



UNIVERSIDAD DE GRANADA

ETSIIT, Facultad de Ciencias

DOBLE GRADO EN ING. INFORMÁTICA Y MATEMÁTICAS

TRABAJO DE FIN DE GRADO

Visualización de superficies en 3D

Representación por *raytracing* de superficies definidas por funciones distancia con signo y conversión de superficies implícitas y paramétricas usando bases de Gröbner

Presentado por:

Daniel Zufrí Quesada

Tutores:

Pedro A. García Sánchez
Departamento de Álgebra

Carlos Ureña Almagro
Departamento de Lenguajes y Sistemas Informáticos

Curso académico 2022-2023

Visualización de superficies en 3D

Representación por *raytracing* de superficies definidas por funciones distancia con signo y conversión de superficies implícitas y paramétricas usando bases de Gröbner

Daniel Zufrí Quesada

Daniel Zufrí Quesada *Visualización de superficies en 3D: Representación por raytracing de superficies definidas por funciones distancia con signo y conversión de superficies implícitas y paramétricas usando bases de Gröbner*.

Trabajo de Fin de Grado. Curso académico 2022-2023.

Tutores

Pedro A. García Sánchez

Departamento de Álgebra

Carlos Ureña Almagro

*Departamento de Lenguajes y Sistemas
Informáticos*

Doble Grado en Ing.
Informática y Matemáticas

ETSIIT, Facultad de
Ciencias
Universidad de Granada

Visualización de superficies en 3D: Representación por *raytracing* de superficies definidas por funciones distancia con signo y conversión de superficies implícitas y paramétricas usando bases de Gröbner

Daniel Zufri Quesada (alumno)

Palabras clave: función distancia con signo, *raytracing*, *spheretracing*, modelo *BlobTree*, polinomios en varias variables, bases de Gröbner, implicitación.

Resumen

La búsqueda constante de métodos para representar información de manera más eficiente y facilitar la aplicación de nuevas técnicas es una característica tanto en el ámbito de la informática como en el de las matemáticas. En este trabajo, nos enfocaremos en el estudio de las funciones de distancia con signo, las cuales miden la distancia de cada punto a un conjunto en función de si se encuentra en un “lado” u otro del conjunto. Exploraremos cómo utilizar estas funciones para representar superficies en tiempo real mediante el uso de *raytracing* y cómo manipularlas mediante el modelo *BlobTree* en forma de árbol.

No obstante, no nos detendremos ahí. También investigaremos cómo obtener una función de distancia con signo aproximada a partir de superficies definidas mediante ecuaciones implícitas o paramétricas, habiendo explorado previamente el problema de la implicitación y su solución mediante bases de Gröbner para estas últimas.

El objetivo fundamental de este trabajo es demostrar la utilidad de esta forma de representar la geometría a través de una aplicación web que implemente eficazmente todos los conceptos y técnicas relacionados con las funciones de distancia con signo y las bases de Gröbner que exploraremos a lo largo del trabajo. Esta aplicación aprovechará la API de rasterización para permitir que el algoritmo de *raytracing* se ejecute en la mayoría de los dispositivos actuales. Además, desarrollaremos nuestra propia librería de código abierto que la aplicación utilizará para trabajar con polinomios de múltiples variables y resolver el problema de la implicitación.

Visualization of 3D surfaces: Raytracing representation of surfaces defined by signed distance functions and conversion of implicit and parametric surfaces using Gröbner bases

Daniel Zufri Quesada (student)

Keywords: signed distance function, raytracing, spheretracing, *BlobTree* model, multivariate polynomials, Gröbner basis, implicitation

Abstract

The constant quest for ways to represent information more efficiently and enable the application of new techniques is a common thread in both the fields of computer science and mathematics. In this work, we will focus on the study of signed distance functions, which measure the distance from each point to a set based on whether it lies on one “side” or the other of the set. We will explore how to use these functions to represent surfaces in real-time through raytracing and manipulate them using the *BlobTree* model in a tree-like structure.

