción de orden admisible, estamos en disposición de definir varios conceptos que nos resultarán imprescindibles.

Definición 2.5. Sea $f = \sum_{\alpha \in \mathbb{N}^n} a_\alpha X^\alpha$ y $\leq$ admisible. Definimos:

- **Exponente:** $exp(f) = \text{máx}_{\leq} \alpha$.
- **Monomio líder:** $\text{lm}(f) = X^{\exp(f)}$
- **Coeficiente líder:** $\text{lc}(f) = a_{\exp(f)}$.
- **Término líder:** $\text{lt}(f) = \text{lc}(f) \cdot \text{lm}(f)$.

Teorema 2.1. *Sea $F = \{f_1, \ldots, f_s\} \subset A[X]$. Entonces todo polinomio $f \in A[X]$ se puede expresar como:*

$$f = q_1 f_1 + \cdots + q_s f_s + r$$

donde $q_i, r \in A[X]$ y r no se puede expresar como combinación lineal de ningún elemento de F o es 0. Notaremos $rem(f, [F]) = r$.

En otras palabras, podemos dividir f entre los polinomios $f_1, \ldots, f_s$ para expresar f como combinación lineal de elementos de F. Sin embargo, esta descomposición no será única, y dependerá del orden que ocupen los polinomios en F. Podemos calcular esta descomposición usando Algoritmo 3.

Demostración. TODO □

2.2. Bases de Groebner

Definición 2.6. Dado un anillo conmutativo R, decimos que $\emptyset \neq I \subseteq R$ es un **ideal** si:

1. $a + b \in I, \ \forall a, b \in I$.
2. $ab \in I, \ \forall a \in I, \ \forall b \in R$.

Lo denotaremos como $I \leq R$.

Definición 2.7. Dado $F \subseteq A$, el **ideal generado** por F es:

$$\langle F \rangle = \{a_1 f_1 + \cdots + a_s f_s : a_1, \ldots, a_s \in A, \ f_1, \ldots, f_s \in F\}$$

Definición 2.8. Dado $I \leq A[X]$, diremos que $G = \{g_1, \ldots, g_s\} \subseteq I$ es una **base de Groebner** para I si $\langle \text{lt}(I) \rangle = \langle \text{lt}(g_1), \ldots, \text{lt}(g_t) \rangle$.

Algorithm 3: División polinomios varias variables

Data: f, $F = [f_1, \ldots, f_S]$
$p \leftarrow f$;
$[q_1, \ldots, q_s] \leftarrow [0, \ldots, 0]$;
$r \leftarrow 0$;
while $p \neq 0$ **do**
  divisorEncontrado $\leftarrow$ *false*;
  for $f_i \in F$ **do**
    if $exp(p) = exp(f_i) + \alpha$ **then**
      $q_i \leftarrow q_i + \frac{\mathrm{lc}(p)}{\mathrm{lc}(f_i)} X^{\alpha}$;
      $p \leftarrow p - f_i \cdot \frac{\mathrm{lc}(p)}{\mathrm{lc}(f_i)} X^{\alpha}$;
      divisorEncontrado $\leftarrow$ *true*;
    end
  end
  if !*divisorEncontrado* **then**
    $r \leftarrow r + \mathrm{lt}(p)$;
    $p \leftarrow p - \mathrm{lt}(p)$;
  end
end
return $[r, q_1, \ldots, q_s]$

Definición 2.9. Dada $f \in A[X]$, definimos su **soporte** como

$$\mathrm{supp}(f) = \{\alpha \in \mathbb{N}^n : f(\alpha) \neq 0\}$$

Definición 2.10. G se dirá una **base de Groebner reducida** para I si para todo $g \in G$ se cumple:

- $\mathrm{lc}(g) = 1$.
- $\mathrm{supp}(g) \cap (\exp(G \setminus \{g\}) + \mathbb{N}^n) = \emptyset$.

