



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

Abstract	v
Introducción	VII
Estado del arte	IX
1. Fundamentos matemáticos: las SDFs	1
1.1. Diferenciabilidad	2
1.2. Funciones cota de distancia	4
1.3. Operaciones sobre SDF	5
1.3.1. Operaciones booleanas	5
1.3.2. Operaciones afines	11
1.3.3. Operaciones deformantes	11
1.3.4. Operaciones de repetición	12
1.4. Obtención a partir de ecuación implícita	13
1.5. Obtención a partir de ecuaciones paramétricas	14
1.5.1. Polinomios en varias variables	14
1.5.2. Bases de Groebner	17
1.5.3. Implicitación Polinomial	20
2. Algoritmos de visualización de SDFs	25
2.1. Renderizado por <i>spheretracing</i>	25
2.1.1. Creación del lienzo	25
2.1.2. Raymarching y spheretracing	29
2.2. Modelos de iluminación y sombras	33
2.2.1. Modelos de Blinn y Blinn-Phong	33
2.2.2. Sombras	40
2.2.3. Oclusión ambiental	44
2.3. <i>Antialiasing</i>	47
3. Desarrollo e implementación	53
3.1. Editor de nodos	54
3.2. Panel de primitivas	56
3.3. Gestor de estado	56
3.4. Librería de polinomios multivariante	56
3.4.1. Clase Monomial	56
3.4.2. Clase Polynomial	56
3.4.3. Clase Ideal	56
4. Pruebas y rendimiento	57
5. Conclusiones y líneas futuras	59

Índice general

A. Resultados de operaciones sobre SDF	61
B. Resultado de técnicas empleadas al enderezar SDFs	63
Bibliografía	65

Abstract

Introducción

En el campo de los gráficos generados por computador los métodos más asentados para describir objetos tridimensionales han sido mediante polígonos o funciones implícitas. Sin embargo, en los últimos años ha surgido un enfoque alternativo que muestra un gran potencial: la utilización de funciones distancia con signo (SDF). Estos permiten representar objetos geométricos mediante una única función escalar que asigna a cada punto del espacio su distancia con signo respecto a la superficie del objeto. Veremos que esta representación es especialmente útil para la generación de imágenes y el renderizado en tiempo real, ofreciendo ventajas significativas en términos de eficiencia y precisión respecto otros métodos tradicionales.

El objetivo principal de este Trabajo de Fin de Grado (TFG) es la creación de una aplicación web interactiva que facilite la creación e interacción con superficies generadas a través de SDF de forma intuitiva para el usuario sin que este necesite conocimientos al respecto. Para lograr este objetivo, llevaremos a cabo el desarrollo de la aplicación aprovechando las tecnologías web y *frameworks* modernos, evaluando la eficiencia y el rendimiento de la aplicación en términos de tiempos de respuesta y calidad visual.

Empezaremos este trabajo presentando los SDF explorando los fundamentos teóricos y matemáticos detrás de los SDF y comprendiendo su capacidad para describir superficies y volúmenes de manera precisa y compacta, así como las técnicas empleados para su manipulación, como las operaciones booleanas o las deformaciones. Posteriormente estudiaremos cómo pueden ser representados mediante *spheretracing* y aplicando técnicas avanzadas como *antialiasing* y oclusión ambiental .

Una vez comprendidos los detalles de los SDF nos centraremos en estudiar como aprovechar sus ventajas para superficies dadas de forma implícita o paramétrica obteniendo una representación suya como SDF usando bases de Groebner

Estado del arte

1. Fundamentos matemáticos: las SDFs

Estamos acostumbrados a representar superficies en \mathbb{R}^3 a través de ecuaciones paramétricas e implícitas. En el caso de las paramétricas se asigna a cada tupla de parámetros un punto, mientras que para las implícitas, dado un punto la ecuación indica si este se encuentra dentro o fuera de la superficie. El tipo de superficies implícitas más comúnmente estudiado es el de las superficies algebraicas, variedades algebraicas de dimensión 2. Existen varios métodos para la visualización de este tipo de superficies. Uno de ellos es tratar de generar una malla de polígonos previamente a partir de la ecuación para después ser visualizada en tiempo real usando los métodos clásicos. El problema de este método es que no siempre se puede aplicar y conlleva pérdida de precisión en la representación de la superficie. Otro método es el *raytracing*, pero este también puede llegar a perder precisión, además de que es muy lento, haciendo que la representación de una superficie tan simple como una esfera sea computacionalmente muy costoso.

Como solución a esto, T. Sederberg y A. Zundel [1] presentaron en 1989 un método capaz de representar superficies algebraicas sin pérdida de información y de manera eficiente, capaz además de trabajar con siluetas, intersección de curvas y operaciones booleanas. En 1989 John C. Hart [2] presenta una técnica de *raymarching* para la representación de fractales usando funciones distancia con signo. Posteriormente, en 1995 [3] generaliza esta técnica con el uso de *spheretracing* para la representación de cualquier superficie implícita (algebraica o no), punto de partida de este trabajo.

Este método nos permitirá representar cualquier superficie en tiempo real con un coste computacionalmente muy bajo y a cualquier nivel de detalle. No obstante, para usarlo debemos comprender qué son las funciones distancia con signo, sus propiedades, y como trabajar con ellas.

Definición 1.1. Sea $\Omega \subset \mathbb{R}^3$. La **función distancia** asociada a Ω , que llamamos d_Ω es el campo escalar que a cada punto de \mathbb{R}^3 le asigna su menor distancia a la frontera de Ω :

$$d_\Omega: \mathbb{R}^3 \rightarrow \mathbb{R}_0^+, \\ x \mapsto \inf\{\|x - y\| : y \in \delta\Omega\}.$$

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

Definición 1.2. Sea $\Omega \subset \mathbb{R}^3$. La **función distancia con signo** asociada a Ω es el campo escalar de la forma:

$$\phi_\Omega: \mathbb{R}^3 \rightarrow \mathbb{R}, \\ x \mapsto \begin{cases} d_\Omega(x), & x \in \mathbb{R}^3 \setminus \overset{\circ}{\Omega}, \\ -d_\Omega(x), & x \in \overset{\circ}{\Omega}. \end{cases}$$

En general, nos referiremos a esta función por sus siglas en inglés SDF (*Signed Distance*

1. Fundamentos matemáticos: las SDFs

Function), y la denotaremos simplemente ϕ siempre que no haya confusión.

Observación 1.1. Un campo escalar $f: \mathbb{R}^3 \rightarrow \mathbb{R}$ cualquiera será una función distancia si existe al menos un $\Omega \subset \mathbb{R}^3$ tal que $f = d_\Omega$. De la misma forma, f será una SDF cuando para dicho Ω se tenga $f = \phi_\Omega$.

Definición 1.3. Dada una función $\phi: \mathbb{R}^3 \rightarrow \mathbb{R}$ 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 sobre el eje Y de radios R y r , con $R > r$:**

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

1.1. Diferenciabilidad

Antes de seguir avanzando vamos a realizar un estudio de la diferenciabilidad de los SDF, pues nos será de utilidad en las siguientes secciones. Empezamos recordando varios conceptos de análisis diferencial [4] fijadas las variables $\{x_1, x_2, x_3\}$ y la base usual

$$B = \{e_1, e_2, e_3\} = \{(1,0,0), (0,1,0), (0,0,1)\}.$$

Cuando sea conveniente identificaremos

$$x_1 = x, \quad x_2 = y, \quad x_3 = z.$$

Definición 1.4. Sea U un abierto de \mathbb{R}^3 y $\phi: U \rightarrow \mathbb{R}$. Definimos la *i*-ésima derivada parcial de ϕ en $p_0 \in \mathbb{R}^3$ como

$$\frac{\partial \phi}{\partial x_i}(p_0) = \lim_{h \rightarrow 0} \frac{\phi(p_0 + he_i) - \phi(p_0)}{h}, \quad i = 1, 2, 3.$$

Definición 1.5. Llamamos **gradiente** de $\phi: \mathbb{R}^3 \rightarrow \mathbb{R}$ a la función

$$\begin{aligned}\nabla\phi: \mathbb{R}^3 &\rightarrow \mathbb{R}^3 \\ p &\mapsto \left(\frac{\partial\phi}{\partial x_1}(p), \frac{\partial\phi}{\partial x_2}(p), \frac{\partial\phi}{\partial x_3}(p) \right).\end{aligned}$$

Definición 1.6. Dada $\phi: \mathbb{R}^3 \rightarrow \mathbb{R}$ diferenciable, definimos la **derivada direccional** en $p_0 \in \mathbb{R}^3$ en la dirección $v \in \mathbb{R}^3$ a:

$$\nabla_v\phi(p_0) = \nabla\phi(p_0) \cdot v = \frac{\partial\phi(p_0)}{\partial x}v_x + \frac{\partial\phi(p_0)}{\partial y}v_y + \frac{\partial\phi(p_0)}{\partial z}v_z$$

Definición 1.7. Sea U un abierto de \mathbb{R}^3 y $\phi: U \rightarrow \mathbb{R}$. Diremos que ϕ es **diferenciable** en $p_0 \in U$ si existen todas sus derivadas parciales en p_0 y son continuas.

Ahora mismo, dado un SDF arbitrario no tenemos información alguna sobre su diferenciabilidad, ya que su expresión puede ser de lo más variada y compleja. Veamos una propiedad que cumplen todos los SDF y que nos permitirá obtener algo de información al respecto [5] [6].

Definición 1.8. Una campo escalar $\phi: \mathbb{R}^3 \rightarrow \mathbb{R}$ se dice **lipschitziano** si existe una constante $L > 0$ tal que

$$|\phi(p) - \phi(q)| \leq L\|x - y\|, \forall p, q \in \mathbb{R}^3.$$

El menor valor posible para la constante L recibe el nombre de **constante de Lipschitz**.

Proposición 1.1. Sea $\phi: \mathbb{R}^3 \rightarrow \mathbb{R}$ una función lipschitziana cualquiera con constante de Lipschitz L . Entonces

$$|\phi'(p)| \leq L$$

en todo punto donde sea derivable.

Lema 1.1. Sea $\phi: \mathbb{R}^3 \rightarrow \mathbb{R}$ un SDF. Entonces ϕ es lipschitziana de constante $L = 1$.

Demostración. Sean p y $q \in \mathbb{R}^3$. Usando la Def. 1.1, para todo $s \in S_\phi$ se tiene

$$\begin{aligned}\phi(p) &\leq \|p - s\| = \|p - q + q + s\| \leq \|p - q\| + \|q - s\| \\ \implies \phi(p) - \|p - q\| &\leq \|q - s\| \implies \phi(p) - \|p - q\| \leq \inf_{s \in S_\phi} (\|q - s\|) = \phi(q) \\ \implies \phi(p) - \phi(q) &\leq \|p - q\|\end{aligned}$$

De forma análoga podemos ver que $\phi(q) - \phi(p) \leq \|q - p\|$, luego

$$|\phi(p) - \phi(q)| \leq 1 \cdot \|p - q\|.$$

□

Lema 1.2 (Teorema de Rademacher). *Sea U un abierto de \mathbb{R}^3 y $\phi: U \rightarrow \mathbb{R}$ lipschitziana. Entonces ϕ es diferenciable en casi todo punto de U .*

Tenemos por tanto asegurado que ϕ será diferenciable en casi todo punto de \mathbb{R}^3 . No obstante, podemos concretar aún más dónde están los puntos de conflicto cuando Ω sea lo

1. Fundamentos matemáticos: las SDFs

suficientemente regular [7] [8]. Para ello necesitaremos introducir el concepto de esqueleto de una superficie [6].

Definición 1.9. Sea $\phi_\Omega: \mathbb{R}^3 \rightarrow \mathbb{R}$ un SDF. Llamamos **esqueleto** de Ω al conjunto de puntos de \mathbb{R}^3 cuya distancia a la superficie puede obtenerse como la distancia a dos o más puntos distintos de $\delta\Omega$:

$$\epsilon(\Omega) = \{p \in \mathbb{R}^3 : \phi(p) = \|p - q\| = \|p - r\|, q, r \in \delta\Omega, q \neq r\}.$$

Definición 1.10. Sea $\Omega \subset \mathbb{R}^3$ y $p \in \Omega$. Llamamos **vector normal** en p al vector perpendicular al borde de Ω en p . Lo denotamos N_p , y siempre supondremos que está normalizado.

Teorema 1.1. Sea $\Omega \subseteq \mathbb{R}^3$ un conjunto abierto acotado con frontera diferenciable y $\phi_\Omega: \mathbb{R}^3 \rightarrow \mathbb{R}$ un SDF. Entonces ϕ_Ω es diferenciable en un entorno tubular U de $\delta\Omega$. Es más, para cada $p \in \mathbb{R}^3$ se cumple una de las siguientes propiedades.

1. $p \in \delta\Omega$ y ϕ_Ω es diferenciable en p con $\nabla S_{\phi(p)} = N_p$,
2. $p \notin \delta\Omega$ y ϕ_Ω es diferenciable en $p \iff p \in \mathbb{R}^3 \setminus \epsilon(\Omega)$, en cuyo caso

$$\nabla \phi_\Omega(p) = \frac{q - p}{\phi_\Omega(p)},$$

donde q es el único punto de $\delta\Omega$ tal que $\phi_\Omega(p) = \|q - p\|$.

Corolario 1.1. Todo SDF $\phi: \mathbb{R}^3 \rightarrow \mathbb{R}$ satisface la ecuación de la eikonal

$$\|\nabla \phi(p)\| = 1$$

en todo punto p donde sea diferenciable.

1.2. Funciones cota de distancia

Hemos visto que las SDFs nos proporcionan la distancia signada exacta en cada punto al más cercano de un conjunto $\Omega \subset \mathbb{R}^3$, pero lo cierto es que para representar la isosuperficie generada por un SDF no se necesita tanta precisión, ya que mientras dos funciones tengan los mismos ceros generarán la misma isosuperficie. Sin embargo, para representar la superficie en las siguientes secciones usaremos el método del *spheretracing* (Subsección 2.1.2), que sí utiliza la información de la distancia en puntos diferentes a la frontera de Ω . Introducimos un concepto que nos permitirá seguir usando este método pero no es tan restrictivo como el de SDF [3].

Definición 1.11. Sea $\Omega \subset \mathbb{R}^3$ y ϕ su SDF. Una **cota de la distancia con signo** asociada a Ω es un campo escalar $\gamma: \mathbb{R}^3 \rightarrow \mathbb{R}$ cumpliendo

$$|\gamma(p)| \leq |\phi(p)|, \forall p \in \mathbb{R}^3.$$

y que tiene el mismo signo que ϕ en cada punto. Nos referiremos a estas funciones como SDB por sus siglas en inglés (Signed Distance Bound).

Observamos que una SDF es un caso especial de SDB en el que se cumple la igualdad. De esta definición es evidente que una SDF y una SDB asociadas a un mismo Ω tienen los

mismos ceros, y generan por tanto la misma isosuperficie. Aunque trabajar con una SDB en *spheretracing* hará que la convergencia sea más lenta, en ocasiones una SDB es más rápida de evaluar que una SDF por tener una expresión más simple, haciendo que sea deseable trabajar con ellas. Habrá otras ocasiones en las que incluso será imposible obtener una SDF para un cierto Ω . Por estos motivos en la literatura no se suele distinguir entre SDF y SDB, y como mucho se utilizan los términos SDF exacta y aproximada cuando se quiere realizar una distinción.

Veamos algunos ejemplos de SDF aproximadas.

Ejemplo 1.2. Para los siguientes conjuntos Ω podemos definir las SDFs aproximadas:

- **Cono centrado en el origen a lo largo del eje Y con altura h y ángulo θ .**

$$\Omega = \{p \in \mathbb{R}^3 : (x^2 + z^2)(\cos \theta)^2 - y^2(\sin \theta)^2\},$$

$$\phi(p) = \max\left((\sin \theta, \cos \theta) \cdot (\|(x, 0, z)\|, y), -h - y\right).$$

- **Elipsoide.**

$$\Omega = \{p \in \mathbb{R}^3 : \frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 0, a, b, c \in \mathbb{R} \setminus \{0\}\},$$

$$\phi(p) = \left\| \left(\frac{x}{a}, \frac{y}{b}, \frac{z}{c} \right) \right\| \cdot \left(\left\| \left(\frac{x}{a}, \frac{y}{b}, \frac{z}{c} \right) \right\| - 1 \right) / \left\| \left(\frac{x}{a^2}, \frac{y}{b^2}, \frac{z}{c^2} \right) \right\|.$$

1.3. 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. Como comentamos en la introducción, una de las principales ventajas del uso de ecuaciones implícitas para representar modelos geométricos es la facilidad de combinación de estas primitivas, por ejemplo mediante operaciones booleanas o deformaciones. Sin embargo los sistemas que hacían uso de esta técnica no estaban lo suficientemente estructurados como para permitir aplicar estas operaciones de manera general e intuitiva, haciendo que no se pudieran aplicar de forma local y por tanto no se pudieran generar escenas complejas.

Esto cambió en 1999 con el artículo de B.Wyvill y otros [9], en el que sugieren usar una estructura de árbol para definir modelos como combinación de otros a través de operaciones básicas. La gran ventaja de este método es que es muy extensible, y además permite ver de forma muy clara la estructura del modelo. En esta sección estudiaremos los principales tipos de estas operaciones. En el [Apéndice A](#) podemos verlas en acción sobre diferentes primitivas.

1.3.1. Operaciones booleanas

Una de las técnicas más útiles para generar nuevas formas a partir de primitivas es la geometría de sólidos constructiva. Por la naturaleza de los SDF, estas operaciones se implementan fácilmente usando las funciones \max y \min .

1. Fundamentos matemáticos: las SDFs

Definición 1.12 (Operaciones Booleanas). Sean A y B isosuperficies generadas por ϕ y γ respectivamente. Definimos los SDF para las siguientes operaciones.

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

Observación 1.2. Solo en el caso de la unión se obtiene un SDF exacto, 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 [10] o videojuegos como Dreams [11], y que ha sido estudiada por Íñigo Quílez en su web [12].

Observación 1.3. Para mayor claridad del razonamiento, en las figuras se representarán funciones de variable real, a pesar de que nosotros trabajamos en \mathbb{R}^3 .