However, we won’t stop there. We will also investigate how to obtain an approximate signed distance function from surfaces defined by implicit or parametric equations having previously explored the problem of implicitization and its solution using Gröbner bases for the latter.

The primary goal of this work is to showcase the utility of this way of representing geometry through a web application that effectively puts into practice all the concepts and techniques associated with signed distance functions and Gröbner bases that we will explore throughout the work. In particular, this application will use the rasterization API to enable the raytracing algorithm to run on most current devices. Additionally, we will develop our own open-source library that the application will use to work with multivariable polynomials and solve the implicitization problem.

DECLARACIÓN DE ORIGINALIDAD

D./Dña. Daniel Zufrí Quesada

Declaro explícitamente que el trabajo presentado como Trabajo de Fin de Grado (TFG), correspondiente al curso académico 2022-2023, es original, entendido esto en el sentido de que no he utilizado para la elaboración del trabajo fuentes sin citarlas debidamente.

En Granada a 7 de septiembre de 2023

Fdo: Daniel Zufrí Quesada

AUTORIZACIÓN DE DEPÓSITO EN LA BIBLIOTECA

D./Dña. Daniel Zufri Quesada

Alumno del Doble Grado en Ingeniería Informática y Matemáticas de la Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación y de la Facultad de Ciencias de la Universidad de Granada, con DNI 77036772C, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

En Granada a 7 de septiembre de 2023

Fdo: Daniel Zufri Quesada

A mi familia, que ha hecho todo lo posible para que llegue hasta aquí.

A Mónica, que me ha acompañado en este camino y siempre ha apoyado mis decisiones.

A mis tutores, Carlos y Pedro, que me han prestado su tiempo y experiencia para guiarme y ayudarme con mis problemas y a expresar mis ideas.

Índice general

Introducción	xvii
Estado del arte	xix
1 Fundamentos matemáticos de las SDFs	1
1.1 Diferenciabilidad	2
1.2 Cálculo del vector normal	5
1.3 Funciones cota de distancia	6
1.4 Operaciones sobre SDF	7
1.4.1 Operaciones booleanas	7
1.4.2 Operaciones afines	12
1.4.3 Operaciones deformantes	13
1.4.4 Operaciones de repetición	14
1.5 Obtención de una SDF a partir de ecuaciones implícitas	16
1.6 Implicitación de parametrizaciones racionales con bases de Gröbner	17
1.6.1 Polinomios en varias variables	18
1.6.2 Bases de Gröbner	21
1.6.3 Teorema de implicitación	27
1.7 Implicitación de parametrizaciones racionales con resultantes	30
1.7.1 Conceptos básicos y caso de una variable	30
1.7.2 Resultante auxiliar	32
2 Algoritmos de visualización de SDFs	37
2.1 Renderizado por <i>raytracing</i> en GPU usando un <i>fragment shader</i>	39
2.1.1 Creación del lienzo	39
2.1.2 Generación de rayos primarios	43
2.1.3 Algoritmos de intersección rayo-escena: <i>raymarching</i> y <i>spheretracing</i>	45
2.2 Modelos de iluminación y sombras	47
2.2.1 Modelos de Phong y Blinn-Phong	48
2.2.2 Sombras	54
2.2.3 Oclusión ambiental	58
2.3 <i>Antialiasing</i>	61
3 Diseño, desarrollo e implementación	67
3.1 Requerimientos	67
3.1.1 Requerimientos funcionales	67
3.1.2 Requerimientos no funcionales	68
3.2 Diseño del interfaz	69
3.3 Herramientas <i>software</i>	74
3.4 Implementación de la aplicación web	77
3.4.1 Gestor de estado	77
3.4.2 Lienzo	79

Índice general

3.4.3	Editor de nodos	92
3.4.4	Librería de polinomios en varias variables	95
3.4.5	Panel de primitivas	106
4	Pruebas y rendimiento	111
4.1	Lienzo	111
4.2	Librería de polinomios en varias variables	117
5	Conclusión y líneas futuras	123
	Bibliografía	127