La base de Groebner es a un ideal lo que un sistema de generadores a un espacio vectorial: un subconjunto a partir del cual podemos obtener el total. Si además la base es reducida, esta será el equivalente a la base del espacio vectorial. Por tanto, resulta de gran interés saber si dado un ideal I siempre existirá una base de Groebner asociada, y de ser así, si podemos calcularla.

Definición 2.11. Dados $\alpha, \beta \in \mathbb{N}^n$, definimos:

- **mínimo común múltiplo**: $\mathrm{lcm}(\alpha, \beta) = \{\mathrm{máx}(\alpha_1, \beta_1), \ldots, \mathrm{máx}(\alpha_n, \beta_n)\}$.
- **máximo común divisor**: $\gcd(\alpha, \beta) = \{\mathrm{mín}(\alpha_1, \beta_1), \ldots, \mathrm{máx}(\alpha_n, \beta_n)\}$.

Definición 2.12. Sean $f, g \in A[X]$ donde $\alpha = \exp(f)$, $\beta = \exp(g)$, $\gamma = \mathrm{lcm}(\alpha, \beta)$. Definimos el **S-polinomio** de f y g como

$$S(f, g) = \mathrm{lc}(g) X^{\gamma - \alpha} f - \mathrm{lc}(f) X^{\gamma - \beta} g$$

Teorema 2.2 (Primer Criterio de Buchberger). *Sean $I \leq A[X]$ y $G = \{g_1, \ldots, g_t\}$ conjunto de generadores de I. Entonces:*

$$G \text{ es base de Groebner para } I \iff rem(S(g_i, g_j), G) = 0,\ \forall 1 \leq i < j \leq t$$

El algoritmo que usaremos para el cálculo de la base de Groebner se basará en este criterio. Sin embargo, antes de presentarlo, estudiamos dos criterios adicionales que lo harán más eficiente evitando el cálculo de S-polinomios innecesarios, que es la operación más costosa del algoritmo al suponer la realización de numerosas divisiones.

Teorema 2.3 (Criterios de Buchberger). *Sean $I \leq A[X]$ y $g_1, g_2 \in G \subseteq A[X]$. Si se cumple cualquiera de las siguientes condiciones:*

1. $lcm(g_1, g_2) = lm(g_1)lm(g_2)$.
2. $\exists f \in G$: $lm(f) | lcm(g_1, g_2)$ *y se verifica alguna de las condiciones:*
   a) *algún* $S(g_i, f) \xrightarrow{G} 0$.
   b) $lm(f) | \frac{lm(g_i}{gcd(g_1, g_2}$ *y* $sm(g_j)lm(f) \neq sm(f)lm(g_j)$.

   donde $i, j \in \{1, 2\}$ *e* $i \neq j$.

Entonces $S(g_1, g_2) \xrightarrow{G} 0$

Aplicando estos criterios, obtenemos Algoritmo 4.

Algorithm 4: Algoritmo de Buchberger optimizado

```
Data: f, F = [f_1, ..., f_S]
G ← F;
repeat
    G' ← G;
    for each pair {f, g} ⊆ G' do
        if !Criterio 1(f, g) AND !Criterio 2(f, g, G') then
            r ← rem(S(f, g), G');
            if r ≠ 0 then
                G ← G ∪ {r};
            end
        end
    end
until G' = G;
return G
```

Observación 2.1. Una decisión necesaria y que aún no hemos mencionado es la de cómo se eligen las parejas $\{f, g\}$. Uno de los métodos más usados es la conocida como **estrategia normal**, debido a su simpleza y haber probado ser de las que completan más rápido el algoritmo. Esta consiste en tomar el par f, g cuyo $\text{lcm}(f, g)$ sea del menor grado posible según el orden admisible usado.

Ahora ya somos capaces de calcular una base de Groebner de todo ideal $I \leq A[X]$. No obstante, nos gustaría que siempre que fuera posible, esta fuera reducida. Veamos que siempre existe esta base reducida y cómo obtenerla.