Explicaremos la técnica poniendo como ejemplo la unión, y al final veremos como la intersección y la diferencia se deducen fácilmente de esta. La idea es, dadas ϕ y γ , añadir una corrección para cada punto a la función \min original para que cumpla ciertos requisitos. Por comodidad, definiremos

$$\begin{aligned}\text{mín}_{\phi,\gamma}: \mathbb{R}^3 &\rightarrow \mathbb{R}, \\ p &\mapsto \min(\phi(p), \gamma(p)),\end{aligned}$$

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

Llamaremos a la mencionada corrección $\omega_k: \mathbb{R}^3 \rightarrow \mathbb{R}$, donde $k \in \mathbb{R}_0^+$ es un coeficiente que controlará la intensidad del suavizado. Por tanto, la versión suavizada de la función \min original será

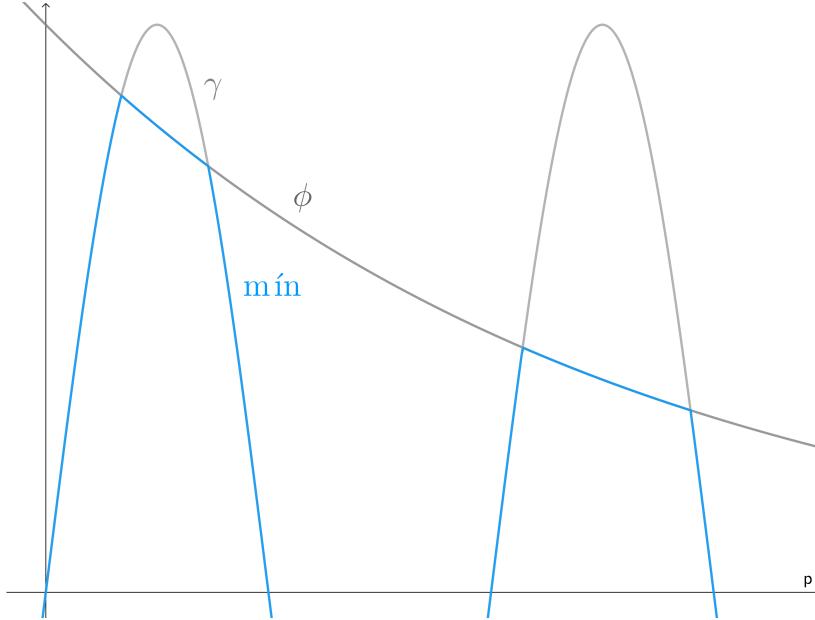
$$\begin{aligned}\text{smín}_{\phi,\gamma}: \mathbb{R}^3 &\rightarrow \mathbb{R}, \\ p &\mapsto \min_{\phi,\gamma}(p) - \omega_k(p).\end{aligned}$$

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 $\min(p) \geq \text{smín}(p)$, esto es,

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

Si estudiamos como se comporta la versión real de \min 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

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

original, luego estudiaremos el comportamiento de smín en el conjunto de entornos de las intersecciones. Usaremos el valor de k para decidir el tamaño de estos entornos, aplicando la corrección únicamente en los puntos del conjunto

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

de forma que $\omega(p) = 0$ cuando $p \notin B_k$.

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

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

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

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

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_k :

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

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

1. Fundamentos matemáticos: las SDFs

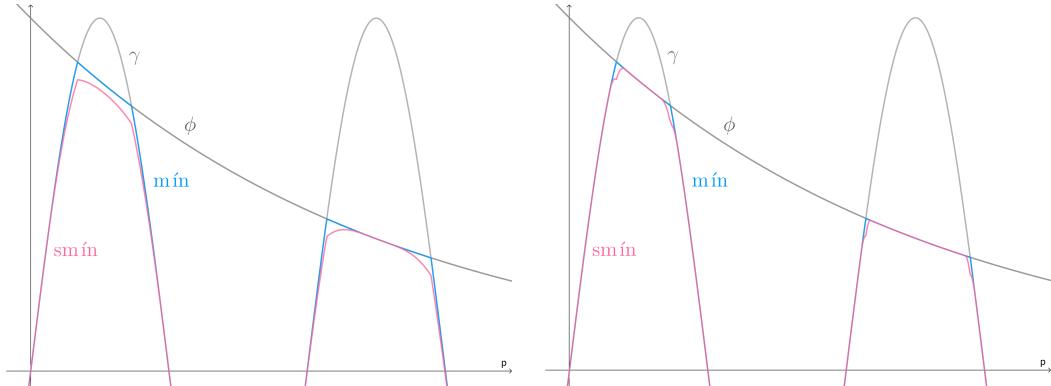


Figura 1.2.: $k = 0.6$

Figura 1.3.: $k = 0.1$

Figura 1.4.: Primera aproximación de $\text{smín}(p)$ con $s = 0.05$ y $n = 2$

Nuestro objetivo es que smín tenga un aspecto natural y varíe de forma suave. Comprobemos propiedades que debería cumplir smín para ser C^1 en cada entorno de B_k . Que es continua es evidente:

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

Otra condición necesaria es que sus derivadas parciales sean continuas. Estas son de la forma

$$\frac{\partial \text{smín}}{\partial x_i}(p) = \begin{cases} \frac{\partial \gamma}{\partial x_i}(p) + sn \left(1 - \frac{\phi(p) - \gamma(p)}{k} \right)^{n-1} \left(\frac{\frac{\partial \phi}{\partial p}(p) - \frac{\partial \gamma}{\partial p}(p)}{k} \right), & \phi(p) > \gamma(p), \\ \frac{\partial \phi}{\partial x_i}(p) + sn \left(1 - \frac{\phi(p) - \gamma(p)}{k} \right)^{n-1} \left(\frac{\frac{\partial \phi}{\partial p}(p) - \frac{\partial \gamma}{\partial p}(p)}{k} \right), & \phi(p) \leq \gamma(p) \end{cases}, \quad i = 1, 2, 3.$$

Por tanto, para que se cumpla la condición imponemos

$$\begin{aligned} \frac{\partial \phi}{\partial x_i} - sn \left(1 + \frac{\phi - \gamma}{k} \right)^{n-1} \left(\frac{\frac{\partial \phi}{\partial x_i} - \frac{\partial \gamma}{\partial x_i}}{k} \right) &= \frac{\partial \gamma}{\partial x_i} + sn \left(1 - \frac{\phi - \gamma}{k} \right)^{n-1} \left(\frac{\frac{\partial \phi}{\partial x_i} - \frac{\partial \gamma}{\partial x_i}}{k} \right) \\ \cancel{\frac{\partial \phi}{\partial x_i}} - \cancel{\frac{\partial \gamma}{\partial x_i}} &= 2sn \left(1 - \frac{\phi - \gamma}{k} \right)^{n-1} \left(\frac{\cancel{\frac{\partial \phi}{\partial x_i}} + \cancel{\frac{\partial \gamma}{\partial x_i}}}{k} \right) \\ s &= \frac{k}{2n} \left(1 - \frac{\phi - \gamma}{k} \right) \end{aligned}$$

Evaluando en $c \in I$:

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

Hemos llegado a la expresión final

$$\begin{aligned} \omega_k(p) &= \begin{cases} \frac{k}{2n} \left(1 - \frac{|\phi(p) - \gamma(p)|}{k} \right)^n, & |\phi(p) - \gamma(p)| \leq k, \\ 0, & \text{otro caso.} \end{cases} \\ &= \frac{\max(k - |\phi(p) - \gamma(p)|, 0)^n}{2n \cdot k^{n-1}}, \quad s \in \mathbb{R}, n \in \mathbb{N}. \end{aligned}$$

Podemos observar los resultados en la [Figura 1.7](#). Finalmente, para obtener una versión suavizada del máximo, es fácil comprobar que

$$\begin{aligned} \text{smáx}_{\phi, \gamma} &: \mathbb{R}^3 \rightarrow \mathbb{R}, \\ p &\mapsto -\text{smín}_{-\phi, -\gamma}(p). \end{aligned}$$

Recogemos los resultados obtenidos a continuación.

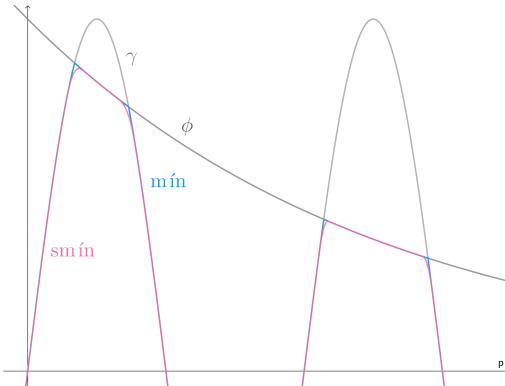


Figura 1.5.: $k = 0.1, n = 2$

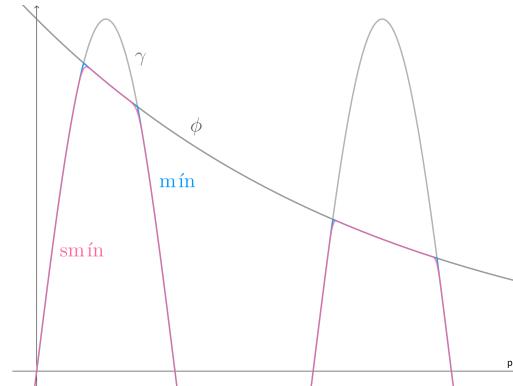


Figura 1.6.: $k = 0.1, n = 3$

Figura 1.7.: Resultado final de $\text{smin}(p)$

Definición 1.13 (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:** $\text{sdf}_{\text{unionS}}(p) = \text{mín}(\phi(p), \gamma(p)) - \frac{\max(k - |\phi(p) - \gamma(p)|, 0)^n}{2n \cdot k^{n-1}},$
- **Intersección suavizada:** $\text{sdf}_{\text{interS}}(p) = -\text{mín}(-\phi(p), -\gamma(p)) + \frac{\max(k - |\phi(p) - \gamma(p)|, 0)^n}{2n \cdot k^{n-1}},$
- **Diferencia suavizada:** $\text{sdf}_{\text{difS}}(p) = -\text{mín}(-\phi(p), \gamma(p)) + \frac{\max(k - |\phi(p) + \gamma(p)|, 0)^n}{2n \cdot k^{n-1}},$

1. Fundamentos matemáticos: las SDFs

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

Observamos que los operadores definidos en la Def. 1.12 no son más que un caso particular de estos últimos cuando $k \rightarrow 0$. Este método se puede generalizar para obtener

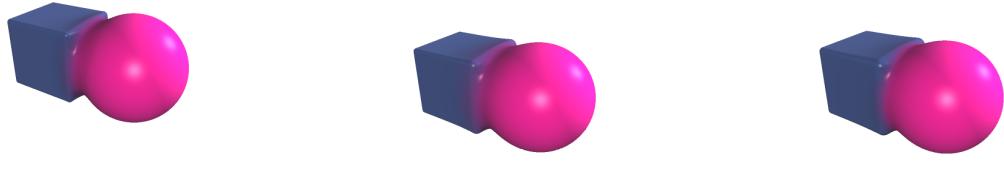
Este método para obtener una versión suavizada de las funciones mín y máx no es la única. Hemos elegido esta debido a que su deducción es bastante natural, el efecto que tiene el valor k sobre el resultado final es intuitivo y, sobretodo, porque es eficiente. En el artículo que hemos mencionado al inicio de la sección, Íñigo Quílez [12] presenta otras tres alternativas a esta versión, a la cual él se refiere como “mínimo suavizado polinomial”, y que también son compatibles con *raymarching*. Si denotamos ahora $\text{smín}(\phi(p), \gamma(p)) = \text{smín}_{\phi, \gamma}$, estas versiones son:

- **Mínimo suavizado exponencial:** $\text{smín}(a, b) = \frac{-\log_2(2^{-ka} + 2^{-kb})}{k}$,
- **Mínimo suavizado potencial:** $\text{smín}(a, b) = \left(\frac{a^k \cdot b^k}{a^k + b^k} \right)^{1/k}$,
- **Mínimo suavizado por raíz:** $\text{smín}(a, b) = \frac{a+b-\sqrt{(a-b)^2+k}}{2}$.

La principal ventaja de la versión polinomial respecto a estas es que es la más rápida al ser sus cálculos computacionalmente más baratos. Por otro lado tanto la exponencial como la potencial permiten ser adaptadas fácilmente para calcular el mínimo de un conjunto arbitrario de puntos, útil cuando se trabaja con patrones de voronoi o nubes de puntos. Además, la versión exponencial produce siempre el mismo resultado independientemente del orden en el que se aplique. Es decir,

$$\text{smín}(a, \text{smín}(b, c)) = \text{smín}(b, \text{smín}(a, c)).$$

En la Figura 1.8 podemos ver un ejemplo de uso de estas versiones, en las que además se ha usado el valor de w_k de la ecuación Figura 1.3.1 para interpolar la componente difusa de ambas primitivas usando el método *mix* de GLSL. Como vemos, no hay diferencias notables entre las distintas versiones, así que nos quedaremos con el método más eficiente: el polinómico.



(a) Polinomial,
 $k = 1.5, n = 2$

(b) Exponencial, $k = 2.5$

(c) Raíz, $k = 1$

Figura 1.8.: Diferentes versiones de la unión suavizada

1.3.2. Operaciones afines

Pasamos ahora a estudiar otro tipo de operaciones que nos permitirán aplicar movimientos rígidos y cambios de escala a las primitivas en la escena. A diferencia de los operadores booleanos que eran binarios, estas operaciones se aplican a una única primitiva, y se basarán en aplicar una transformación $t : \mathbb{R}^3 \rightarrow \mathbb{R}^3$ a cada punto de la isosuperficie S_ϕ para obtener la transformada 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 son movimientos rígidos y 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. 1.14.

Definición 1.14 (Operaciones afines). Sea A una isosuperficie. Definimos los SDF para las siguientes operaciones.

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

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

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

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

1.3.3. Operaciones deformantes

Siguiendo el mismo razonamiento, podemos definir operaciones que modifiquen la geometría de la superficie aplicando rotaciones o traslaciones al punto en el que se evalúa la SDF original. De esta forma podemos obtener operadores que de otra forma sería mucho más complicado implementar, como la torsión o el redondeo de bordes.

Definición 1.15 (Operaciones Deformantes). Sea A una isosuperficie. Definimos los SDF para las siguientes operaciones.

- **Torsión:** $\text{sdf}_{\text{torsion}}(p) = \text{sdf}_A(p'), \text{ con } p' = R_z(ky) \cdot (x, z, y)^t,$

1. Fundamentos matemáticos: las SDFs

- **Plegado:** $\text{sdf}_{\text{plegado}} = \text{sdf}_A(p')$, con $p' = R_z(kx) \cdot p^t$,
- **Redondeo:** $\text{sdf}_{\text{redondeo}}(p) = \text{sdf}_A(p) - k$,
- **Desplazamiento:** $\text{sdf}_{\text{desplazamiento}}(p) = \text{sdf}_A(\delta(p))$,
- **Elongación de tamaño $h \in \mathbb{R}^3$:** $\text{sdf}_{\text{elongacion}}(p) = \text{sdf}_A(p')$, con $p' = p - c(p, -h, h)$,

donde

- $k \in \mathbb{R}_0^+$ controla la intensidad de la deformación,
- $\delta: \mathbb{R}^3 \rightarrow \mathbb{R}^3$ 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. 1.14,
- $c: \mathbb{R}^3 \times \mathbb{R}^3 \times \mathbb{R}^3 \rightarrow \mathbb{R}^3$, $c(x, a, b)$ acota cada componente de x entre las de a y b .

1.3.4. Operaciones de repetición

También podemos usar la técnica de cambiar el punto en el que evaluamos el SDF para, en lugar de modificar la geometría original, añadir copias de la primitiva identificando varios puntos con uno que pertenezca a la superficie. La manera más inmediata de conseguir esto es a través de la función valor absoluto, que nos permitirá identificar la componente de cada punto con su opuesta para generar simetrías, y el operador módulo, que identificará puntos a una distancia fija en cada eje.

Definición 1.16 (Operadores de Posicionamiento). Sea A una isosuperficie. Definimos los SDF para las siguientes operaciones.

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

$$\begin{aligned}\text{sdf}_{\text{sim}X}(p) &= \text{sdf}_A(|x|, y, z), & \text{sdf}_{\text{sim}Y}(p) &= \text{sdf}_A(x, |y|, z), \\ \text{sdf}_{\text{sim}Z}(p) &= \text{sdf}_A(x, y, |z|),\end{aligned}$$

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

$$\begin{aligned}\text{sdf}_{\text{sim}XY}(p) &= \text{sdf}_A(|x|, |y|, z), & \text{sdf}_{\text{sim}XZ}(p) &= \text{sdf}_A(|x|, y, |z|), \\ \text{sdf}_{\text{sim}YZ}(p) &= \text{sdf}_A(x, |y|, |z|),\end{aligned}$$

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

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

- **Repetición infinita:**

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

donde

- $c: \mathbb{R} \times \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$, $c(x, a, b) = \min(\max(x, a), b)$ acota x en $[a, b]$,
- $r: \mathbb{R}^3 \rightarrow \mathbb{R}^3$ redondea las componentes de un vector a sus enteros más cercanos.

Observación 1.4. Hay casos en los que los SDF definidos en la Def. 1.16 podrían no ser exactos, al igual que ocurría con la intersección y la diferencia en la Def. 1.12:

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

Este tipo de operaciones evidencia el potencial que tienen las SDFs en cuanto a eficiencia, ya que podemos visualizar miles de objetos al precio de uno. Por ejemplo, podríamos generar un campo de césped a partir de una única brizna de hierba.

1.4. Obtención a partir de ecuación implícita

Empezábamos el capítulo diciendo que una de las representaciones más comunes de una superficie es a través de implícitas, pero hasta ahora nos hemos centrado en estudiar un subconjunto de esta familia. Si intentásemos aplicar el algoritmo de *spheretracing* a una función implícita cualquiera podríamos observar que el resultado presenta defectos, tales como deformaciones o grietas, o que incluso no se visualiza. Veamos qué podemos hacer para, dada una función implícita ϕ , obtener información aproximada de S_ϕ [13]. Esto nos será útil cuando no conozcamos o no podamos calcular explícitamente el SDF de una superficie, pero sí su representación implícita.

Proposición 1.2. *Sea $\phi: \mathbb{R}^3 \rightarrow \mathbb{R}$ una función implícita cualquiera. Entonces*

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

Demostración. Fijamos el punto p del cual queremos aproximar la distancia a S_ϕ . Sea q el punto de S_ϕ más cercano a p y $v = \vec{pq}$. Queremos calcular la distancia de p a S_ϕ , que será justamente $\|v\|$. Podemos realizar el desarrollo de Taylor de ϕ centrado en p :

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

Suponemos ahora que p está cerca de S_ϕ , esto es, $\exists \varepsilon > 0$ tal que $|v| < \varepsilon$, de forma que podemos obviar el residuo. Como $\phi(p + v) = \phi(q) = 0$, tenemos que

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

donde hemos usado la desigualdad triangular y la linealidad del producto escalar. \square

Corolario 1.2. *Sea $\phi: \mathbb{R}^3 \rightarrow \mathbb{R}$ lipschitziana de constante L . Entonces*

$$|sdf_{S_\phi}(p)| \leq \frac{|\phi(p)|}{\|L\|}.$$

1. Fundamentos matemáticos: las SDFs

Este resultado solo nos proporciona una cota superior de la función distancia (sin signo). En nuestro caso esto es suficiente, pues esta nos sigue permitiendo representar la frontera de S_ϕ . En su artículo [14], Pierre-Alain Fayolle describe un método para obtener un SDF asociado a una superficie implícita que representa de manera exacta su frontera. Para ello descompone el SDF como

$$sdf_{S_\phi}(p; \theta) = \phi(p)g(p; \theta) \quad \text{o} \quad sdf_{S_\phi}(p; \theta) = sign(\phi(p))g(p; \theta),$$

donde g es una función paramétrica de parámetros θ y $sign$ es una versión suavizada de la función signo, por ejemplo $sign(x) = \tanh(kx)$, $k \in \mathbb{R}$. Para obtener la expresión de g introduce la función implícita ϕ en la capa final de una red neuronal entrenada para minimizar una función pérdida asociada al SDF, y para ajustar θ expresa $sdf_{S_\phi}(p; \theta)$ como la solución de un problema variacional. No obstante, esta técnica está fuera del ámbito de este trabajo, de forma que nos limitaremos a obtener la cota de la función distancia.