Introducción

En el campo de los gráficos generados por computador, el método más asentado para describir objetos tridimensionales es mediante mallas de polígonos. Cuando se trata de superficies, la representación implícita es también bastante usada debido a la facilidad que aporta a la hora de realizar geometría de sólidos constructiva o representar objetos sólidos. Sin embargo, esta representación no es apta para ser usada en tiempo real, y es aquí donde en los últimos años ha surgido un enfoque alternativo que muestra un gran potencial: la utilización de funciones distancia con signo (SDF). Estas permiten representar objetos geométricos realmente complejos a través de una única función escalar que asigna a cada punto del espacio su distancia signada respecto a la frontera del objeto en tiempo real. Esta técnica está cada vez más presente en los programas de modelado y visualización actuales, como explica en el siguiente capítulo, principalmente porque 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 propósito principal de este Trabajo de Fin de Grado es la creación de una aplicación web interactiva que facilite la creación e interacción con superficies generadas a través de SDFs usando una estructura en forma de árbol de forma intuitiva para el usuario sin que este necesite conocimientos al respecto. No obstante, dado que en el ámbito matemático la representación de superficies suele ser a través de ecuaciones implícitas o paramétricas, estudiaremos también como obtener una SDF que represente la misma información que estas ecuaciones, siendo de especial interés este último caso. Para lograr este objetivo, llevaremos a cabo el desarrollo de la aplicación aprovechando las tecnologías web y *frameworks* modernos así como desarrollando nuestras propias herramientas, evaluando la eficiencia y el rendimiento de la aplicación en términos de tiempos de respuesta y calidad visual. Así, los objetivos que nos proponemos alcanzar con nuestra aplicación son los siguientes.

- Comprendión profunda de las **funciones distancia con signo**, sus propiedades y aplicaciones tanto en el ámbito matemático como en el de la Informática Gráfica (IG).
- Análisis de la teoría de polinomios en varias variables y soluciones al problema de implicitación, principalmente usando **bases de Gröbner** y enfocando su uso al **problema de implicitación** para obtener la SDF asociada a una superficie paramétrica.
- Desarrollo de una aplicación web para la representación y manipulación de SDFs a través de un modelo de operaciones con estructura de árbol.

El código fuente de la aplicación y su *build* para ejecución local se encuentra disponible en el repositorio <https://github.com/Daniel2000815/SDF-Visualizer/>. También se puede ejecutar directamente desde el navegador en <https://daniel2000815.github.io/SDF-Visualizer/>.

Estructura del documento

Los temas que acabamos de introducir se desarrollan con detalle a lo largo del trabajo como sigue:

- En el [Capítulo 1](#) exploraremos los fundamentos teóricos y matemáticos de las SDFs que nos servirán de motivación y serán necesarios en el resto de capítulos, estudiando con especial interés su diferenciabilidad ([Sección 1.1](#)). Comprenderemos la capacidad de las SDFs para describir superficies y volúmenes de manera precisa y compacta, así como las posibilidades para su manipulación ([Sección 1.4](#)), como las operaciones booleanas o las deformaciones. Acabaremos explicando como podemos obtener un SDF a partir de una ecuación implícita ([Sección 1.5](#)) y resolviendo el problema de implicitación con bases de Gröbner ([Sección 1.6](#) y resultantes ([Sección 1.7](#)) para así poder sacar provecho de las ventajas que nos aporta el uso de las SDFs también a superficies definidas mediante ecuaciones implícitas o paramétricas.
- En el [Capítulo 2](#) analizaremos las diferencias y similitudes de los dos principales métodos de renderizado actuales: la rasterización y el *raytracing*. Veremos paso a paso como combinar ambas técnicas para conseguir representar cualquier superficie dada por una SDF mediante *spheretracing* ([Sección 2.1](#)). Para ello empezaremos definiendo la geometría necesaria con un *vertex shader* asociado en la [Subsección 2.1.1](#), habiéndonos familiarizado previamente con los diferentes sistemas de coordenadas comúnmente usados en IG. Posteriormente veremos cómo colorear la geometría definida anteriormente para representar la superficie sobre ella a través de un *fragment shader*. Finalmente, incluiremos el modelo de iluminación de Blinn-Phong ([Sección 2.2](#)), al cual añadiremos realismo usando algoritmos avanzados de cálculo de sombras ([Subsección 2.2.2](#)) y oclusión ambiental ([Subsección 2.2.3](#)), y usaremos *antialiasing* para obtener una imagen lo más libre de imperfecciones posible, todo esto usando un *hardware* accesible a todo tipo de usuario.
- En el [Capítulo 3](#) explicaremos el proceso de diseño y desarrollo de la aplicación en React, poniendo especial atención en cómo trasladar la teoría que vimos en la [Sección 2.1](#) sobre *spheretracing* a la práctica usando WebGL y GLSL ([Subsección 3.4.2](#)) y el funcionamiento interno del editor de nodos en árbol, con el que crear nuevas superficies ([Subsección 3.4.3](#)) a partir de las ya existentes, y nuestra librería de polinomios en varias variables ([Subsección 3.4.4](#)).
- En el [Capítulo 4](#) analizaremos el proceso de validación que se ha seguido a lo largo del desarrollo de la aplicación y la librería, y como estas se comportan en escenarios de estrés, incluyendo pruebas de rendimiento del visualizador basado en SDFs.
- Finalmente, en el [Capítulo 5](#) haremos una reflexión sobre lo que aporta este trabajo al panorama actual, qué objetivos de los propuestos se han alcanzado, y como este puede ser continuado en el futuro.

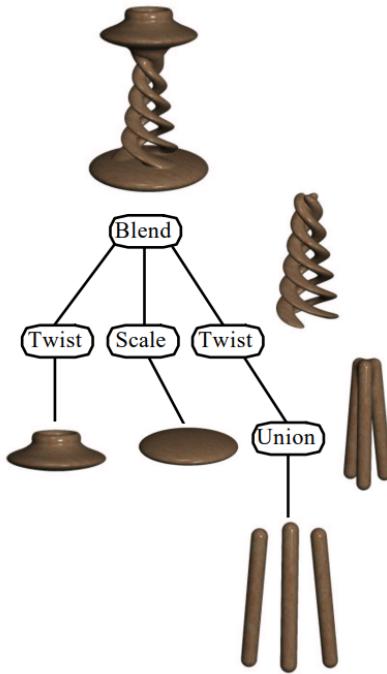
Estado del arte

La introducción original de la idea de modelar usando superficies implícitas fue hecha por Jim Blinn en 1982, en un artículo [1] donde presenta un método para representar superficies dadas por ecuaciones de orden mayor a dos usando funciones de densidad Gaussianas. Este modelo tenía la ventaja de permitir la combinación suave de primitivas, y en los siguientes años surgirían modelos similares tratando de conseguir esto mismo bajo la denominación de *Blobby Molecules*, debido a la capacidad que tienen estos de combinar primitivas. Entre ellos destaca el de A. Pasko en 1995 [2], que era más completo que el de Blinn al incluir la posibilidad de aplicar operadores de deformación y booleanos. Llegado 1999, Brian Wyvill, Eric Galin y Andrew Guy presentan en su artículo *The BlobTree: Warping, Blending and Boolean Operations in an Implicit Surface Modeling System* [3] el modelo *BlobTree*, que permite realizar las mismas funciones que los modelos anteriores de forma más estructurada, y motiva el desarrollo de este trabajo.