Definición 2.13. Sea $M \leq \mathbb{N}^n$. Decimos que A es un **conjunto generador minimal** de M si:

$$M = a + \mathbb{N}^n \quad \wedge \quad M \neq (A \setminus \{\alpha\}) + \mathbb{N}^n,\ \forall \alpha \in A$$

Lema 2.1. *Todo ideal tiene un único conjunto generador minimal.*

Teorema 2.4. *Todo ideal I admite una única base de Groebner reducida para un orden admisible dado.*

Demostración. Existencia Sea G un conjunto generador minimal de $\exp(I)$, que sabemos que existe por Lema 2.1. Sea $g \in G$ y $r = \text{rem}\,(g, [G \setminus \{g\}])$. Tenemos que:

$$\exp(g) \notin \exp(G \setminus \{g\} + \mathbb{N}^n \implies \exp(g) = \exp(r) \implies \exp(G) = \exp\left((G \setminus \{g\}) \cup \{r\}\right)$$

Como $g - r \in \langle G \setminus \{g\} \rangle \subseteq I$, tenemos que $r \in I$. Por tanto $G' = (G \setminus \{g\} \cup \{r\}$ es base de Groebner de I y cumple $\text{supp}(r) \cap (\exp(G' \setminus \{r\}) + \mathbb{N}^n) = \emptyset$. Aplicando este procedimiento a cada $g \in G$ obtenemos una base reducida de I.

Unicidad Sean G_1, G_2 dos bases minimales de I. Por Lema 2.1:

$$\exp(G_1) = \exp(G_2) \implies \forall g_1 \in G_1,\ \exists! g_2 \in G_2 : \exp(g_1) = \exp(g_2)$$

Como $g_2 - g_1 \in I \implies \text{rem}(g1 - g_2, G_1) = 0$. Por otro lado:

$$\left.\begin{array}{l} \text{supp}(g1 - g_2) \subseteq (\text{supp}(g_1) \cup \text{supp}(g_2)) \setminus \{\exp(g_1)\} \\ \text{supp}(g_i) \setminus \{\exp(g_i)\} \cap (\{\exp(G_i) + \mathbb{N}^n\}) = \varnothing,\ i \in \{1,2\} \end{array}\right\} \implies \text{supp}(g_1 - g_2) \cap (\exp(G_1) + \mathbb{N}^n) = \varnothing$$

$$\implies \text{rem}(g_1 - g_2, G_1) = g_1 - g_2 \implies g_1 = g_2 \implies G_1 = G_2$$

□

De esta demostración podemos obtener Algoritmo 5 para reducir una base de Groebner cualquiera a su respectiva minimal:

Algorithm 5: Minimización de base de Groebner

```
Data: G base a minimizar
G ← F;
foreach g ∈ G do
    g ← g/lc(g);
    r ← rem(g, [G \ {g}]);
    if r ≠ 0 then
        g ← r;
    end
end
```

2.3. Implicitación Polinomial

Definición 2.14. Dado $F \subseteq A[X]$, llamamos **variedad afín** definida por F al conjunto:

$$\mathbb{V}(F) = \{(a_1, \ldots, a_n) \in A^n : f_i(a_1, \ldots, a_n) = 0,\ \forall 1 \leq i \leq s\}$$

Definición 2.15. Dado $I \leq A[x_1, \ldots, x_n]$, diremos que su **ideal de l-eliminación** es:

$$I_l = I \cap A[x_{l+1}, \ldots, x_n] \leq A[x_{l+1}, \ldots, x_n]$$

Definición 2.16. Decimos que un orden admisible $\leq$ es un **orden de l-eliminación** si

$$\beta \leq \alpha \implies \beta \in \mathbb{N}_l^n,\ \forall \alpha \in \mathbb{N}_l^n,\ \forall \beta \in \mathbb{N}^n$$

donde $\mathbb{N}_l^n = \{\alpha \in \mathbb{N}^n : \alpha_i = 0,\ 1 \leq i \leq l\}$.