1.5. Obtención a partir de ecuaciones paramétricas

Ahora que sabemos representar las superficies generadas por una función implícita cualquiera, nos proponemos ser capaces de representar también superficies definidas paramétricamente. Para ello fijaremos un anillo comutativo A y un conjunto de variables distintas $X = \{x_1, \dots, x_n\}$. Nuestro objetivo será, dado un conjunto $V \subseteq A^n$ por las ecuaciones paramétricas

$$\begin{aligned} x_1 &= g_1(t_1, \dots, t_r), \\ &\vdots \\ x_n &= g_n(t_1, \dots, t_r), \end{aligned}$$

donde g_i son polinomios de varias variables en A , obtener una ecuación implícita para V . En el caso que nos atañe $A = \mathbb{R}^3$, pero presentaremos todos los resultados de forma general.

El contenido de esta sección está fuertemente basado en el libro *Ideals, Varieties, and Algorithms* de Cox, Little y O’Shea [15], el cual introduce de forma bastante completa resultados y algoritmos de álgebra comutativa. Empezaremos explicando a qué nos referimos con polinomios de varias variables y recordando el concepto de ideal y sus propiedades. Despues veremos que este problema equivale a uno de pertenencia a un ideal y cómo resolverlo usando la teoría de bases de Groebner.

1.5.1. Polinomios en varias variables

Estamos acostumbrados a trabajar con polinomios de una única variable como una suma o colección de monomios. Podemos mantener esta filosofía en el caso de varias variables adaptando el concepto que tenemos de estos.

Definición 1.17. Llamamos **monomio** en X al producto de la forma

$$x_1^{\alpha_1} \cdots x_n^{\alpha_n}, \quad \alpha_i \in \mathbb{N}, \quad i \in \{1, \dots, n\}.$$

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

Definición 1.18. Llamamos **grado** de $\alpha \in \mathbb{N}^n$ a $|\alpha| = \alpha_1 + \dots + \alpha_n$.

Definición 1.19. Definimos el **polinomio** $f : A^n \rightarrow A$ en X con coeficientes en A a toda combinación lineal finita de monomios

$$f = \sum_{\alpha \in \mathbb{N}^n} a_\alpha X^\alpha.$$

Proposición 1.3. El conjunto de polinomios es un anillo conmutativo con las operaciones

- Suma heredada de A : $(f + g)(a) = f(a) + g(a)$,
- Producto de convolución: $(fg)(a) = \sum_{\alpha} \sum_{\beta+\gamma=\alpha} f(a)g(a)$,

donde

$$f = \sum_{\alpha \in \mathbb{N}^n} a_\alpha X^\alpha \quad y \quad g = \sum_{\beta \in \mathbb{N}^n} b_\beta X^\beta.$$

Denotaremos como $A[X] = A[x_1, \dots, x_n]$ a este anillo.

En las siguientes secciones veremos que el problema de pertenencia de polinomios a un ideal se puede resolver mediante el procedimiento de la división. Este es bien conocido en polinomios de una variable, y ahora queremos extenderlo a un número arbitrario de ellas y varios divisores. Para ello en primer lugar necesitaremos una forma de ordenar los monomios que forman un polinomio. En una variable la forma “natural” de comparar dos monomios es a través de su exponente. En el caso de varias variables la elección no es tan clara, y hay varias opciones que parecen igual de válidas. Vamos a formalizar el concepto de orden para introducir algunas de las posibilidades de las que disponemos.

Definición 1.20. Un **orden total** sobre un conjunto Δ es una relación binaria \leq que cumple las propiedades

1. Reflexiva: $a \leq a$, $\forall a \in \Delta$,
2. Transitiva: $a \leq b$ y $b \leq c \implies a \leq c$, $\forall a, b, c \in \Delta$,
3. Antisimétrica: $a \leq b$ y $b \leq a \implies a \leq b$, $\forall a, b \in \Delta$,
4. Completitud: $a \leq b$ o $b \leq a$, $\forall a, b \in \Delta$.

Definición 1.21. 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. $\alpha < \beta \implies \alpha + \gamma < \beta + \gamma$, $\forall \alpha, \beta, \gamma \in \mathbb{N}^n$.

Proposición 1.4. Todo orden admisible es un buen orden, esto es, todo subconjunto no vacío tiene un elemento mínimo.

A partir de ahora siempre supondremos que todo orden que usemos es admisible en \mathbb{N}^n , luego podremos ordenar los monomios que conforman un polinomio ordenando sus exponentes según dicho orden. Veamos algunos de los órdenes más usados.

Definición 1.22. Definimos el **orden lexicográfico** \leq_{lex} como

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

1. Fundamentos matemáticos: las SDFs

Definición 1.23. Dado $\omega \in \mathbb{R}^n$, un orden admisible \leq se dice **ω -graduado** cuando

$$\alpha \leq \beta \implies \langle \alpha, \omega \rangle < \langle \beta, \omega \rangle,$$

donde $\langle \alpha, \omega \rangle$ se llama el w -grado de α y se define como

$$\langle \alpha, \omega \rangle = \alpha_1 \omega_1 + \cdots + \alpha_n \omega_n.$$

Definición 1.24. Dado un orden admisible \leq , definimos el **orden ω -graduado asociado** como

$$\alpha \leq_\omega \beta \iff \begin{cases} \langle \alpha, \omega \rangle < \langle \beta, \omega \rangle \\ \text{o} \\ \langle \alpha, \omega \rangle = \langle \beta, \omega \rangle \text{ y } \alpha \leq \beta \end{cases}.$$

Cuando $\omega = (1, \dots, 1)$ simplemente diremos que el orden es graduado, y usaremos las notaciones

$$\leq_{(1, \dots, 1)} = \leq_{\text{deg}}, \quad (\leq_{\text{lex}})_{\text{deg}} = \leq_{\text{deglex}}, \quad (\leq_{\text{lex}})_\omega = \leq_{\omega\text{-lex}}.$$

Definición 1.25. Definimos el **orden lexicográfico graduado inverso** $\leq_{\text{degrevallex}}$ como:

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

Proposición 1.5. Sea \leq un orden admisible. Entonces \leq_ω es admisible.

Proposición 1.6. Los órdenes \leq_{lex} y $\leq_{\text{degrevallex}}$ son admisibles.

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

Definición 1.26. Sea $f = \sum_\alpha a_\alpha X^\alpha$ un polinomio y \leq un orden admisible. Definimos los siguientes conceptos asociados a f .

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

Antes de presentar el algoritmo de la división, nos cercioramos de que esta siempre tiene sentido con el siguiente teorema.

Teorema 1.2 (Algoritmo de división). *Sea $F = \{f_1, \dots, f_s\} \subset A[X]$. 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 $r = 0$ ó $\exp(r) \leq \exp(f)$. Llamaremos a r el resto de dividir f por F , y lo notaremos $\text{rem}(f, [F]) = r$. Además, cuando $r = 0$ diremos que f reduce a 0, y escribiremos $f \xrightarrow{G} 0$.

En otras palabras, podemos dividir f entre cualquier conjunto de polinomios $F = \{f_1, \dots, f_s\}$ para expresarlo como combinación de sus elementos multiplicados por ciertos coeficientes polinómicos. El método es similar al usado en una variable, consistente en intentar reducir el monomio líder de f restándole un múltiplo de cierto f_i . Para encontrar este f_i simplemente se recorre el conjunto F hasta encontrar uno válido, y de no haberlo se pasa el término líder al resto y se continua con el siguiente. Esto es justo lo que hace el **Algoritmo 1**. Cabe destacar que esta forma de buscar el f_i hace que la descomposición de f obtenida no sea única, pues la elección dependerá de la posición que ocupen los divisores en el conjunto F , y por tanto del orden elegido. Más adelante veremos que la elección del orden influye en el resultado de más algoritmos.

Algorithm 1: División de polinomios en varias variables

Data: dividendo f , divisores $F = [f_1, \dots, f_s]$
Result: Tupla con el resto r y los coeficientes q_i para cada $f_i \in F$

```

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

```

1.5.2. Bases de Groebner

Ya tenemos claras las ideas sobre qué es un polinomio en varias variables, así que ahora pasamos a repasar el concepto de ideal y como podemos usar las bases de Groebner para trabajar con ellos en el caso de ideales de polinomios.

Definición 1.27. Decimos que $\emptyset \neq I \subseteq A[X]$ es un **ideal** de $A[X]$ si:

1. $0 \in I$,
2. $a + b \in I$, $\forall a, b \in I$,
3. $af \in I$, $\forall a \in I$, $\forall f \in A[x]$.

Lo denotaremos como $I \leq A$.

Proposición 1.7. Dados los ideales $I, J \leq A[X]$, son también ideales de $A[X]$:

1. Fundamentos matemáticos: las SDFs

1. $I + J = \{f + g : f \in I, g \in J\}$,
2. $IJ = \{f_1g_1 + \cdots + f_tg_t : f_i \in I, g_i \in J, 1 \leq i \leq t\}$,
3. $I \cap J = \{h : h \in I \text{ y } h \in J\}$.

Podemos calcular estos ideales usando sus conjuntos de generadores.

Definición 1.28. Dado $F = \{f_1, \dots, f_s\} \subseteq A[X]$, el **ideal generado** por F es

$$\langle F \rangle = \{a_1f_1 + \cdots + a_sf_s : a_1, \dots, a_s \in A, f_1, \dots, f_s \in F\} \leq A[X].$$

Diremos que F es un **conjunto de generadores** de I .

Proposición 1.8. Sean $I = \langle F \rangle$ y $J = \langle G \rangle$ ideales de $A[X]$. Entonces:

1. $I + J = \langle F \cup G \rangle$,
2. $IJ = \langle fg : f \in F, g \in G \rangle$,
3. $I \cap J = \langle tF, (1-t)G \rangle \cap A[x_1, \dots, x_n]$.

Pasamos a presentar el concepto de base de Groebner asociada a un ideal. Podemos pensar que una 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.

Definición 1.29. Sea $I \leq A[X]$. Denotamos el conjunto de los términos líder de I como

$$\text{lt}(I) = \{\text{lt}(f) : f \in I\}.$$

Definición 1.30. Dado $I \leq A[X]$, diremos que $G = \{g_1, \dots, g_s\} \subseteq I$ es una **base de Groebner** para I si

$$\langle \text{lt}(I) \rangle = \langle \text{lt}(g_1), \dots, \text{lt}(g_s) \rangle.$$

La analogía con el sistema de generadores de un espacio vectorial nos conduce de forma natural a la pregunta de si habrá también un análogo al concepto de base, y si dado un ideal I siempre existirá una base de Groebner asociada a este. La respuesta es afirmativa en ambos casos.

Definición 1.31. Dada $f \in A[X]$, definimos su **soporte** como

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

Definición 1.32. Sea $I \leq A[X]$. Diremos que G es una **base de Groebner reducida** para I si para todo $g \in G$ se cumple

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

Es decir, una base será reducida si ningún elemento se puede expresar como combinación del resto. Esto equivale a

$$\text{rem}(g, G \setminus \{g\}) \neq 0, \forall g \in G.$$

Definición 1.33. Sea $M \leq \mathbb{N}^n$. Decimos que A es un **conjunto generador minimal** de M si

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

Lema 1.3. Todo ideal tiene un único conjunto generador minimal.

Lema 1.4. Sea $I \leq A$. Se cumple $a - b \in I, \forall a, b \in I$.

Demostración. Basta observar que tomando $-1 \in A$ obtenemos que $b \cdot (-1) \in I$, de donde

$$a - b = a + b(-1) \in I.$$

□

Teorema 1.3. 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 el [Lema 1.3](#). Sea $g \in G$ y $r = \text{rem}(g, [G \setminus \{g\}])$. Tenemos que

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

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

Unicidad Sean G_1, G_2 dos bases minimales de I . Por el [Lema 1.3](#),

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

Por otro lado,

$$\left. \begin{aligned} \text{supp}(g_1 - g_2) &\subseteq \left(\text{supp}(g_1) \cup \text{supp}(g_2) \right) \setminus \{\exp(g_1)\} \\ \text{supp}(g_i) \setminus \{\exp(g_i)\} &\cap \left(\{\exp(G_i) + \mathbb{N}^n\} \right) = \emptyset, \quad i \in \{1, 2\} \end{aligned} \right\} \implies$$

$$\implies \text{supp}(g_1 - g_2) \cap (\exp(G_1) + \mathbb{N}^n) = \emptyset \implies \text{rem}(g_1 - g_2, G_1) = g_1 - g_2.$$

Finalmente, como $g_2 - g_1 \in I$ por el [Lema 1.5.2](#), obtenemos que

$$\text{rem}(g_1 - g_2, G_1) = 0 \implies g_1 = g_2 \implies G_1 = G_2.$$

□

Terminamos la sección obteniendo un algoritmo para calcular la base de Groebner reducida para un ideal dado su conjunto de generadores G . Este se basará en eliminar de G aquellos polinomios cuyos exponentes podamos poner como combinación lineal del resto. Claro está que no podemos realizar esta comprobación directamente, y deberemos buscar alguna condición equivalente que sí podamos calcular.

Definición 1.34. Dados $\alpha, \beta \in \mathbb{N}^n$, definimos los términos

1. Fundamentos matemáticos: las SDFs

- **Mínimo común múltiplo:** $\text{lcm}(\alpha, \beta) = \{\max(\alpha_1, \beta_1), \dots, \max(\alpha_n, \beta_n)\}$,
- **Máximo común divisor:** $\text{gcd}(\alpha, \beta) = \{\min(\alpha_1, \beta_1), \dots, \min(\alpha_n, \beta_n)\}$.

Definición 1.35. Sean $f, g \in A[X]$. Tomando $\alpha = \exp(f)$, $\beta = \exp(g)$ y $\gamma = \text{lcm}(\alpha, \beta)$, se define el **S-polinomio** de f y g como

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

Teorema 1.4 (Primer Criterio de Buchberger). *Sean $I \leq A[X]$ y $G = \{g_1, \dots, g_t\}$ un conjunto de generadores de I . Entonces:*

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

El algoritmo que usaremos para el cálculo de la base de Groebner se basará en este criterio. Sin embargo, antes de presentarlo estudiaremos dos criterios adicionales [16] [17] que lo harán más eficiente descartando S-polinomios antes de comprobar su resto, ahorrando el cómputo de numerosas divisiones.

Teorema 1.5 (Criterios de Buchberger). *Sean $I \leq A[X]$, $G \subseteq A[X]$ un conjunto de generadores de I , y $g_1, g_2 \in G$. Si se cumple cualquiera de las siguientes condiciones entonces $S(g_1, g_2) \xrightarrow{G} 0$.*

1. $\text{lcm}(g_1, g_2) = \text{lm}(g_1)\text{lm}(g_2)$,
 2. $\exists f \in G: \text{lm}(f) \mid \text{lcm}(g_1, g_2)$ y además
 - a) algún $S(g_i, f) \xrightarrow{G} 0$ ó
 - b) $\text{lm}(f) \mid \frac{\text{lm}(g_i)}{\text{gcd}(g_1, g_2)}$ y $\text{sm}(g_j)\text{lm}(f) \neq \text{sm}(f)\text{lm}(g_j)$,
- donde $i, j \in \{1, 2\}$ e $i \neq j$.

Usando los criterios obtenidos obtenemos el **Algoritmo 2** para calcular la base de Groebner de cualquier ideal. La salida de este no es una base minimal, pero la demostración del **Teorema 1.3** nos proporciona un método para reducir una base cualquiera a la minimal asociada, y mostramos en el **Algoritmo 3**.

Ya somos capaces de obtener una base de Groebner minimal de cualquier ideal dado un conjunto de generadores suyo, pero en este proceso se toma una decisión que aún no hemos discutido: 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, y consiste en tomar el par f, g cuyo $\text{lcm}(f, g)$ sea del menor grado posible según el orden admisible usado. Vemos por tanto que de nuevo la elección de un orden u otro nos proporcionará resultados diferentes, y en este caso esto se traduce en que la base reducida tenga muchos menos elementos en un orden que en otro.

1.5.3. Implicitación Polinomial

Empezábamos la sección diciendo que el problema de implicitación equivalía al de pertenencia a un ideal. Antes de ver de qué ideal se trata tenemos que introducir unos últimos conceptos que nos ayuden a entender por qué.

Algorithm 2: Algoritmo de Buchberger optimizado

Data: polinomio f , conjunto de generadores $F = [f_1, \dots, f_s]$
Result: base de Groebner G

```

 $G \leftarrow F;$ 
repeat
     $G' \leftarrow G;$ 
    for each pair  $\{f, g\} \subseteq G'$  do
        if  $\neg \text{Criterio 1}(f, g)$  AND  $\neg \text{Criterio 2}(f, g, G')$  then
             $r \leftarrow \text{rem}(S(f, g), G');$ 
            if  $r \neq 0$  then
                 $G \leftarrow G \cup \{r\};$ 
            end
        end
    end
until  $G' = G$ ;
return  $G$ 

```

Algorithm 3: Minimización de base de Groebner

Data: G base a minimizar
 $G \leftarrow F;$

```

foreach  $g \in G$  do
     $g \leftarrow g/\text{lc}(g);$ 
     $r \leftarrow \text{rem}(g, [G \setminus \{g\}]);$ 
    if  $r \neq 0$  then
         $g \leftarrow r;$ 
    end
end

```

Definición 1.36. Dado $F = \{f_1, \dots, f_s\} \subseteq A[X]$, llamamos **variedad afín** definida por F al conjunto:

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

Proposición 1.9. Sean $\mathbb{V}(F)$ y $\mathbb{V}(G)$ variedades afines. Entonces

- $\mathbb{V}(F) = \mathbb{V}(\langle F \rangle)$,
- $\mathbb{V}(F \cup G) = \mathbb{V}(F) \cap \mathbb{V}(G)$,
- $\mathbb{V}(FG) = \mathbb{V}(F) \cup \mathbb{V}(G)$.

Proposición 1.10. Sean los ideales $I, J \subseteq A[X]$. Entonces $\mathbb{V}(I \cap J) = \mathbb{V}(I) \cup \mathbb{V}(J)$.

Definición 1.37. Sea $B \subseteq A^n$. Definimos el **ideal asociado** a B como

$$\mathbb{I}(B) = \{f \in A[X] : f(b_1, \dots, b_n) = 0, \forall (b_1, \dots, b_n) \in B\}.$$

Para resolver el problema de implicación deberemos aprender antes a eliminar variables de un ideal.