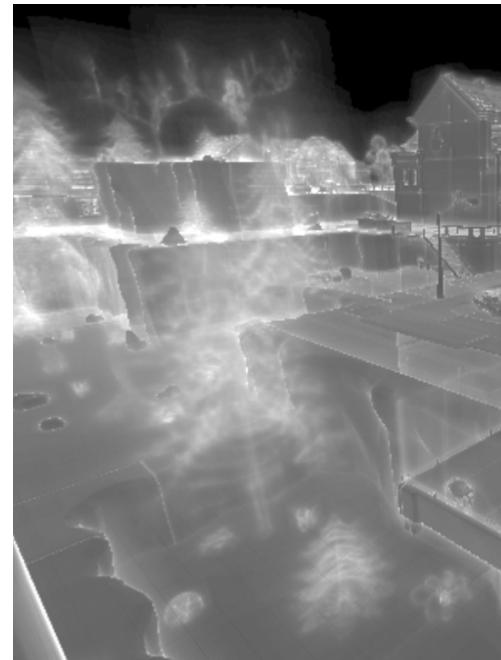
Las ventajas que aporta el uso de funciones distancia con signo hace que podamos ver su uso en multitud de productos actuales, entre los que destacamos los siguientes.

- A través de un algoritmo que permite obtener la representación por SDF de una malla de triángulos, Blender [4] utiliza SDFs para realizar geometría de sólidos constructiva y otras operaciones de cuerpo blando.
- Clayxels [5] es un paquete para el motor Unity3D que permite esculpir modelos combinando primitivas definidas por SDFs y convertir el resultado a una malla de polígonos que pueda ser usada de forma convencional en el motor.
- El motor Unreal Engine [6] incorpora una funcionalidad para aproximar la función distancia con signo de los modelos estáticos en escena y almacenar dicha información en una textura. Una vez realizado este proceso, usa los datos para cálculos de sombras y oclusión ambiental dinámica. También permite usar la SDF para calcular colisiones de partículas (Unity3D también tiene esta característica), calcular mapas de flujo, etc.
- Claybook [7] es un videojuego que implementa un escenario totalmente destructible definido mediante SDFs y renderizado con *raytracing*. Utiliza varias de las técnicas de renderizado que veremos en el [Capítulo 2](#).
- Dreams [8] es un videojuego que permite a los jugadores crear y programar nuevas experiencias interactivas que compartir con los usuarios. Entre sus funcionalidades se incluye la creación de modelos 3D en tiempo real, que usa operadores booleanos suavizados sobre funciones distancia con signo de forma interna y transparente al jugador.

No podemos dejar de mencionar la contribución del español Íñigo Quílez. Su blog [9] contiene artículos relacionados con la IG y las matemáticas usadas en ella, y su sección dedicada a las SDFs y el *raymarching* ha sido una inspiración a lo largo del desarrollo de este trabajo. Esta incluye artículos que discuten varios de los temas aquí tratados, como la



(a) Ejemplo de uso del modelo *BlobTree* [1]



(b) Visualización de la SDF de una escena en Unreal Engine [6]

occlusión ambiental, cálculo de normales, etc., e incluye una lista muy útil de primitivas y operadores de SDFs. Además de esta contribución teórica, Íñigo Quílez es también el coautor junto a Pol Jeremías de ShaderToy [10], una plataforma en la que crear y compartir obras generadas con un *fragment shader* sobre un lienzo. En ella hay muchos ejemplos de uso de técnicas avanzadas sobre SDFs con *raytracing* ([Figura 2](#)), similares a las que desarrollaremos en profundidad en los siguientes capítulos.