Teorema 2.5 (Eliminación)**.** *Sea $I \leq A[x_1, \ldots, x_n]$ y G una base de Groebner suya respecto a un orden $\leq$ de l-eliminación. Entonces, una base de Groebner para I_l viene dada por:*

$$G_l = G \cap A[x_{l+1}, \ldots, x_n]$$

Teorema 2.6 (Implicitación Polinomial)**.** *Dados $f_1, \ldots, f_n \in A[t_1, \ldots, t_r]$ con A cuerpo infinito, sea*

$$\begin{aligned} \phi \colon A^r &\to A^n \\ (a_1, \ldots, a_r) &\mapsto (f_1(a_1, \ldots, a_r), \ldots, f_n(a_1, \ldots, a_r)) \end{aligned}$$

Definimos los ideales:

- $I = \langle x_1 - f_1, \ldots, x_n - f_n \rangle \leq A[t_1, \ldots, t_r, x_1 \ldots, x_n]$
- $J = I \cap A[x_1, \ldots, x_n]$ *el ideal de r-eliminación de I*

Entonces, $\mathbb{V}(J)$ es la menor variedad que contiene a $\phi(A^r)$.

3. Desarrollo de la aplicación

Nos centramos ahora en los aspectos de la implementación de una aplicación que haga uso de todas las técnicas y conceptos tratados en los capítulos anteriores. Para ello, se ha usado React, una biblioteca de JavaScript (y TypeScript) para interfaces de usuario. Las principales ventajas de este enfoque son:

- La aplicación puede ser ejecutada en cualquier navegador, haciendo que sea mucho más accesible.
- React está basada en componentes modulares, lo que la hace escalable. Además. debido a su popularidad, hay una infinidad de librerías de terceros a nuestra disposición, ya sea específicas de React o de JavaScript.

Un aspecto fundamental a lo largo de todo el desarrollo será el del rendimiento. Esto es debido a que, por defecto, las aplicaciones web solo tienen a su disposición una hebra de ejecución (la de interfaz de usuario), haciendo de cuello de botella para el resto de cálculos.

3.1. Librería de polinomios multivariable

Si bien tenemos a nuestra disposición un gran número de librerías externas, en el momento de realización de la aplicación no encontré ninguna alternativa viable para trabajar con polinomios multivariable en JavaScript de forma nativa. Como alternativas se barajó el uso de la API de Geogebra o realizar llamadas a código Python que usara SageMath. Sin embargo, por motivos de rendimiento y completitud, se decidió desarrollar una librería nativa en TypeScript para el manejo de polinomios en varias variables y cálculo de bases de Groebner. Se encuentra disponible en GitHub junto a su documentación, ejemplos de uso y tests usados.

La librería consta de tres clases que pasamos a estudiar a continuación.

3.1.1. Clase `Monomial`

3.1.2. Clase `Polynomial`

3.1.3. Clase `Ideal`

A. Resultados de operaciones sobre SDF

TODO

Bibliografía

[1] B. Foundation, "Blender - 3d creation software - git repository," Último acceso: 6 mayo 2023.

[2] M. Molecule, "Dreams," 2020.

[3] I. Quílez, "Smooth minimun," [En línea]. Último acceso: 6 mayo 2023.

[4] O. Foundation, "Opengl - api for rendering 2d and 3d graphics."

[5] OpenGL, "Coordinate systems." https://learnopengl.com/Getting-started/Coordinate-Systems. Accessed: 2023-5-9.

[6] OpenGL, "Matrices." http://www.opengl-tutorial.org/es/beginners-tutorials/tutorial-3-matrices/#matrices-modelo-vista-y-proyecci%C3%B3n. Accessed: 2023-5-10.