1. Fundamentos matemáticos: las SDFs

Definición 1.38. Dado $I \leq A[x_1, \dots, x_n]$, diremos que su **ideal de l -eliminación** es:

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

Definición 1.39. 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 1.6 (Eliminación). *Sea $I \leq A[x_1, \dots, x_n]$ 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*

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

Con estos resultados podemos decir que el problema de implicitación consiste en encontrar la variedad asociada a las ecuaciones paramétricas

$$x_1 = g_1(t_1, \dots, t_r),$$

⋮

$$x_n = g_n(t_1, \dots, t_r).$$

Si escribimos $g_i = f_i/q_i$ con $f_i, q_i \in A[t_1, \dots, t_r]$, $i = 1, \dots, r$, podemos definir la aplicación

$$\begin{aligned} \phi: A^r \setminus W &\rightarrow A^n, \\ (a_1, \dots, a_r) &\mapsto \left(\frac{f_1(a_1, \dots, a_r)}{q_1(a_1, \dots, a_r)}, \dots, \frac{f_n(a_1, \dots, a_r)}{q_n(a_1, \dots, a_r)} \right), \end{aligned}$$

donde $W = \mathbb{V}(q_1 \cdots q_r)$. Veamos cómo encontrar la menor variedad que contiene la imagen de ϕ cuando $q_i = 1$ para cada $i = 1, \dots, r$.

Teorema 1.7 (Implicitación Polinomial). *Dados $f_1, \dots, f_n \in A[t_1, \dots, t_r]$ con A cuerpo infinito, sea*

$$\begin{aligned} \phi: A^r &\rightarrow A^n, \\ (a_1, \dots, a_r) &\mapsto (f_1(a_1, \dots, a_r), \dots, f_n(a_1, \dots, a_r)). \end{aligned}$$

Definimos los ideales:

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

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

La extensión al caso racional es la siguiente.

Teorema 1.8 (Implicitación Racional). *Sea $f_1, \dots, f_n, q_1, \dots, q_n \in A[t_1, \dots, t_r]$ con A cuerpo*

infinito, $W = \mathbb{V}(q_1, \dots, q_n)$ y

$$\begin{aligned}\phi: A^r \setminus W &\rightarrow A^n, \\ (a_1, \dots, a_r) &\mapsto \left(\frac{f_1(a_1, \dots, a_r)}{q_1(a_1, \dots, a_r)}, \dots, \frac{f_n(a_1, \dots, a_r)}{q_n(a_1, \dots, a_r)} \right).\end{aligned}$$

Definimos los ideales:

- $I = \langle q_1x_1 - f_1, \dots, q_nx_n - f_n, 1 - q_1 \cdots q_n y \rangle \leq A[y, t_1, \dots, t_r, x_1, \dots, x_n]$,
- $J = I \cap A[x_1, \dots, x_n]$ el ideal de 1 + r-eliminación de I .

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

Observación 1.5. En el caso $r = 1$ y cuando f_i y q_i sean primos relativos para cada $1 \leq i \leq n$, basta tomar

$$I = \langle q_1x_1 - f_1, \dots, q_nx_n - f_n \rangle.$$

Con este resultado, una vez obtenida la variedad, si resulta que esta tiene un único generador esta será la ecuación implícita, luego el ideal al que llevamos haciendo referencia desde el principio de la sección y del que queríamos comprobar la pertenencia es el ideal J de los teoremas anteriores. Sin embargo, no tenemos asegurado que vaya a haber un único generador del ideal, de forma que la superficie satisfaría varias ecuaciones implícitas y no podría ser representada por una sola. A continuación presentamos un resultado que aporta información al respecto.

Definición 1.40. Dado un ideal $I \leq A[X]$, definimos su radical como

$$\sqrt{I} = \{f \in A[X] : f^m \in I \text{ para algún } m \in \mathbb{N}\}.$$

Proposición 1.11. Sea $I \leq A[X]$. Entonces \sqrt{I} es un ideal y contiene a I .

Definición 1.41. Decimos que un ideal $I \leq A[X]$ es **radical** si $\sqrt{I} = I$.

Proposición 1.12. Sea $B \subseteq A^n$. Entonces $\mathbb{I}(B)$ es un ideal radical.

Teorema 1.9 (Nullstellensatz fuerte). Si A es algebraicamente cerrado, dado un ideal $I \leq A[X]$ se cumple

$$\sqrt{I} = \mathbb{I}(\mathbb{V}(I)).$$

Proposición 1.13. Sea $I \leq A[X]$ y $f \in A[X]$. Entonces

$$f \in \sqrt{I} \iff \langle I \rangle + \langle 1 - fy \rangle = A[X].$$

Así, si pudiéramos calcular el radical del ideal J de los teoremas de implicitación y este tuviera un solo elemento, tendríamos asegurado que la variedad está generada por esa única ecuación implícita. Sin embargo, el cálculo del radical o su número de elementos es en general muy complicado, y no es una opción viable. Por tanto, lo que haremos será simplemente aplicar el algoritmo y comprobar si efectivamente se obtiene un único generador, en cuyo caso contrario concluiremos que no podemos realizar la implicitación.

Otra vía que permite la eliminación de variables y por tanto resolver el problema de implicitación es el uso de resultantes. TODO: añadir sección resultantes

2. Algoritmos de visualización de SDFs

2.1. Renderizado por *spheretracing*

Una vez definida la escena a partir de un SDF necesitamos una forma para visualizarla, para lo que utilizaremos la API de OpenGL [18] y aplicaremos la técnica de *raymarching*.

2.1.1. 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 2.1](#).

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* utilizaremos

- **in vec4 gl_Vertex:** contiene las coordenadas locales del vértice actual y es pasado automáticamente 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.

2. Algoritmos de visualización de SDFs

```

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();

```

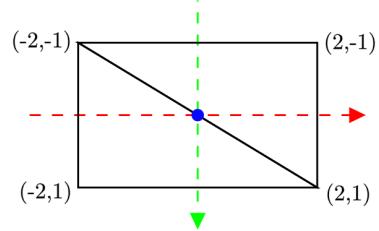


Figura 2.1.: Construcción del lienzo

En el *fragment shader* utilizaremos

- **in vec4 gl_FragCoord:** coordenadas de dispositivo para el centro del 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 [19] [20]. En particular, usaremos las que siguen.

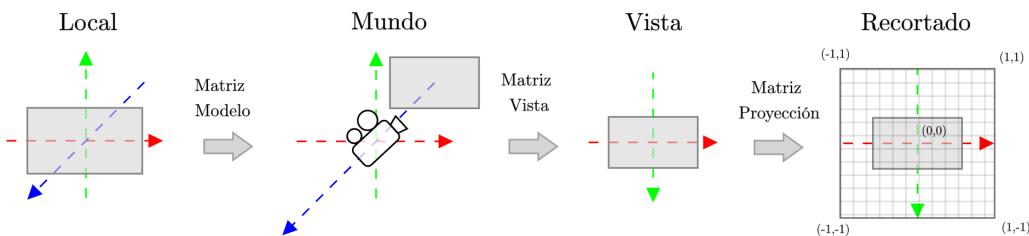


Figura 2.2.: Coordenadas locales a recortadas

- **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. 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*:

Algorithm 4: Fragment Shader

Data: matriz de proyección M_P , matriz de vista M_V , matriz de modelo M_M

Result: Posición transformada del vértice actual

$gl_Position \leftarrow M_P \cdot M_V \cdot M_M \cdot gl_Vertex$

Figura 2.3.: Cuerpo del método main del *vertex shader*

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 para 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`,

2. Algoritmos de visualización de SDFs

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. Finalmente, para que el eje vertical tome valores en $[-1, 1]$ multiplicamos por 2, obteniendo

$$uv = \frac{2 \cdot (gl_FragCoord.xy - 0.5 \cdot u_resolution.xy)}{u_resolution.y}.$$

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 2.7](#) (los valores que se salen del rango $[-1, 1]$ son visualizados como si hubieran sido acotados en dicho intervalo).

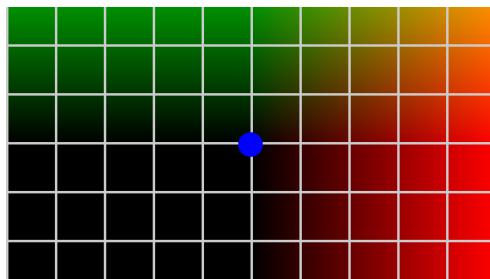


Figura 2.4.: Eje X

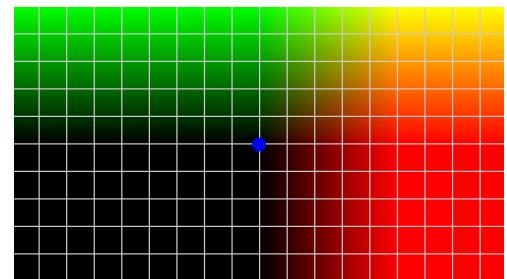


Figura 2.5.: Eje Y

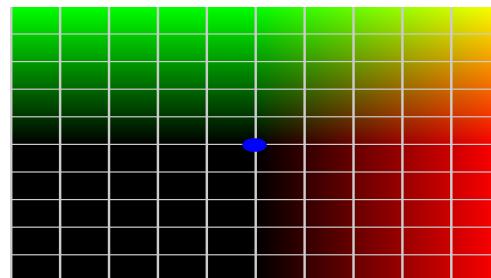


Figura 2.6.: Ejes X e Y

Figura 2.7.: Normalización de coordenadas sobre distintos ejes

Veamos cómo usar estas coordenadas para dibujar nuestra superficie sobre el lienzo.

2.1.2. 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 $B_u = \{e_x, e_y, e_z\} = \{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}$, donde colocamos los siguientes elementos.

- La **isosuperficie** S_ϕ ,
- **Plano de visión:** rejilla perpendicular al eje óptico de la cámara, 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,
- **Punto de atención o lookat point** l : hacia que punto del espacio debe mirar la cámara. En general tomaremos $l = (0, 0, 0)$.

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.

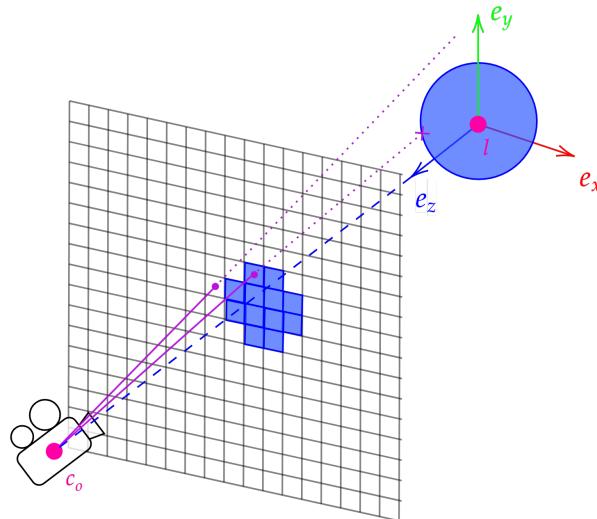


Figura 2.8.: Trazado de rayos a través del plano de visión

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 2.8 en el que S_ϕ es una esfera centrada en el origen y el observador se encuentra sobre el eje Z, 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

$$r_d = (u, v, 0) - c_0.$$

2. Algoritmos de visualización de SDFs

Tomando $c_0 = (0, 0, c)$, el valor c actuaría 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 manteniendo la orientación hacia l tendremos que poder trabajar con orientación arbitraria suya. Para ello deberemos construir un marco cartesiano relativo a la cámara, esto es, una base $B = \{f_1, f_2, f_3\}$ de \mathbb{R}^3 alineada con ella. 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 $c_r = (0, 1, 0) \times c_d$,
- **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:

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

Ahora solo queda transformar el vector $r_d = (u, v, -1)$ 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 $\{e_1, e_2, e_3\}$, que al ser la base usual, coincidirá con escribir por columnas $\{f_1, f_2, f_3\}$, de forma que

$$\text{rayo} = (u, v, -1)_B^t = (f_1 \mid f_2 \mid f_3) \cdot \begin{pmatrix} u \\ v \\ -1 \end{pmatrix}.$$

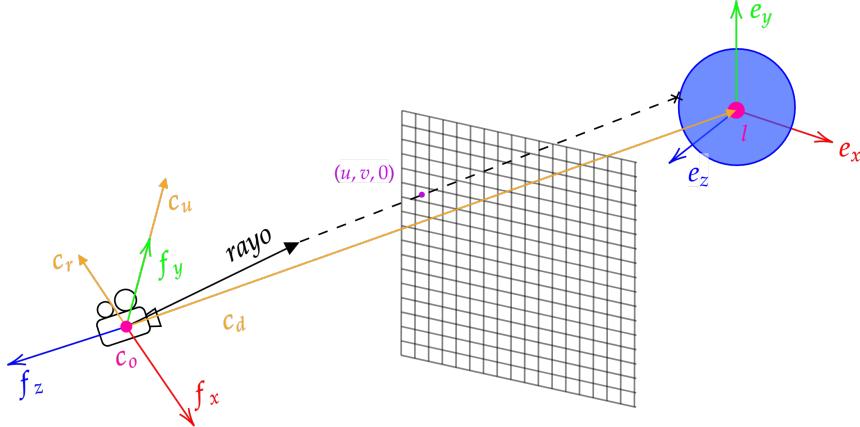


Figura 2.9.: Obtención de la dirección del rayo

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 2.11](#) ilustra este procedimiento, donde `DibujarSuperficie()` y `DibujarFondo()` devuelven ternas RGBA que serán asignadas al píxel actual dependiendo de si hay intersección o no.

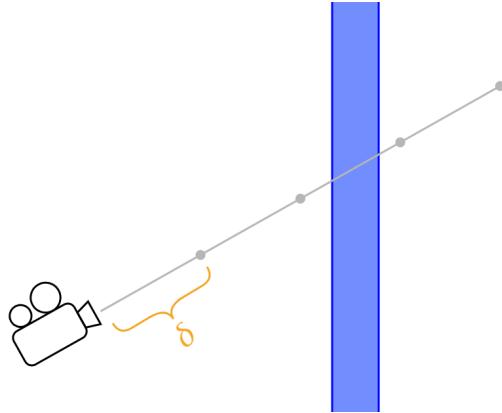


Figura 2.10.: Pérdida de intersección en *raymarching* para valores elevados de δ

Algorithm 5: Raymarching

Data: origen c_o , dirección v
Result: Terna RGB con el color asignado
 al píxel actual

```

 $d \leftarrow 0 //$  distancia total
for  $i \in MAX\_ITERACIONES$  do
     $p \leftarrow c_o + d \cdot v$ 
     $sdf \leftarrow \phi(p)$ 
    if  $sdf < \epsilon$  then
        return DibujarSuperficie( $p, v, sdf$ )
    end
     $d \leftarrow d + \delta;$ 
    if  $d > MAX\_DISTANCIA$  then
        return DibujarFondo()
    end
end

```

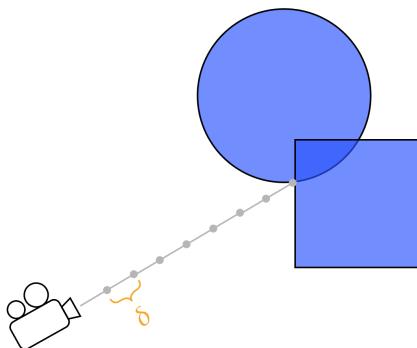


Figura 2.11.: Algoritmo de *raymarching*

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 alto, pues si no queremos perder ninguna intersección como ocurre en la [Figura 2.29](#), el valor

2. Algoritmos de visualización de SDFs

de incremento δ tendrá 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.

Este será por tanto el algoritmo que utilizaremos para detectar qué píxeles de la pantalla corresponden a la superficie S_ϕ , y se encuentra descrito en la [Figura 2.12](#).

Algorithm 6: Spheretracing

Data: origen c_o , dirección v
Result: Terna RGB con el color asignado
al píxel actual
 $d \leftarrow 0$ // distancia actual
for $i \in \text{MAX_ITERACIONES}$ **do**
 $p \leftarrow c_o + d \cdot v$
 $\text{sdf} \leftarrow \phi(p)$
 if $\text{sdf} < \varepsilon$ **then**
 | **return** *DibujarSuperficie*(p, v, sdf);
 end
 $d \leftarrow d + \text{sdf}$
 if $d > \text{MAX_DISTANCIA}$ **then**
 | **return** *DibujarFondo*();
 end
end

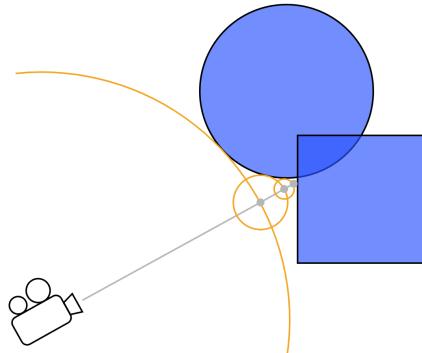


Figura 2.12.: Algoritmo de *spheretracing*

Con esto al fin podemos describir la forma que tendrá nuestro *fragment shader* en la [Figura 2.13](#).

Algorithm 7: Fragment Shader

Data: posición del observador c_0 , punto de atención l
Result: Terna RGBA con el color asignado al píxel actual
 $uv \leftarrow \frac{gl\text{_FragCoord.xy} - 0.5 \cdot u\text{_resolution.xy}}{u\text{_resolution.y}}$
 $r_d \leftarrow (f_1 | f_2 | f_3) \cdot \text{normalizar}((uv.xy, -1))$
 $\text{color} \leftarrow \text{spheretracing}(c_0, r_d)$
 $gl\text{_FragColor} = (\text{color}, 1)$

Figura 2.13.: Cuerpo del método *main* del *fragment shader*

Claro está que esta versión todavía no es funcional, pues no sabemos qué forma tiene *DibujarSuperficie*, y como mucho podremos obtener una imagen que separe la isosuperficie

del fondo usando colores planos como muestra la [Figura 2.14](#) para una esfera centrada en el origen. Veremos como mejorar esto en la próxima sección.