(a) "Selfie Girl" [11], Íñigo Quílez



(b) Mandelbulbo [12], usuario EvilRyu

Figura 2: Ejemplos de obras creadas en Shadertoy usando únicamente SDFs

Comentamos por último las soluciones *software* existentes más en línea con lo que pretendemos realizar nosotros. En cuanto a definición de SDFs, la librería hg_sdf [13] permite definir primitivas y combinarlas usando operadores booleanos. No obstante, para poder dibujar una escena como la mostrada en la [Figura 3](#) el usuario tiene que implementar su propio algoritmo de *raytracing*, que junto a que la definición y combinación de primitivas se tenga que

realizar a través de código, supone una gran barrera de entrada para un usuario promedio. Si hablamos de recursos para trabajar con superficies definidas implícitamente, las únicas opciones son visualizadores como el desarrollado por el usuario matkcy llamado Implicit3D [14], que permite visualizar superficies implícitas (también curvas parametrizadas), pero no manipularlas de ninguna forma que no sea modificar la propia ecuación. Lo mismo ocurre para las superficies paramétricas, solo encontramos visualizadores, como el de GeoGebra [15].

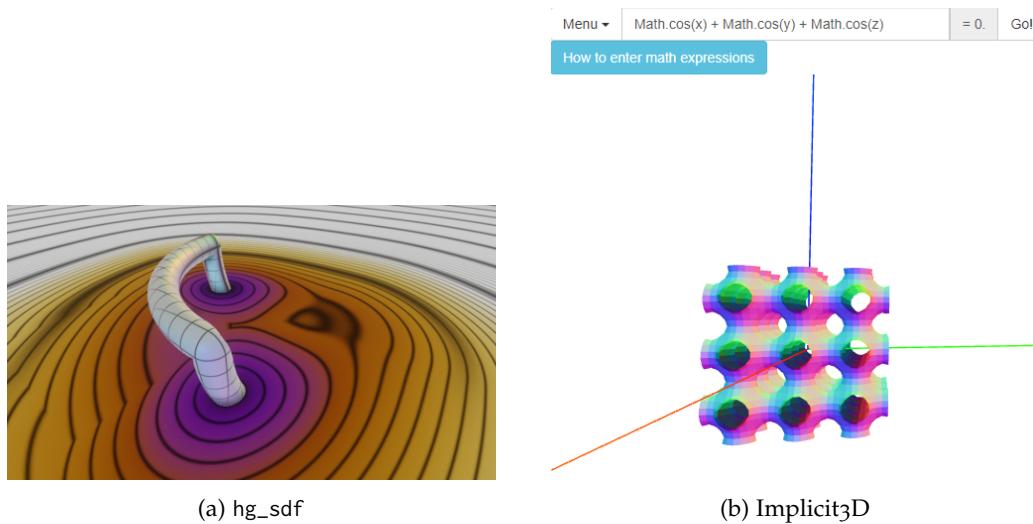


Figura 3: Aplicaciones de visualización de superficies

Respecto a aplicaciones que implementen un editor en forma de árbol [16], encontramos la mayoría de ejemplos en el ámbito de la composición de *shaders* o sistemas de programación basados en nodos, como ocurre en el caso de los ya mencionados Unity3D, Unreal Engine o Blender. No obstante, encontramos ejemplos interesantes de editores de nodos para SDFs, como Material Maker [17], que a pesar de ser principalmente un editor de materiales, permite también representar primitivas predefinidas y aplicarles operaciones, SDFPerf [18], cuyo uso requiere de compilación previa del código fuente y aún está en desarrollo, o SDF-UI [19], que ofrece bastantes posibilidades pero cuya curva de aprendizaje inicial es bastante pronunciada y todavía no se encuentra en versión estable.