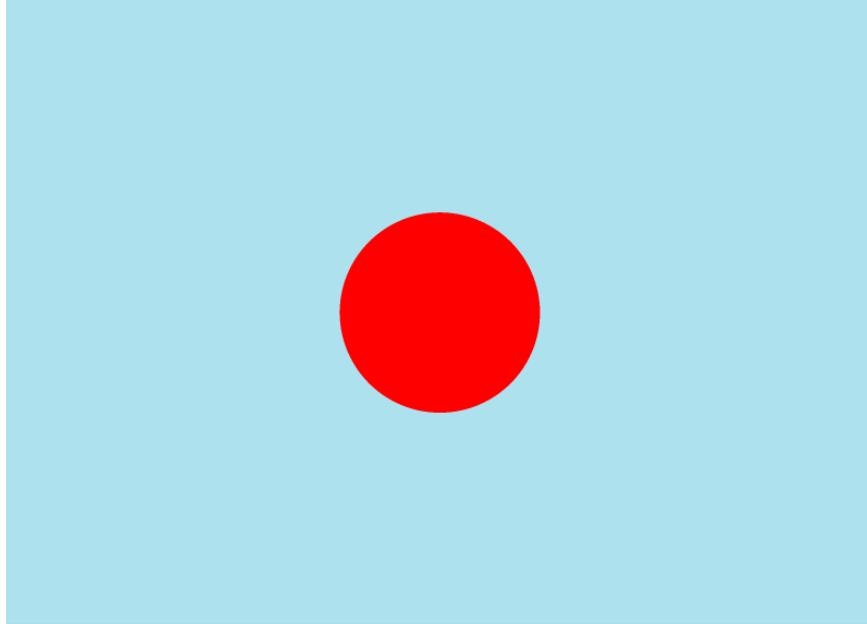


Figura 2.14.: Resultado de *spheretracing* asignando colores planos

2.2. Modelos de iluminación y sombras

Ya sabemos qué píxeles pertenecen a la superficie, pero no de qué color deben dibujarse. En esta sección estudiaremos diversas técnicas que en conjunto nos permitirán simular de forma plausible qué ocurre cuando se añaden a la escena una o varias fuentes de luz.

2.2.1. Modelos de Blinn y Blinn-Phong

Empezamos viendo cómo las fuentes de luz presentes en la escena iluminan directamente la superficie. Hay multitud de modelos que simulan este comportamiento de forma más o menos realista, siendo uno de los más extendidos el renderizado basado en física (*physically based rendering* o PBR). Sin embargo, este y otros acercamientos similares son utilizados cuando se requiere de un alto grado de fidelidad y adaptabilidad. Nosotros usaremos el modelo de reflexión de Blinn-Phong [21], también popular pero mucho más simple y computacionalmente menos costoso. A su vez este modelo se basa en el de Blinn, el cual pasamos a estudiar a continuación.

Vamos a considerar que nuestra escena consta de los siguientes elementos.

- La **isosuperficie** S_ϕ como único objeto a ser dibujado (aunque el modelo es válido para cualquier número de objetos en escena),
- Un **observador** que se encuentra en la posición $c_o \in \mathbb{R}^3$ mirando a un punto $p \in \mathbb{R}^3$,

2. Algoritmos de visualización de SDFs

- Un número finito n de **fuentes de luz**. Llamaremos l_i con $i = 1, \dots, n$ a los vectores normalizados que apuntan desde p a la posición de cada fuente.

Empecemos comprendiendo el fenómeno físico que tratamos de simular. La luz que generan las fuentes no es más que radiación electromagnética. De forma ideal, esta radiación se puede ver como un flujo en el espacio de partículas llamadas **fotones** que siguen trayectorias rectilíneas a la par que interaccionan con el entorno. Cada una de estas partículas tendrá una energía radiante única en función de su longitud de onda, que irá transfiriendo a aquellos objetos con los que interaccione.

Definición 2.1 (Radiancia). Dado un punto $p \in \mathbb{R}^3$, llamamos **radiancia** a la densidad de energía radiante por unidad de tiempo de los fotones que pasan por un entorno de p en una determinada dirección $v \in \mathbb{R}^3$ con $\|v\| = 1$. La denotaremos $L(p, v)$, y será representada mediante una terna RGB no acotada. Podemos distinguir a su vez varios tipos de radiancia.

- **Radiancia emitida** $L_E(p, v)$: radiancia que emite el propio objeto, también llamada emisividad. Normalmente es de intensidad baja y la consideraremos constante.
- **Radiancia incidente** $L_I(p, v)$: radiancia que recibe el punto p desde la dirección v .
- **Radiancia reflejada** $L_R(p, v)$: cantidad de la radiancia incidente en p que se refleja en la dirección v .

El objetivo del modelo será por tanto describir la radiancia que percibe el observador desde su posición en el punto p . Para ello, se llevan a cabo una serie de simplificaciones:

- En un modelo físicamente correcto la luz reflejada en cada punto se dispersaría por el entorno, contribuyendo a la radiancia incidente en otros puntos de la escena. Sin embargo, nosotros no consideraremos la radiancia incidente que no provenga directamente de fuentes de luz. Incluso teniendo un solo objeto en escena como es nuestro caso este modelo es mejorable, pues el objeto puede reflejar radiancia sobre sí mismo. Por tanto, usaremos una radiancia ambiente constante L_A para suplir esta iluminación indirecta.
- La radiancia se conserva en el espacio entre objetos.
- Las fuentes de luz son direccionales, de forma que no serán visibles en la escena. Además supondremos que emiten una radiancia constante S_i para $i = 1, \dots, n$.
- No se consideran objetos con transparencia.

Es natural pensar que la radiancia percibida en un punto $p \in \mathbb{R}^3$ será la suma de la radiancia que emita y la que sea capaz de reflejar. Así, teniendo en cuenta las consideraciones anteriores tenemos que

$$L(p, v) = L_A + L_E + \sum_{i=1}^n L_R(p, l_i).$$

Como L_A y L_E son constantes sólo nos falta estudiar cómo obtener la **radiancia reflejada** para cada fuente de luz. Para ello fijamos un índice $m \in \{1, \dots, n\}$ y suponemos a partir de ahora que $p \in S_\phi$, pues de lo contrario

$$L(p, v) = L_A, \quad p \in \mathbb{R}^3 \setminus S_\phi,$$

ya que la única luz que se puede reflejar es la del ambiente, la cual ya está siendo considerada con L_A , y al trabajar únicamente con fuentes de luz direccionales $L_E = 0$.

Sabemos que cada objeto refleja la luz de manera distinta en función de su material y las propiedades de la fuente. Para representar este comportamiento definimos una función que indique la fracción de radiancia proveniente de l_m que se refleja en un punto p en la dirección v para cada fuente de luz

$$f_r: \mathbb{R}^3 \times \mathbb{R}^3 \times \mathbb{R}^3 \rightarrow \mathbb{R}^3.$$

Así, la radiancia reflejada sería

$$L_R(p, v, l_m) = S_m \cdot f_r(p, v, l_m).$$

Podemos distinguir diferentes tipos de reflexión, cada uno contribuyendo de forma diferente a la radiancia reflejada final.

- **Reflexión ambiental:** cantidad de iluminación indirecta proveniente de la fuente de luz que refleja el objeto. Al igual que hicimos con L_A , tomaremos un valor constante R_A para ella, de forma que la fracción de radiancia ambiental reflejada será

$$f_{ra} = R_A \in \mathbb{R}^3.$$

- **Reflexión especular:** define cómo se refleja la luz en objetos brillantes teniendo en cuenta la posición de la fuente de luz y la del observador. Según la ley de refracción, el ángulo de incidencia de la luz será igual al de reflexión, luego podemos obtener la dirección de reflexión r_m reflejando l_m sobre el vector normal unitario en p de la superficie, que llamaremos N_p . Así,

$$r_m = 2(l_m \cdot N_p)N_p - l_m \in \mathbb{R}^3.$$

Sin embargo, solo queremos que haya reflejos en los puntos orientados hacia la fuente de luz y cuando r_m haya sido reflejado en una dirección que el observador pueda apreciar, siendo la intensidad del reflejo mayor cuanto más alineado esté el observador con el vector reflejado. Esto equivale a que se cumpla

$$N_p \cdot l_m > 0 \quad \text{y} \quad R_m \cdot v > 0.$$

Para controlar el color y la intensidad de los reflejos usaremos una radiancia R_E , de forma que podemos expresar la fracción de radiancia especular reflejada como

$$\begin{aligned} f_{re}: \mathbb{R}^3 \times \mathbb{R}^3 \times \mathbb{R}^3 &\rightarrow \mathbb{R}^3, \\ (p, v, l_m) &\mapsto R_E \cdot \max(0, l_m \cdot N_p) \cdot \max(0, r_m \cdot v). \end{aligned}$$

- **Reflexión difusa:** modela cómo se refleja la luz en objetos mates en función de la posición de la fuente de luz. Al contrario de lo que ocurre con la reflexión especular, debido a la irregularidad de la superficie del objeto la luz no se refleja en una sola dirección, haciendo que se disperse en direcciones impredecibles. Este comportamiento se simula a través de una radiancia R_D que represente el valor promedio resultado de estos reflejos, y que consideraremos constante. Al igual que antes, solo queremos que

2. Algoritmos de visualización de SDFs

el punto esté iluminado cuando esté de cara a la fuente de luz, obteniendo la mayor cantidad de luz cuando está alineado con la fuente. Así, la fracción de radiancia difusa será

$$f_{rd}: \mathbb{R}^3 \times \mathbb{R}^3 \times \mathbb{R}^3 \rightarrow \mathbb{R}^3,$$

$$(p, v, l_m) \mapsto R_D \cdot \max(0, l_m \cdot N_p).$$

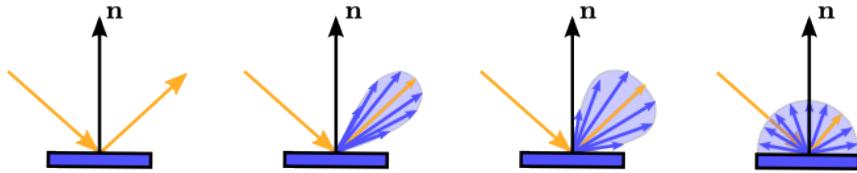


Figura 2.15.: Reflexión de un rayo en una superficie progresivamente más mate [22]

Observación 2.1. Es necesario que los vectores l_i , v y N_p sean unitarios, pues de lo contrario su producto escalar no coincidiría con el coseno del ángulo que forman.

En vista de las definiciones anteriores, podríamos simplemente definir

$$f_r(p, v, l_m) = f_{ra} + f_{re}(p, v, l_m) + f_{rd}(p, v, l_m).$$

De esta forma, para diferenciar entre un material totalmente mate como el yeso y uno espejular como el metal bastaría tomar valores de R_D y R_E tal que $\|R_D\| \gg \|R_E\|$. Sin embargo, a la hora de comparar materiales especulares podríamos observar que aunque ambos generen zonas brillantes no lo hagan de la misma forma. Por ejemplo, tanto el mármol como el metal generan brillos sobre su superficie, pero en el caso del metal estos son más pequeños y brillantes debido a que se trata de un material más pulido. Por tanto, para añadir control sobre el tamaño e intensidad de estas zonas brillantes introducimos el **coeficiente de brillo** $\alpha \in \mathbb{R}$ en la expresión de f_{re} , de forma que cuanto mayor sea su valor más pequeños e intensos serán los brillos generados. En la Figura 2.20 podemos ver el efecto que tienen R_E , R_D sobre la radiancia reflejada [22].

Definición 2.2. Dado un objeto, definimos su **material** como la tupla $\{R_A, R_E, R_D, \alpha\}$.

Una vez asociado un material a S_ϕ podemos escribir la expresión final para f_r :

$$\begin{aligned} f_r(p, v, l_i) &= f_{ra} + f_{re}(p, v, l_i) + f_{rd}(p, v, l_i) \\ &= R_A + R_E \cdot \max(0, l_i \cdot N_p) \cdot \max(0, r_i \cdot v)^\alpha + R_D \cdot \max(0, l_i \cdot N_p). \end{aligned} \quad (2.1)$$

Recogemos los resultados obtenidos en la siguiente definición.

Definición 2.3 (Modelo de Blinn). La radiancia percibida en el punto $p \in \mathbb{R}^3$ desde la

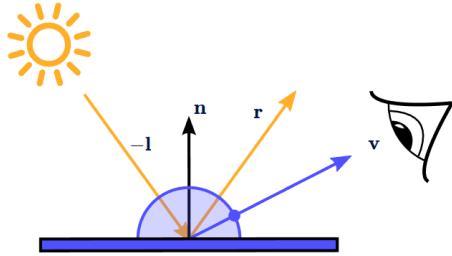
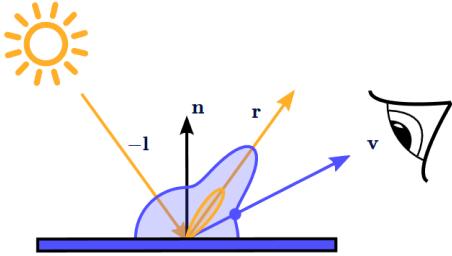
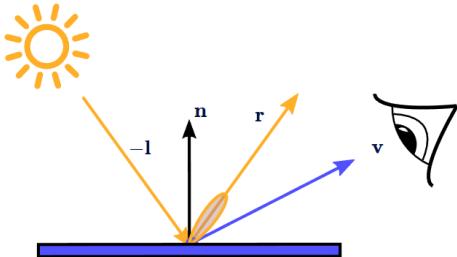
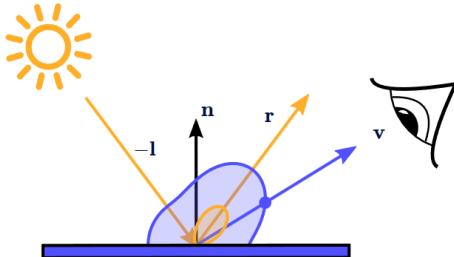

 Figura 2.16.: $\|R_E\| = 0$

 Figura 2.17.: $\|R_E\| = \|R_D\|$ y α grande

 Figura 2.18.: $\|R_D\| = 0$

 Figura 2.19.: $\|R_E\| = \|R_D\|$ y α pequeño

 Figura 2.20.: Ejemplo de distintos valores para R_E , R_D y α

dirección $v \in \mathbb{R}^3$ con $\|v\| = 1$ según el modelo de Blinn viene dada por

$$L(p, v) = L_A + L_E + \sum_{i=0}^n S_i \left[k_a R_A + \max(0, l_i \cdot N_p) \left(k_d R_D + k_e R_E \cdot \max(0, r_i \cdot v) \right) \right],$$

donde:

- $n \in \mathbb{N}$ es el número de fuentes de luz y $l_i \in \mathbb{R}^3$ es el vector normalizado que apunta a p desde cada una de ellas,
- $L_A, L_E \in \mathbb{R}^3$ son ternas RGB no acotadas representando la radiancia ambiente y emitida respectivamente,
- $S_i \in \mathbb{R}^3$ es una terna RGB no acotada representando la radiancia emitida por la fuente de luz i -ésima,
- $\alpha \in \mathbb{R}$ es el coeficiente de brillo,
- $R_A, R_D, R_E \in \mathbb{R}^3$ son ternas RGB (no acotadas) representando la radiancia reflejada de forma ambiental, difusa y especular respectivamente,
- N_p es el vector normal de la superficie en p y r_i es el vector l_i reflejado sobre N_p .

En 1975 Phong introdujo una variante a este modelo [23] que hoy conocemos como **modelo de Blinn-Phong**. Su única diferencia con el de Blinn consiste en el uso del llamado *halfway vector*

$$h_m = \frac{l_m + v}{\|l_m + v\|}.$$

2. Algoritmos de visualización de SDFs

Ahora, en lugar de usar el valor $r_m \cdot v$ hacemos que el brillo sea proporcional al coseno del ángulo entre h_m y N_p , de forma que no depende del punto p y solo necesita ser calculado una vez. En la Figura 2.21 podemos ver el comportamiento de h_m para distintas configuraciones de l_m y v .

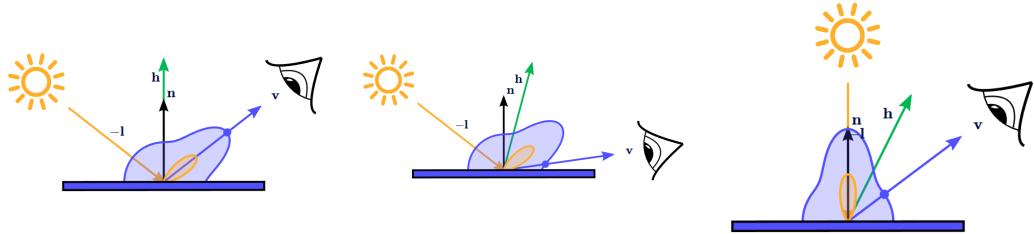


Figura 2.21.: Comportamiento de h_m con $\|R_S\| = \|R_D\|$

Aunque pueda parecer una simplificación del modelo de Blinn, lo cierto es que produce resultados más convincentes que este. En particular, mientras que el modelo de Blinn siempre produce brillos redondos en superficies planas, el de Blinn-Phong los genera con una forma más elíptica cuando se observa la superficie desde un ángulo acusado, como se observa en la Figura 2.24.

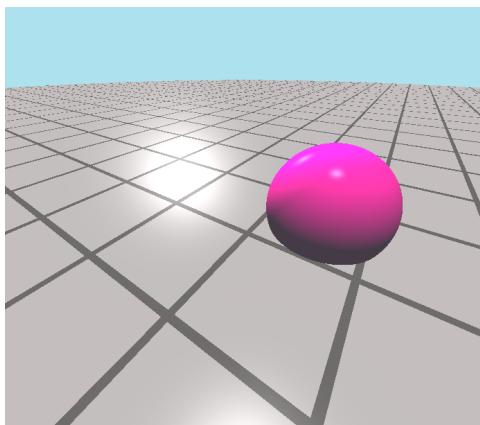


Figura 2.22.: Blinn

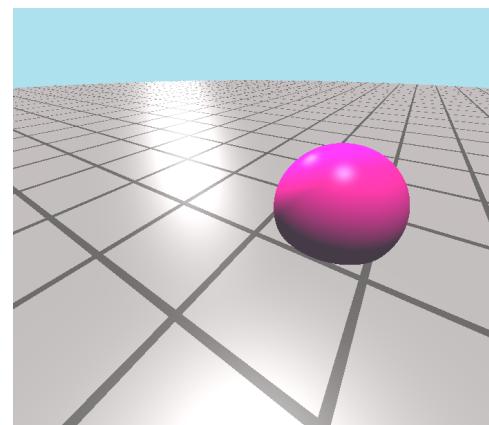


Figura 2.23.: Blinn-Phong

Figura 2.24.: Zonas brillantes en modelos de Blinn y Blinn-Phong

Definición 2.4 (Modelo de Blinn-Phong). En el contexto de la Def. 2.3, la radiancia percibida en el punto $p \in \mathbb{R}^3$ desde la dirección $v \in \mathbb{R}^3$ con $\|v\| = 1$ según el modelo de Blinn-Phong viene dada por

$$L(p, v) = L_A + L_E + \sum_{i=0}^n S_i \left[k_a R_A + \max(0, l_i \cdot N_p) \left(k_d R_D + k_e R_E \cdot \left(N_p \cdot \frac{l_i + v}{\|l_i + v\|} \right)^\alpha \right) \right],$$

Ya podemos darle forma a las funciones DibujarSuperficie y DibujarFondo usadas en la Figura 2.12, suponiendo que pasamos como *uniforms* los parámetros del material y los valores l_i y S_i para cada $i = 1, \dots, n$.

Algorithm 8: DibujarSuperficie

Data: punto p , dirección del rayo v , distancia $\phi(p)$
 $L \leftarrow L_A$ // Radiancia final
for $i \in \{1, \dots, n\}$ **do**
 $h \leftarrow \text{normalizar}(L_i - v)$ // Observador en dirección opuesta a la del rayo
 $N_p \leftarrow \text{CalcularNormal}(p)$
 $NLi \leftarrow \max(0, N_p \cdot l_i)$
 $NH \leftarrow \max(0, N_p \cdot h)$
 $f_{ra} = R_A$
 $f_{rd} = NLi \cdot R_D$
 $f_{re} = NLi \cdot R_E \cdot NH^\alpha$
 $L \leftarrow L + S_i \cdot (f_{ra} + f_{rd} + f_{re})$
end
return L

Algorithm 9: DibujarFondo

return L_A

Figura 2.25.: Implementación de las funciones DibujarSuperficie y DibujarFondo

Sólo queda un asunto por tratar. A la vista de la expresión de f_r (2.1) y del código anterior, somos capaces de calcular todos los valores a excepción de uno, el del vector normal.

2.2.1.1. Cálculo del vector normal

En cualquier modelo de iluminación el acceso al vector normal es indispensable. Cuando se trabaja con mallas de polígonos el vector normal viene dado para cada vértice, pero este no es nuestro caso. En su lugar nosotros usaremos el gradiente del SDF para obtenerlo.

Proposición 2.1. Sea $\phi: \mathbb{R}^3 \rightarrow \mathbb{R}$ diferenciable. Entonces $\nabla\phi$ es perpendicular a S_ϕ .

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

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

cumpliendo $\alpha(t_0) = s$ para algún $t_0 \in [0, 1]$. 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 contenido en el plano tangente de S_ϕ en s . Por tanto $\nabla\phi(s) \perp S_\phi$. \square

2. Algoritmos de visualización de SDFs

Hemos visto que calcular el vector normal en cualquier punto equivale a calcular $\nabla\phi$ y que este existe en casi todo punto de S_ϕ por el [Teorema 1.1](#), pero esto no significa que podamos o debamos obtenerlo de forma analítica. Si bien en muchos casos sería posible hacerlo de forma analítica, esto podría tener asociado un coste computacional que no podemos asumir. Existen varios métodos numéricos para aproximar el gradiente de una función. Uno de los más triviales es el de las diferencias centrales, basado en aproximar el límite de la [Def. 1.4](#) tomando un valor pequeño para h . Necesitaríamos entonces realizar seis evaluaciones de ϕ para obtener el gradiente, dos por cada parcial. Nosotros usaremos el **método del tetraedro** [24], que sin ser el más preciso, produce buenos resultados y es rápido, haciendo uso únicamente de cuatro evaluaciones de 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 2.2 (Método del tetraedro). *Dado $p \in S_\phi$, una aproximación de su vector normal N_p se obtiene normalizando el vector*

$$\hat{N}_p = \sum_{i=0}^3 k_i \cdot f(p + hk_i) \quad , \text{ donde } h \approx 0.$$

Demostración. Por la proposición [Proposición 2.1](#), basta comprobar que \hat{N} es colineal a $\nabla\phi(p)$.

$$\begin{aligned} \hat{N} &= \sum_{i=0}^3 k_i \cdot f(p + hk_i) = \sum_{i=0}^3 k_i \cdot f(p + hk_i) - k_i \cdot f(p) = \sum_{i=0}^3 k_i \cdot [f(p + hk_i) - f(p)] \\ &= h \sum_{i=0}^3 k_i \nabla_{k_i} f(x) = h \sum_{i=0}^3 k_i \cdot (k_i \cdot \nabla f(p)) = h \sum_{i=0}^3 (k_i \cdot k_i) \nabla f(p) = h \sum_{i=0}^3 \nabla f(p) = 4h \nabla f(p) \end{aligned}$$

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. \square

2.2.2. Sombras

Los resultados obtenidos en la [Figura 2.24](#) presentan ciertas carencias, siendo la más flagrante la ausencia de sombras arrojadas, que no son consideradas en el modelo de Blinn-Phong. Afortunadamente el uso de los SDF nos hará la obtención de la información necesaria para añadir sombras a nuestra escena muy sencilla. Para saber si un punto $p \in \mathbb{R}^3$ recibe luz de una i -ésima fuente de luz bastará comprobar si hay algún obstáculo entre dicha fuente y el punto. Para hacer esta comprobación usaremos de nuevo *spheretracing*, pero en esta ocasión desde el punto hacia la fuente de luz. Si se detecta alguna intersección significará que el punto p no recibe luz de la fuente y por tanto $L_R(p, v, l_i) = 0$, $\forall v \in \mathbb{R}^3$. Podemos modificar *DibujarSuperficie* como se muestra en la [Figura 2.26](#) para añadir este comprobación.

Algorithm 10: DibujarSupercicie

Data: punto p , dirección del rayo v , distancia $\phi(p)$
 $L \leftarrow L_A + L_E$ // Radiancia final
for $i \in \{1, \dots, n\}$ **do**
 // ...
 $sombras \leftarrow CalcularSombras(p, l_i)$
 $L \leftarrow L + S_i \cdot (f_{ra} + f_{rd} + f_{re}) \cdot sombras$
end
return L

Algorithm 11: CalcularSombras

Data: punto p_0 , dirección de luz l_i
Result: $sombra \in [0, 1]$
 $d \leftarrow \delta$ // distancia actual
for $i \in MAX_ITERACIONES$ **do**
 $p \leftarrow p_0 + d \cdot v$
 $sdf \leftarrow \phi(p)$
 if $sdf < \varepsilon$ **then**
 | **return** 0;
 end
 $d \leftarrow d + sdf$
 if $d > MAX_DISTANCIA$ **then**
 | **return** 1
 end
end
return 1

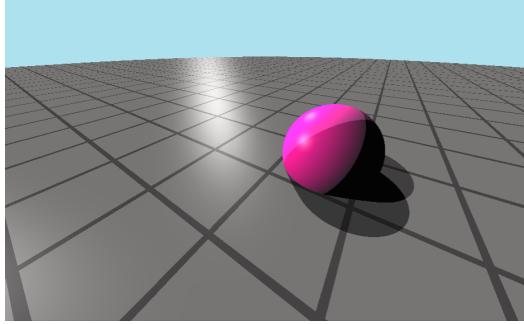


Figura 2.26.: Cálculo básico de sombras

Realizamos las siguientes apreciaciones respecto al método `CalcularSombras` propuesto:

- Dado que estamos trabajando con luces direccionales situadas a distancia infinita solo podemos hacer *spheretracing* desde p en dirección a la fuente, a pesar de que lo intuitivo sería hacerlo desde el foco de luz hacia el punto,
- A diferencia del algoritmo propuesto en [Figura 2.12](#) no podemos inicializar $d = 0$, ya que entonces se detectaría una intersección en el mismo punto p .

Estudiando los resultados obtenidos vemos que al añadir sombras obtenemos una imagen mucho más cohesiva y otorgamos a la esfera mayor presencia en la escena. Sin embargo también podremos apreciar que las sombras que genera este método son muy planas y duras. Realmente ahora mismo no tenemos control alguno sobre esto, ya que según nuestra implementación un punto o está totalmente en sombra o totalmente iluminado. Esto no siempre es así en el mundo real, donde podemos encontrar que no toda la región sombreada sea igual de oscura o el borde esté más o menos difuminado en función de las propiedades de la fuente. Podemos simular estos fenómenos de forma muy sencilla usando información de la que ya que disponemos en el algoritmo de *spheretracing*. En particular,

- Haremos que cuanto más cerca se encuentre el punto del obstáculo que le hace sombra menos luz reciba. Por tanto la intensidad de la sombra será proporcional a la evaluación

2. Algoritmos de visualización de SDFs

del SDF en el punto actual del rayo:

$$sombra \propto sdf,$$

- Cuando un punto sea alcanzado por la luz pero haya estado muy cerca de ser obstruido dejaremos que refleje solo una fracción de la luz total. Esta cantidad deberá ser mayor cuanto menos haya faltado para perder la intersección, creando un difuminado en el borde. Por tanto

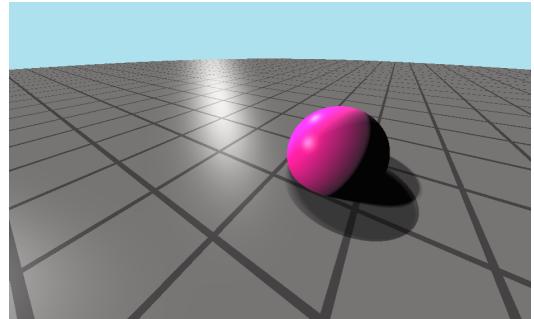
$$sombra \propto \frac{1}{d}.$$

Esta nueva versión de CalcularSombras se encuentra descrita en la [Figura 2.27](#). En ella se ha añadido un parámetro $k \in \mathbb{R}_0^+$ para controlar la intensidad del efecto de suavizado. En realidad este parámetro hace referencia al **tamaño de la fuente de luz**, en concreto a su inversa. Así, cuanto más pequeño sea este valor más grande será la fuente de luz, produciendo sombras más difusas. Por ejemplo, el Sol tendrá un valor pequeño para k , mientras que una bombilla lo tendría grande. A partir de ahora en nuestra escena de ejemplo fijamos $k = 1.5$ para la fuente de luz que apunta más hacia la cámara, que actuará como el Sol, y $k = 10$ para la otra, que actuará como una linterna.

Algorithm 12: CalcularSombras

```

Data: punto  $p_0$ , dirección de luz  $l_i$ ,
       tamaño de luz  $k$ 
Result:  $sombra \in [0, 1]$ 
 $sombra \leftarrow 1$ 
 $d \leftarrow \delta //$  distancia actual
for  $i \in \text{MAX\_ITERACIONES}$  do
     $p \leftarrow p_0 + d \cdot v$ 
     $sdf \leftarrow \phi(p)$ 
    if  $sdf < \varepsilon$  then
         $\mid$  return  $o$ ;
    end
     $sombra \leftarrow \min(sombra, k \cdot \frac{sdf}{d})$ 
     $d \leftarrow d + sdf$ 
    if  $d > \text{MAX\_DISTANCIA}$  then
         $\mid$  return  $1$ 
    end
end
return  $1$ 
```



[Figura 2.27](#):: Cálculo de sombras suavizadas

Si bien este método genera resultados más realistas en general, también puede generar ciertas imperfecciones en el borde de la sombra, como se puede apreciar en la [Figura 2.28](#). Esto es debido a que en el proceso de *spheretracing* podemos saltarnos una intersección que habría aportado más oscuridad que la que finalmente se ha encontrado, generando fugas de luz que siguen el patrón de los puntos en los que se evalúa el SDF. Hay varias formas de solventar esto, como la propuesta por Sebastian Aaltonen [\[25\]](#) en la GDC de 2018. Su idea se basa en comprobar intersecciones también en los puntos que se estiman como los más

cercanos a la superficie en cada iteración. Nosotros usaremos una técnica introducida por el usuario nurof3n [26] en Shadertoy y estudiada posteriormente por Íñigo Quílez.

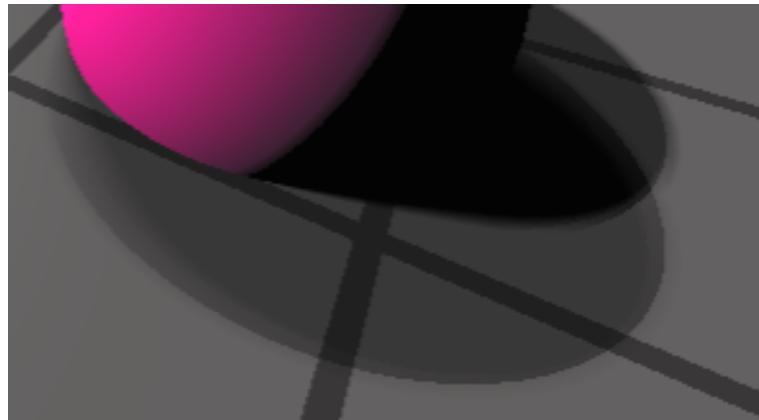


Figura 2.28.: Detalle de las fugas de luz al calcular sombras

La diferencia con nuestro método actual radica en que se permite que el rayo penetre un poco la superficie para detectar los puntos que casi no son alcanzados por un rayo de luz. Por tanto, ahora para cada punto se tiene en cuenta si casi ha sido alcanzado y si casi no ha sido alcanzado por un rayo de luz. Para permitir que el rayo entre en la geometría basta con modificar la condición de ruptura sobre sdf a un número negativo, con la precaución de siempre sumar una cantidad positiva a d , pues de lo contrario el trazado del rayo retrocedería. Fijando este valor a -1 la variable *sombra* tendrá un valor en el rango $[-1, 1]$ al salir del bucle, pero aún queremos obtener un valor entre $[0, 1]$ para representar la cantidad de luz que recibe el punto. Para remapear *sombra* a este rango podemos usar la función `smoothstep(a, b, x)` de GLSL, que interpola x suavemente entre 0 y 1 en relación con los límites a y b . En particular la interpolación que se lleva a cabo es la de Hermite, haciendo que la transición entre distintos puntos de sombra no sea lineal y parezca más natural. Podemos ver el algoritmo final en la Figura 2.30 y los resultados que consigue en la Figura 2.38.

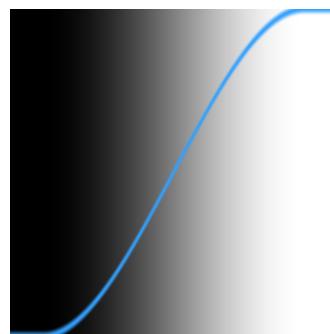


Figura 2.29.: Visualización de `smoothstep` [27]

2. Algoritmos de visualización de SDFs

Algorithm 13: CalcularSombras

Data: punto p_0 , dirección de luz l_i , tamaño de luz k
Result: Valor entre 0 y 1 representando la cantidad de sombra recibida en p

```

sombra ← 1
d ← δ // distancia actual
for  $i \in \text{MAX\_ITERACIONES}$  do
    |  $p \leftarrow p_0 + d \cdot v$ 
    | sombra ←  $\min(res, k \cdot \frac{sdf}{d})$ 
    |  $sdf \leftarrow \phi(p)$ 
    |  $d \leftarrow d + |sdf|$ 
    | if  $sombra < -1 \text{ || } d > \text{MAX\_DISTANCIA}$  then
    |   | break
    | end
end
return  $\text{smoothstep}(-1, 1, sombra)$ 

```

Figura 2.30.: Cálculo de sombras suavizadas mejorado

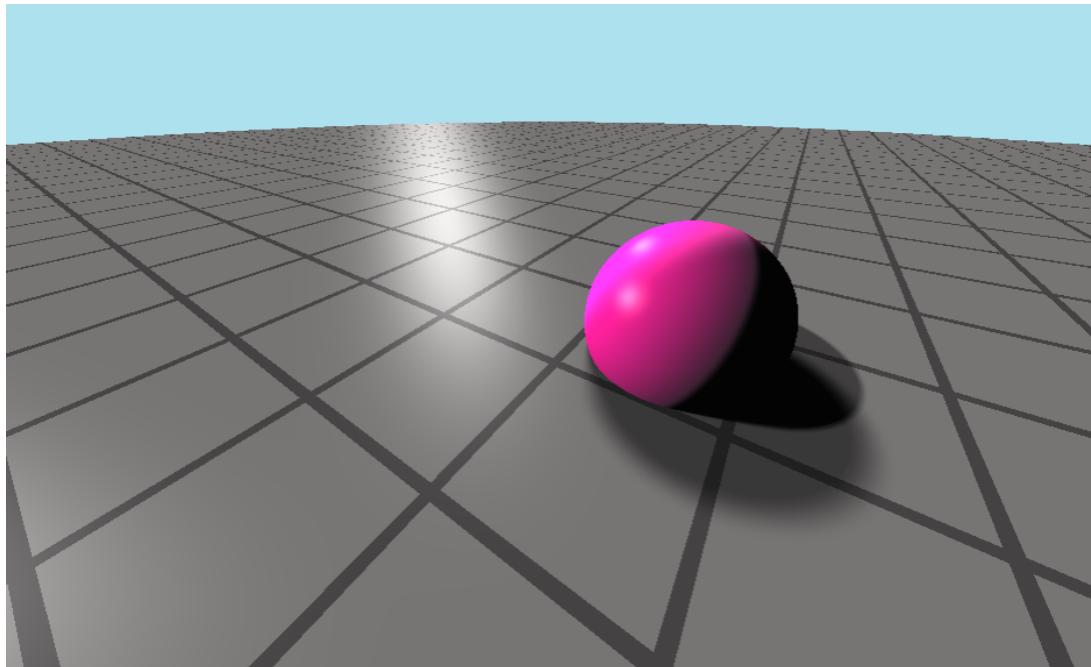


Figura 2.31.: Resultado final del cálculo de sombras

2.2.3. Oclusión ambiental

Al añadir sombras los objetos están mucho más integrados en la escena, pero todavía podemos conseguir un mayor grado de cohesión. En el estado actual de la escena aún hay puntos que no es convincente que reciban luz pero se encuentran totalmente iluminados. Un ejemplo son los puntos de intersección entre la esfera y el suelo. Uno esperaría que poca luz fuera capaz de alcanzar un espacio tan cóncavo, pues la propia geometría de la esfera y el suelo

ocluirían la luz. Este fenómeno recibe el nombre de **occlusión ambiental**, y de nuevo gracias a los SDF nos resultará muy fácil y computacionalmente barato simularlo.

Cuando se trabaja con geometría de polígonos, una de las técnicas más comunes es la occlusión ambiental del espacio de pantalla, o SSAO por sus siglas en inglés. En su versión más básica esta solución usa la información del fotograma actual para consultar por cada píxel el buffer de profundidad o *depth buffer* de los píxeles cercanos. Con esta información realiza una aproximación de las características de la geometría en ese entorno y deduce la cantidad de luz que debería poder pasar. El principal problema de esta y otras técnicas basadas en el espacio de pantalla es que al no usar la información real de la geometría, los resultados obtenidos varían según la orientación de la cámara, posición relativa de los objetos en pantalla, etc. Otro método basado en espacio de pantalla que pone de manifiesto este problema es el de los reflejos de espacio de pantalla o SSR, que suele ser usado para simular reflejos como los del agua o espejos en videojuegos. Al usar el mismo principio que SSAO, solo podrá reflejar correctamente los píxeles que estén dibujados en pantalla. Esta limitación hace que cuando un objeto oculta a otro este no se pueda reflejar correctamente y se generen reflejos erróneos como se muestra en la [Figura 2.34](#), o que si un objeto no aparece en pantalla directamente no sea reflejado, como se representa en la [Figura 2.37](#).