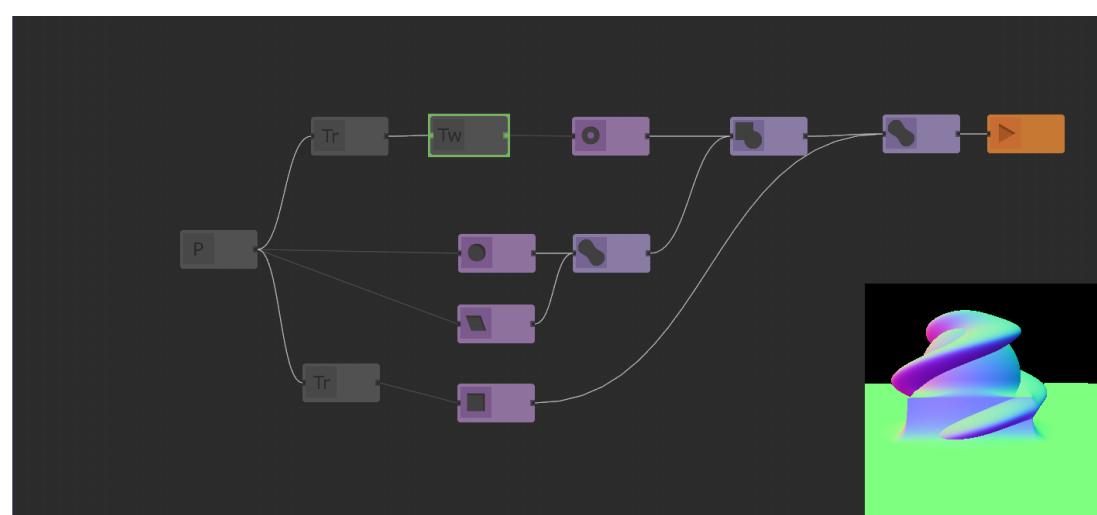


Figura 4: Aplicación SDFPerf

1 Fundamentos matemáticos de 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 dos. 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 [20] presentaron en 1989 un método para la representación 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 [21] presenta una técnica de *raymarching* para la representación de fractales usando funciones distancia con signo. Posteriormente, en 1995 [22] 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 cómo 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 :

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

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:

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

En la literatura es común referirse a ellas por sus inglés SDF (*Signed Distance Function*), y la denominaremos 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 es construir una escena definida como la isosuperficie generada por una SDF. A partir de ahora, tomaremos $p = (x,y,z) \in \mathbb{R}^3$.

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

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

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

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

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

- **Toro sobre el eje Y de radios R y r , con $R > r$:**

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

$$\phi((x,y,z)) = \left\| (\|(x,0,z)\| - R, y) \right\| - r.$$

1.1. Diferenciabilidad

Antes de seguir avanzando vamos a realizar un estudio de la diferenciabilidad de las funciones distancia con signo, pues nos será de utilidad en las siguientes secciones. Empezamos recordando varios conceptos de análisis diferencial [23] 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 $\phi: U \rightarrow \mathbb{R}$. Para $i \in \{1, 2, 3\}$, definimos la **i -ésima derivada parcial** de ϕ en $p_0 \in \mathbb{R}^3$ como

$$\frac{\partial \phi}{\partial x_i}(p_0) = \lim_{h \rightarrow 0} \frac{\phi(p_0 + he_i) - \phi(p_0)}{h}.$$

Definición 1.5. 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. Definimos la

diferencial de ϕ en p_0 como la suma de todas sus parciales en dicho punto, y la denotamos como $d\phi(p_0)$.

Definición 1.6. Dado un abierto $U \subseteq \mathbb{R}^3$, diremos que la **clase de diferenciabilidad** de una función $\phi : U \rightarrow \mathbb{R}$ es $C^n(U)$ si para $i \in \{1, 2, 3\}$ y $j \in \{0, \dots, n\}$ existen y son continuas todas las parciales

$$\frac{\partial^j \phi}{\partial x_i}(p), \text{ para todo } p \in U.$$

Definición 1.7. 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 1.8. 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:

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

Ahora mismo, dado una función distancia con signo arbitraria no tenemos información alguna sobre su diferenciabilidad, ya que su expresión puede ser de lo más variada y compleja. Veamos una propiedad que cumplen todas las funciones distancia con signo y que nos permitirá obtener algo de información al respecto [24, 25].

Definición 1.9. Una campo escalar $\phi : \mathbb{R}^3 \rightarrow \mathbb{R}$ se dice **lipschitziano** si existe una constante $L > 0$ tal que

$$|\phi(p) - \phi(q)| \leq L \|p - q\|, \text{ para todo } p, q \in \mathbb{R}^3.$$

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

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

en todo punto donde sea diferenciable.