Figura 2.32.: SSR



Figura 2.33.: Raytracing

Figura 2.34.: Videojuego Ratchet & Clank: Una dimensión aparte usando SSR y raytracing
[[28](#)]

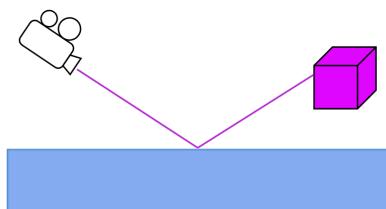


Figura 2.35.: Reflejo detectado

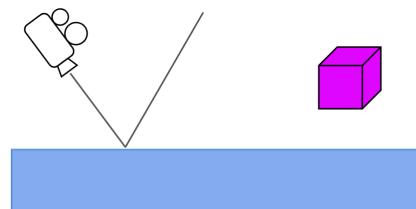


Figura 2.36.: Reflejo no detectado

Figura 2.37.: Reflejos en agua con SSR

2. Algoritmos de visualización de SDFs

La solución a estos problemas cuando se trabaja con vértices es el uso de técnicas más avanzadas y computacionalmente costosas como el *raytracing*. La buena noticia es que al estar usando SDFs nosotros podremos usar la información real de la geometría de nuestra escena. Obtendremos por tanto información más precisa, y además de forma muy barata, ya que requeriremos de muchas menos evaluaciones del SDF que el algoritmo de *spheretracing*. La técnica que vamos a usar fue ideada por Alex Evans en 2006 [29], y se conoce como **occlusión ambiental muestreada por la normal**.

El método se basa en dado un punto $p \in S_\phi$ evaluar el SDF en varios puntos del vector normal N_p a distancias d_i de p para obtener la información de la geometría cercana. Si en el entorno de p hay geometría que le esté obstruyendo la llegada de luz, en alguna de estas evaluaciones se obtendrá un valor menor que d_i , mientras que de lo contrario uno esperaría que

$$\phi(p + d_i N_p) = d_i,$$

ya que eso significaría que el punto más cercano a p de S_ϕ es el propio p . Así, si hacemos M evaluaciones igualmente espaciadas a lo largo de N_p , consideraremos que el punto p no está ocluido si

$$\sum_{i=1}^M \phi\left(p + \frac{i}{M} N_p\right) - \sum_{i=1}^M \frac{i}{M} = 0.$$

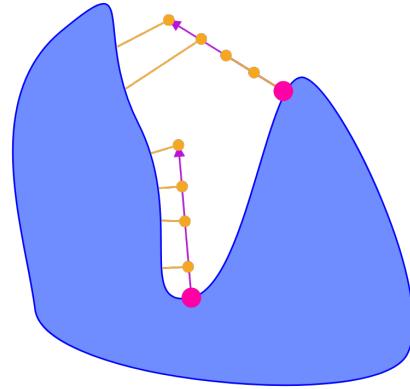


Figura 2.38.: Cálculo de oclusión ambiental muestreada por la normal

Cuanto mayor sea este valor (no puede ser menor que 0 por definición de SDF) menos luz será capaz de alcanzar p . Así, podemos representar la cantidad de luz ocluida como

$$\sum_{i=1}^M \frac{1}{2^i} \cdot \left(\frac{i}{M} - \phi\left(p + \frac{i}{M} N_p\right) \right) \in [0, 1].$$

El hecho de que este valor esté acotado en $[0, 1]$ viene de que suponemos que $\|N_p\| = 1$, de forma que

$$\phi\left(p + \frac{i}{M} N_p\right) \leq 1,$$

ya que siempre habrá algún punto a lo largo de N_p que esté a una unidad o menos de

Algorithm 14: DibujarSupercicie

Data: punto p , dirección del rayo v , distancia $\phi(p)$
 $L \leftarrow L_A + L_E$ // Radiancia final
for $i \in \{1, \dots, n\}$ **do**
 // ...
 $N_p \leftarrow CalcularNormal(p)$
 $sombras \leftarrow CalcularSombras(p, l_i)$
 $ao \leftarrow CalcularAO(p, N_p)$
 $L \leftarrow L + S_i \cdot (f_{ra} + f_{rd} + f_{re}) \cdot sombras \cdot ao$
end
return L

Algorithm 15: CalcularAO

Data: punto p , vector normal N_p
Result: $ao \in [0, 1]$
 $ao \leftarrow 1$
 $increment \leftarrow 1/M$
 $i \leftarrow increment$
while $i < 1$ **do**
 $sdf \leftarrow \phi(p + iN_p)$
 $ao \leftarrow ao - 2^{-iM} \cdot (i - sdf)$
 $i \leftarrow i + increment$
end
return ao

Figura 2.39.: Cálculo de oclusión ambiental

distancia de p : él mismo. Por otro lado, hemos usado la exponencial para dar más peso sobre el resultado final a aquellos puntos más cercanos a p .

Con esto ya podemos obtener la nueva versión del método DibujarSuperficie que tiene en cuenta la oclusión ambiental descrita en la Figura 2.39. En ella hemos introducido una pequeña optimización [30] cambiando el índice del bucle para no tener que calcular una división en cada iteración. Otra posible optimización sería sustituir la potencia por un flotante que fuéramos multiplicando por un factor menor que 1 en cada iteración. Finalmente, podemos apreciar los resultados obtenidos en la Figura 2.42, donde para valores tan pequeños de M como 2 o 4 ya conseguimos resultados más que convincentes.

2.3. Antialiasing

Vamos a introducir una última mejora en la forma en la que generamos la imagen. Un defecto típico en computación gráfica es el conocido como *aliasing*. Este se caracteriza por la presencia de dientes de sierra en líneas curvas o diagonales, y en nuestro caso se puede apreciar muy fácilmente en los bordes del cubo y en la cuadrícula del suelo en la distancia (Figura 2.46a). En el mercado actual existen multitud de alternativas como solución a este problema. Algunos ejemplos son FXAA, basado en espacio de pantalla, o SSAA y MSAA, que usan la técnica del *supermuestreo* o *supersampling* junto con filtros de suavizado. Nosotros

2. Algoritmos de visualización de SDFs

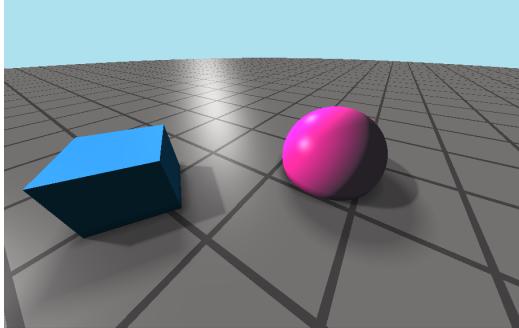


Figura 2.40.: $M = 2$

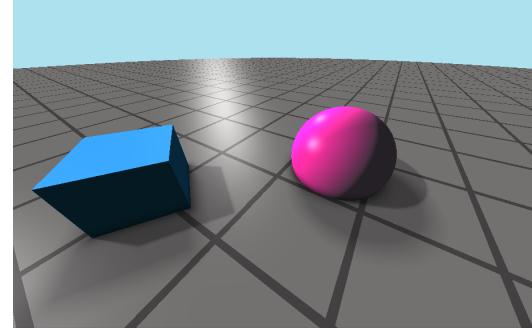


Figura 2.41.: $M = 4$

Figura 2.42.: Resultado del cálculo de oclusión ambiental

implementaremos una versión de SSAA (*Supersampling Anti-Aliasing*), pero antes debemos entender por qué aparece el problema del *aliasing* en primer lugar.

Toda pantalla tiene resolución finita, y por tanto la definición con la que puede mostrar la información es limitada. Al realizar la proyección sobre la pantalla puede ocurrir que una primitiva no ocupe un píxel completo, y como cada píxel solo puede mostrar un único color hay que elegir algún criterio para determinar qué hacer en esos casos. El más común es considerar que el píxel pertenece a la primitiva si su proyección cubre el centro del píxel. En nuestro caso esto se traducía en trazar el rayo a través del centro del píxel en el algoritmo de *spheretracing*. Al hacer esta aproximación es cuando aparecen los dientes de sierra, pues a no ser que se trate de una línea totalmente vertical u horizontal, es como si intentásemos construir una rampa con escalones.

Lo cierto es que no podemos hacer desaparecer este problema, pues es algo intrínseco de la naturaleza discreta de las pantallas y los sistemas de muestreo. En nuestro caso esto último se traduce en que no podemos trazar infinitos rayos. No obstante, lo que sí podemos hacer es tratar de disimularlo. En lugar de tomar una decisión binaria de si un píxel debe ser de un color u otro podemos intentar tener en cuenta la aportación de otras primitivas que estén cercanas dentro del píxel aunque no ocupen su centro. Una primera idea podría ser que una vez asignado un color a un píxel se hiciera la media con sus píxeles vecinos para así generar una transición suave entre ellos. Sin embargo este acercamiento presenta dos grandes inconvenientes:

- Estaríamos perdiendo parte de la información original, y por ende, haciendo la imagen más borrosa.
- El responsable de asignar el color de cada píxel es una instancia del *fragment shader*, y como ya comentamos, los *shaders* son programas independientes y no tienen información sobre el resto de instancias. Por tanto este método sería de postprocesado, es decir, sería ejecutado una vez hubiera sido generada la imagen,

De esto podemos sacar la conclusión de que la solución debe ser local a cada píxel, y de ser posible que no conlleve la pérdida de información. La opción de hacer una media entre varias muestras de píxeles sigue pareciendo razonable, lo que nos lleva a la idea detrás de

SSAA: tomar más muestras dentro de cada píxel. Para ello, tendremos que trazar rayos por más puntos dentro del píxel, esto es, dibujar la imagen con mayor resolución, de donde viene el nombre de supermuestreo. El patrón en el que tomamos las nuevas muestras es de nuestra elección. En la Figura 2.43 se muestran algunos patrones comunes, de los cuales optaremos por el uniforme por su sencillez y porque proporciona resultados bastante buenos en general. El número de evaluaciones también está a nuestra elección, pero no hay que olvidar el factor del rendimiento, ya que usando este patrón el número de rayos crece exponencialmente por cada nuevo nivel adicional de precisión.

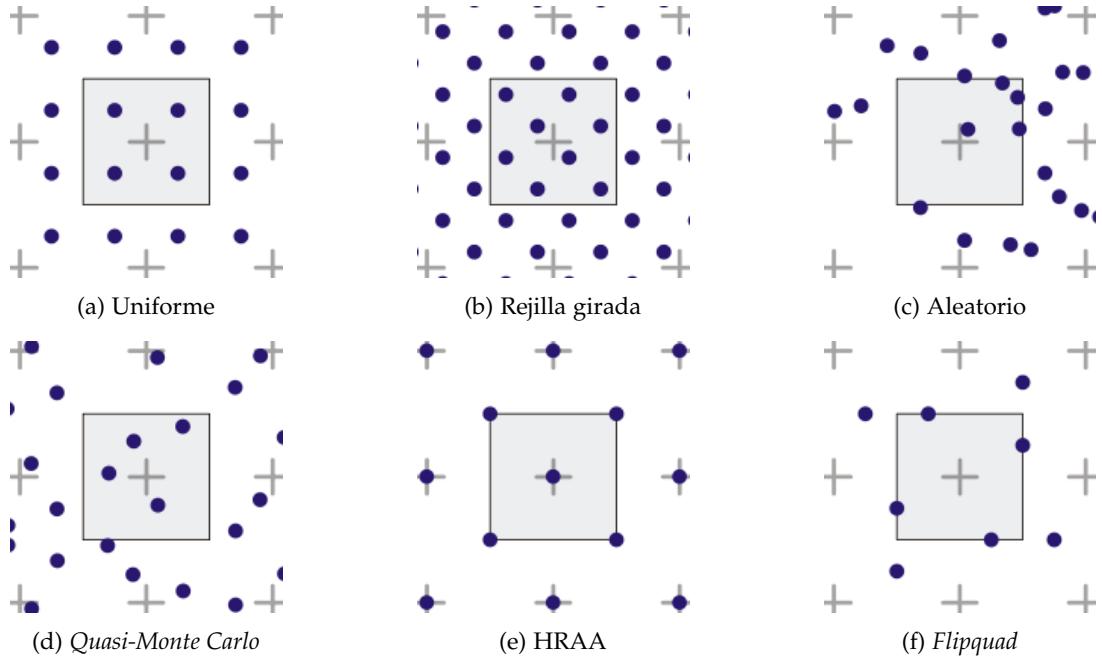


Figura 2.43.: Patrones de supermuestreo [31]

Modificar por dónde pasan los rayos dentro de cada píxel equivale a cambiar cómo calculamos los uv. Llamaremos a partir de ahora AA al factor de escalado de la imagen, de forma que por cada píxel haremos pasar 4^{AA} rayos. Para hallar los nuevos puntos de muestra bastará con subdividir el píxel en AA^2 cuadrantes y quedarnos con el centro de cada uno. Si recordamos que `gl_FragCoord` devuelve las coordenadas del centro del píxel, que el ancho y alto del píxel es una unidad, y que tenemos que hacer AA subdivisiones en cada eje, es evidente que podemos obtener los nuevos puntos de muestra desplazando el origen usual del píxel la cantidad

$$offset_{m,n} = (m + 1/2, n + 1/2) \cdot subdivision - \left(\frac{lado}{2}, \frac{lado}{2} \right) = \frac{(m + 1/2, n + 1/2)}{AA} - \left(\frac{1}{2}, \frac{1}{2} \right),$$

donde $m, n \in \{0, \dots, AA - 1\}$.

Para trasladar esto a nuestro *fragment shader* tan solo habrá que realizar un doble bucle e ir sumando el color obtenido por *spheretracing* en una variable que luego ponderaremos por el número total de muestras, AA^2 . La nueva versión del *fragment shader* se describe en la

2. Algoritmos de visualización de SDFs

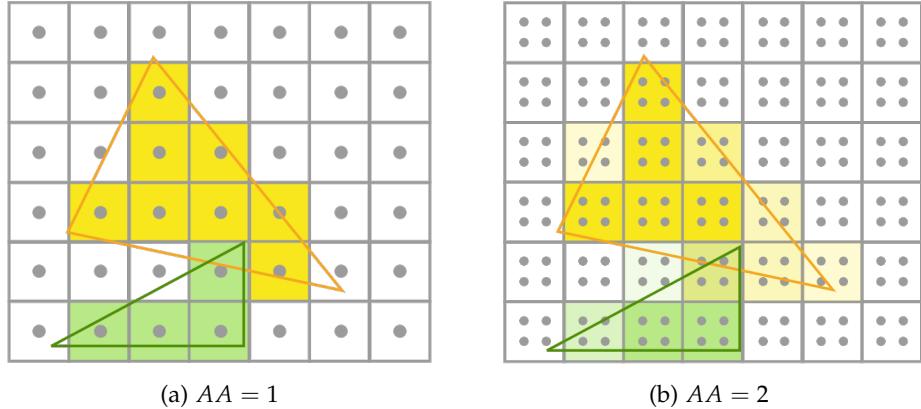


Figura 2.44.: Antialiasing para diferentes valores de AA

Figura 2.45.

Algorithm 16: Fragment Shader

Data: posición del observador c_0 , punto de atención l
Result: Terna RGBA con el color asignado al píxel actual
 $color \leftarrow (0, 0, 0)$
for $m \in \{0, \dots, AA - 1\}$ **do**
 for $n \in \{0, \dots, AA - 1\}$ **do**
 $offset \leftarrow \frac{(m, n)}{AA} - (0.25, 0.25)$
 $uv \leftarrow 2 \cdot \frac{(gl_FragCoord.xy + offset) - 0.5 \cdot u_resolution.xy}{u_resolution.y}$
 $r_d \leftarrow (f_1 | f_2 | f_3) \cdot \text{normalizar}((uv.xy, -1))$
 $color \leftarrow color + \text{spheretracing}(c_0, r_d)$
 end
end
 $color \leftarrow color / AA^2$
 $gl_FragColor \leftarrow (color, 1)$

\backslash	0	1	2
0	•	•	•
1	•	•	•
2	•	•	•

Figura 2.45.: Cuerpo del método `main` del *fragment shader*

Es evidente que $AA = 1$ equivale a no aplicar *antialiasing*, pero tendrá el efecto de desplazar la imagen medio píxel hacia abajo y a la izquierda, pues el único de cada píxel pasará por su esquina inferior. Por tanto, aunque no sea totalmente necesario, se puede añadir una comprobación para calcular los uv como hacíamos originalmente en este caso. En la [Figura 2.46](#) podemos ver que la pérdida de rendimiento no es en vano y obtenemos una imagen mucho más suave que la original. Sin embargo, no conviene tomar un valor de AA mayor que 3, pues generará mucha sobrecarga y la mejora no es muy apreciable. Finalmente y para concluir la sección, en el [Apéndice B](#) podemos ver la construcción que hemos hecho de la escena paso a paso, viendo el efecto que ha tenido en el aspecto final cada técnica que hemos ido añadiendo.

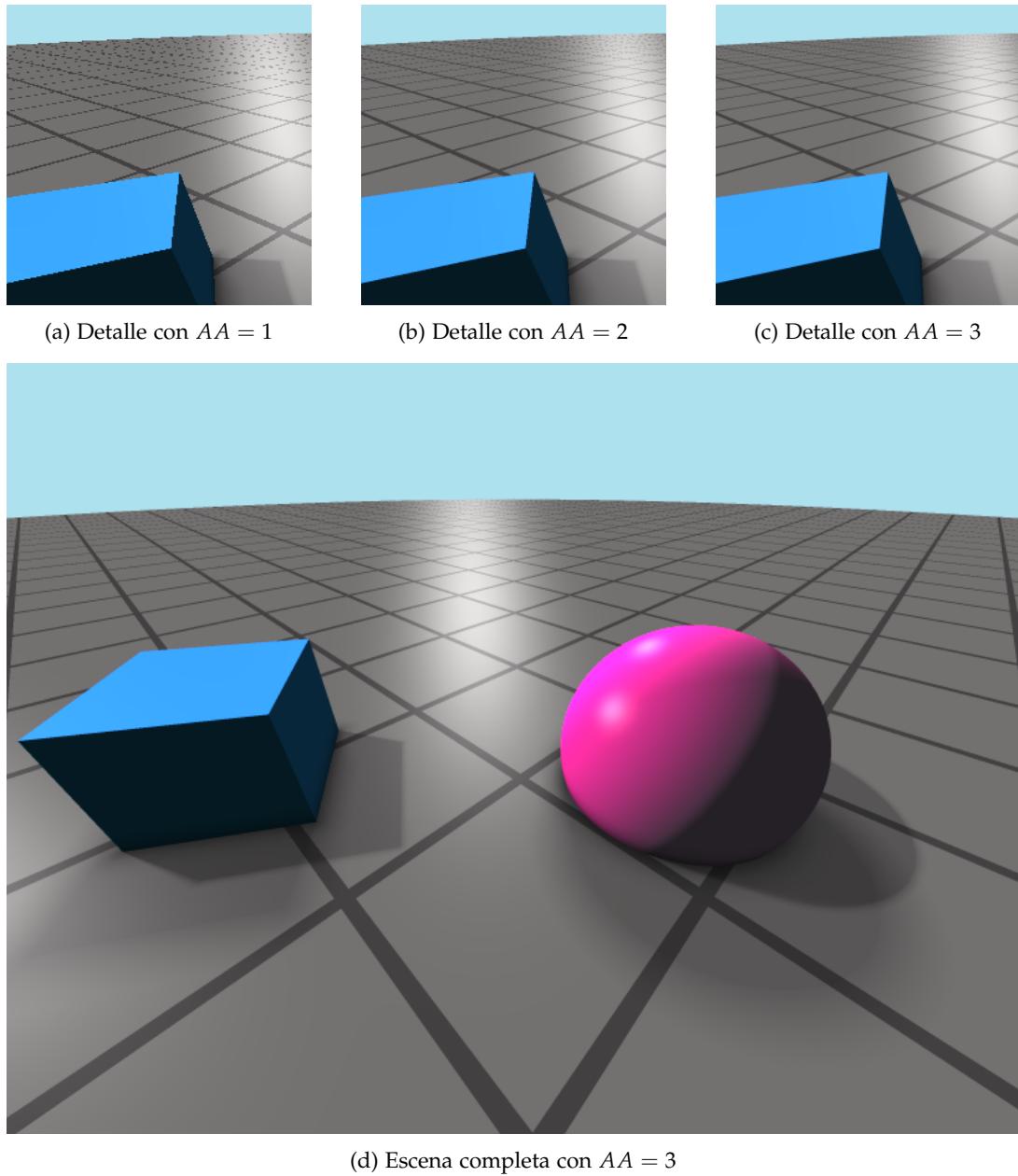


Figura 2.46.: Resultados de añadir *antialiasing*

3. Desarrollo e implementación

En esta sección veremos cómo se han usado las técnicas y conceptos presentados para la realización de una aplicación web que permita al usuario crear e interactuar superficies a través de SDFs, ecuaciones implícitas y paramétricas. El motivo de desarrollar una aplicación web es que sea accesible al mayor número de usuarios y de la forma más cómoda posible. Se ha decidido usar **React** para esta tarea, una biblioteca de JavaScript (y TypeScript) para interfaces de usuario. Es una librería muy popular, y por tanto muy bien documentada y con muchos paquetes de la comunidad disponibles. Las principales características de React son:

- Utiliza la **extensión de sintaxis JSX**, la cual permite escribir código JavaScript como si se tratase de HTML o XML. Se pueden usar expresiones JSX dentro de bucles `for` o entornos condicionales `if`, y dentro de ellas se pueden agregar expresiones JS entre corchetes.
- Se basa en **componentes autocontenidos y reutilizables**. La forma más común actualmente de declarar componentes es a través de funciones que reciben argumentos, o `props`, y devuelven una expresión JSX. Todos los componentes reciben el parámetro `children` por defecto, contenido la expresión JSX de los componentes que se encuentren entre las etiquetas de apertura y cierre del componente. El flujo de datos es unidireccional del componente padre a sus hijos.
- Utiliza un **DOM virtual** para solo actualizar los componentes cuyo estado o `props` han cambiado. En componentes funcionales, la manera de indicar variables que desencadenan un re-renderizado al ser modificadas es a través de `hooks`. En general, estos son funciones de JS que permiten crear y acceder al estado y ciclos de vida de React. Los principales son `useState`, usado para declarar una variable junto con su *setter*, y `useEffect`, que permite ejecutar código cuando se actualice el componente. Si solo se quiere reaccionar a cambios de ciertos `hooks` se puede indicar en las dependencias.

Un ejemplo de uso básico de JSX, componentes funcionales y manejo de estado sería el siguiente:

```
function Tarjeta(props) {
  return (
    <div>
      {props.children}
      {props.nombre}
    </div>
  );
}

function Main() {
  const [miNombre, setMiNombre] = React.useState("Daniel");
```

3. Desarrollo e implementación

```
useEffect(()=>{
  console.log("Solo me ejecuto una vez al inicio");
}, []);

useEffect(()=>{
  console.log("Has cambiado el nombre");
}, [miNombre]);

return (
<TarjetaNombre nombre={miNombre}>
  <h1>Hola, mi nombre es</h1>
</TarjetaNombre>
);
}
```

Las principales ventajas que aporta son:

- La aplicación puede ser ejecutada en cualquier navegador, haciendo que sea mucho más accesible,
- 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.

La aplicación consta de tres componentes principales. Dos de ellos son con los que interactúa el usuario, uno en la que se le permite crear primitivas introduciendo directamente una SDF, ecuaciones implícitas o paramétricas, y otro que contiene un editor de nodos en forma de árbol para aplicar operaciones sobre las primitivas creadas y guardar los resultados obtenidos. El último componente actúa como gestor de almacenamiento y estado de la aplicación. A continuación estudiamos cada componente por separado describiendo los subcomponentes que la conforman y cómo estos interaccionan entre sí.

3.1. Editor de nodos

Este componente se basa en [React Flow](#), un paquete muy completo que permite la implementación de diagramas interactivos basados en nodos. Cada nodo tendrá cierto número de puertos de entrada (solo permite la conexión con un nodo) y uno de salida (permite conectarse a varios nodos). Se nos permite declarar tipos de nodos según nuestras necesidades. Nosotros usaremos dos categorías principales de nodos.

El **nodo de primitiva** es el más sencillo, y permite seleccionar una primitiva entre las guardadas para ser conectada a uno o varios nodos de operaciones.

Los **nodos de operaciones** implementan las operaciones explicadas en la [Sección 1.3](#) y las aplican a las primitivas que reciben por sus puertos de entrada. A su vez hay cuatro tipos diferentes de estos nodos, uno por cada tipo de operación. Los nodos de operaciones de transformación, deformación y repetición tienen un único puerto de entrada, pues son operadores unarios. El nodo booleano sin embargo es capaz de recibir un número arbitrario de primitivas, ya que aunque los operadores booleanos son binarios, si se quiere realizar una

misma operación de forma reiterada sobre varias primitivas puede ser muy tedioso. Así, si un nodo booleano recibe las primitivas A_i con $i = 1, \dots, n$, irá aplicando la operación de forma sucesiva sobre el resultado anterior según el orden de conexión. Por ejemplo, para la unión tendríamos

$$\bigcup_{i=0}^n A_i = A_n \bigcup (A_{n-1} \bigcup (\dots A_2 \bigcup A_1)).$$

La estructura de todos los nodos es similar. Todos cuentan con un encabezado que muestra de qué tipo son, un desplegable para elegir la primitiva u operación a usar seguido de un área con controles para los parámetros que pueda tener la primitiva y operación, un lienzo para mostrar el resultado de las operaciones aplicadas hasta el momento y un botón para contraer el nodo ocultando toda la información excepto el encabezado. Debido a esto tiene sentido tener un componente nodo general que se pueda adaptar a diferentes tipos de uso. El encabezado y elementos del desplegable se pasan fácilmente a través de props. Sin embargo para los parámetros sí que depende fuertemente del tipo de nodo en particular, y serán implementados por cada tipo de nodo por separado y pasado al general a través del parámetro `children`.

Para gestionar el estado del editor de nodos usamos de nuevo Zustand, que principalmente contendrá la información de los nodos, las conexiones entre ellos, y varias funciones para gestionarlos (añadir, eliminar, actualizar, etc.). En particular, la información de cada nodo incluye su SDF, de forma que cuando un nodo detecta una nueva conexión en algún puerto de entrada se leen las SDFs de los nodos conectados y junto con los propios parámetros del nodo se actualiza la SDF del nodo. Cuando se elimina alguna conexión en el caso de los nodos diferentes al booleano simplemente la SDF pasa a ser indefinida, ya que solo tienen una entrada. En el caso del booleano habrá que tener en cuenta si todavía queda alguna entrada, reorganizar las restantes para que no haya puertos vacíos distintos al último, y reducir el número de puertos a uno menos. Para esto, se detecta la posición del puerto que se ha eliminado y se modifican las conexiones siguientes para cambiar su puerto al inmediatamente anterior.

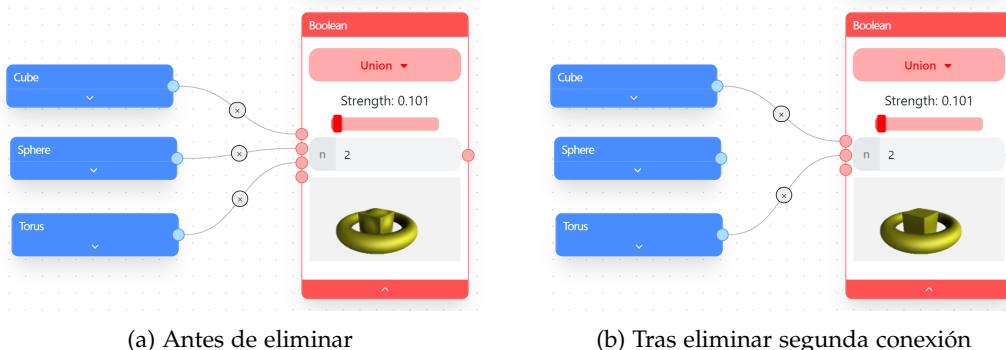


Figura 3.1.: Ejemplo de eliminación de conexión en nodo booleano

Cada nodo tiene una instancia de un componente Shader. Este recibe como parámetro la SDF de cada nodo, y lo renderiza usando *spheretracing* como se explicó en la [Sección 2.1](#) aplicando los algoritmos de iluminación y sombras de la [Sección 2.2](#). Para la creación del lienzo se ha usado el paquete `gl-react`

3. Desarrollo e implementación

Como uniforms se pasan:

- Material de la primitiva como varios vec3 y float,
- Resolución del lienzo en píxeles como un vec2,
- La dirección, color y tamaño de las luces como float[] agrupados de tres en tres en el caso de la dirección y el color. Dado que GLSL solo admite arrays de longitud fija, se ha fijado el número de luces en cuatro, aunque por defecto solo se utilizan dos al igual que en el ejemplo de la [Sección 2.2](#),
- Dos ángulos como un vec2 y una distancia como float actuando como coordenadas esféricas del observador respecto al origen. Ambos parámetros se controlan por el usuario, los ángulos con el movimiento del ratón y la distancia con la rueda.

3.2. Panel de primitivas

3.3. Gestor de estado

Para esta tarea se ha hecho uso de [Zustand](#), un paquete de gestión de estado para JavaScript. Con él se pueden crear contenedores formados por atributos y métodos para gestionarlos. Cuando un componente quiere acceder a un contenedor, basta con que se suscriba a sus cambios a través del *hook* que proporciona Zustand: `useStore`. Se hace uso de dos contenedores: uno para las primitivas definidas y otro para gestionar el estado del editor de nodos. De este último hablaremos en la siguiente sección, pues solo es usado por el componente del editor de nodos. Sin embargo el contenedor de primitivas es usado tanto por el editor de nodos como por el creador de primitivas, ya que ambos deben leer de él para saber cuales son las primitivas disponibles y pueden escribir para crear una nueva primitiva, ya sea a través del diálogo un diálogo de creación o como resultado del editor de nodos.

3.4. 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.

3.4.1. Clase Monomial

3.4.2. Clase Polynomial

3.4.3. Clase Ideal

4. Pruebas y rendimiento

5. Conclusiones y líneas futuras

A. Resultados de operaciones sobre SDF

TODO

B. Resultado de técnicas empleadas al enderezar SDFs

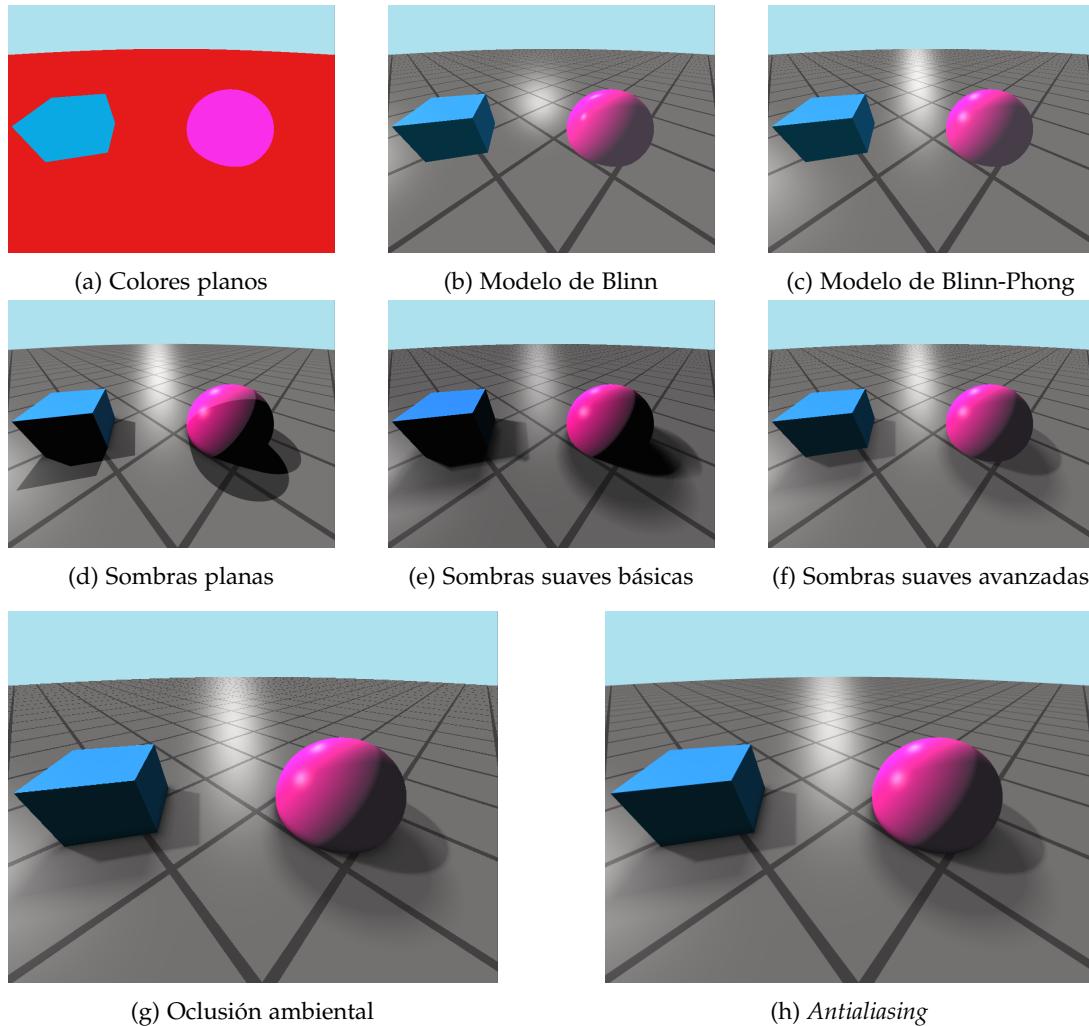


Figura B.1.: Renderizado de SDF aplicando técnicas de iluminación y renderizado avanzadas

Bibliografía

- [1] T. W. Sederberg and A. K. Zundel, "Scan line display of algebraic surfaces," in *International Conference on Computer Graphics and Interactive Techniques*, 1989.
- [2] J. C. Hart, D. J. Sandin, and L. H. Kauffman, "Ray tracing deterministic 3-d fractals," in *Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pp. 289–296, 1989.
- [3] J. C. Hart, "Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces," *The Visual Computer*, vol. 12, no. 10, pp. 527–545, 1996.
- [4] <https://math.okstate.edu/people/binegar/4013-U98/4013-106.pdf>. Accessed: 2023-5-30.
- [5] C. Bálint, G. Valasek, and L. Gergó, "Operations on signed distance functions," *Acta Cybern.*, vol. 24, no. 1, pp. 17–28, 2019.
- [6] Wikipedia contributors, "Función distancia con signo." https://es.wikipedia.org/w/index.php?title=Funci%C3%B3n_distancia_con_signo&oldid=149321457. Accessed: 23-05-21.
- [7] C. Dapogny and P. Frey, "Computation of the signed distance function to a discrete contour on adapted triangulation," *Calcolo*, vol. 49, pp. 193–219, Sep 2012.
- [8] M. Delfour and J.-P. Zolésio, *Shapes and Geometries: Analysis, Differential Calculus and Optimization*, vol. 4. 01 2001.
- [9] B. Wyvill, A. Guy, and E. Galin, "Extending the csg tree. warping, blending and boolean operations in an implicit surface modeling system," in *Computer Graphics Forum*, vol. 18, pp. 149–158, Wiley Online Library, 1999.
- [10] B. Foundation, "Blender - 3d creation software - git repository," Último acceso: 6 mayo 2023.
- [11] M. Molecule, "Dreams," 2020.
- [12] I. Quílez, "Smooth minimun." <https://iquilezles.org/articles/smin/>, [En línea]. Último acceso: 6 mayo 2023.
- [13] I. Quílez, "Distance stimation." <https://iquilezles.org/articles/distance/>, [En línea]. Último acceso: 6 junio 2023.
- [14] P.-A. Fayolle, "Signed distance function computation from an implicit surface," *arXiv preprint arXiv:2104.08057*, 2021.
- [15] D. A. Cox, J. Little, and D. O'Shea, *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra*. Undergraduate Texts in Mathematics, Springer New York, 2008.
- [16] D. Y. Melesse, "Groebner bases and an improvement on buchberger's algorithm," 2007.
- [17] A. S. Semyonov, *Buchberger's Criteria and Trivial Syzygies*.
- [18] O. Foundation, "Opengl - api for rendering 2d and 3d graphics."
- [19] OpenGL, "Coordinate systems." <https://learnopengl.com/Getting-started/Coordinate-Systems>. Accessed: 2023-5-9.
- [20] OpenGL, "Matrices." <http://www.opengl-tutorial.org/es/beginners-tutorials/tutorial-3-matrices/#matrices-modelo-vista-y-proyecci%C3%B3n>. Accessed: 2023-5-10.
- [21] C. Ureña, "Apuntes de la asignatura informática gráfica de la universidad de granada," curso 21-22.

Bibliografía

- [22] T. Thormählen, "Light and materials - graphics programming - part 10 - chapter 1." https://www.mathematik.uni-marburg.de/~thormae/lectures/graphics1/graphics_10_1_eng_web.html, Dec. 2022. Accessed: 2023-5-20.
- [23] B. T. Phong, "Illumination for computer generated pictures," *Commun. ACM*, vol. 18, p. 311–317, jun 1975.
- [24] I. Quílez, "Normals for an sdf." <https://iquilezles.org/articles/normalsSDF/>, [En línea]. Último acceso: 6 mayo 2023.
- [25] S. Aaltonen, "GPU-based clay simulation and ray-tracing tech in claybook," 2018.
- [26] nurof3n, "Ray marched improved shadows."
- [27] "The book of shaders." <https://thebookofshaders.com/glossary/?search=smoothstep>. Accessed: 2023-6-1.
- [28] Digital Foundry, "Ratchet and clank: Rift apart PS5 - performance + graphics - all modes tested!," June 2021.
- [29] A. Evans, "Fast approximations for global illumination on dynamic scenes," in *ACM SIGGRAPH 2006 Courses*, pp. 153–171, 2006.
- [30] G. Geer, "Normally sampled ambient occlusion." <https://www.csh.rit.edu/~gman/PersonalWebpage/ao.html>. Accessed: 2023-6-4.
- [31] Wikipedia contributors, "Supersampling." <https://en.wikipedia.org/w/index.php?title=Supersampling&oldid=1143405382>, Mar. 2023. Accessed: 2023-6-4.