Lema 1.1. Sea $\phi : \mathbb{R}^3 \rightarrow \mathbb{R}$ la función distancia con signo asociada a Ω . Entonces ϕ es lipschitziana con constante $L = 1$.

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

$$\phi(p) \leq \|p - s\| = \|p - q + q + s\| \leq \|p - q\| + \|q - s\|.$$

Por tanto, $\phi_\Omega(p) - \|p - q\| \leq \|q - s\|$, luego $\phi_\Omega(p) - \|p - q\| \leq \inf_{s \in \delta\Omega} (\|q - s\|) = \phi_\Omega(q)$ y obtenemos

$$\phi_\Omega(p) - \phi_\Omega(q) \leq \|p - q\|.$$

De forma análoga podemos ver que $\phi_\Omega(q) - \phi_\Omega(p) \leq \|q - p\|$, concluyendo que

$$|\phi_\Omega(p) - \phi_\Omega(q)| \leq 1 \cdot \|p - q\|. \quad \square$$

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_Ω 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 suficientemente regular [26, 27]. Para ello necesitaremos introducir el concepto de esqueleto de una superficie [25] y repasar algunas definiciones básicas asociadas a superficies en el espacio [28].

Definición 1.10. Sea $\phi_\Omega : \mathbb{R}^3 \rightarrow \mathbb{R}$ una función distancia con signo. 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 $\delta\Omega$:

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

Definición 1.11. Dado $I \subseteq \mathbb{R}$, llamamos **curva parametrizada** a una aplicación

$$\begin{aligned}\alpha : I &\rightarrow \mathbb{R}^3, \\ t &\mapsto (x(t), y(t), z(t)),\end{aligned}$$

donde $x, y, z : I \rightarrow \mathbb{R}$ son diferenciables.

Definición 1.12. Decimos que $\Omega \subseteq \mathbb{R}^3$ es una **superficie regular** si para cada $p \in \Omega$ existen abiertos $U \subseteq \mathbb{R}^2$ y $V \subseteq \mathbb{R}^3$ junto a una aplicación $\psi : U \rightarrow V \cap \Omega$ tal que:

1. ψ es un homeomorfismo, es decir, es continua, biyectiva y con inversa continua,
2. ψ es diferenciable y su diferencial es inyectiva.

Definición 1.13. Sea Ω una superficie regular y $p \in \Omega$. Dados $\varepsilon > 0$ y una curva parametrizada diferenciable

$$\alpha :]\varepsilon, \varepsilon[\rightarrow \mathbb{R}^3 \text{ tal que } \text{Img}(\alpha) \subset \Omega \text{ y } \alpha(0) = p,$$

diremos que $\alpha'(0)$ es un **vector tangente** a Ω en p .

Definición 1.14. Sea Ω una superficie regular, $p \in \Omega$ y $T_p\Omega$ el plano vectorial contenido todos los vectores tangentes a Ω en p . Llamamos **plano tangente** a Ω en p al conjunto $p + T_p\Omega$.

Definición 1.15. Sea Ω una superficie regular y $p \in \Omega$. El **vector normal** a Ω en p es el vector $N_p \in \mathbb{R}^3$ de norma uno perpendicular al plano tangente de Ω en p .

El siguiente teorema, cuya demostración podemos consultar en [26], nos proporciona una caracterización geométrica de la diferenciabilidad de cualquier función distancia con signo ϕ_Ω bajo ciertas hipótesis de regularidad para Ω .

Teorema 1.1. *Sea $\Omega \subseteq \mathbb{R}^3$ cuya frontera es regular y $\phi_\Omega : \mathbb{R}^3 \rightarrow \mathbb{R}$ la función distancia con signo asociada a Ω . 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 $\nabla S_{\phi_\Omega}(p) = N_p$,

2. $p \notin \delta\Omega$ y ϕ_Ω es diferenciable en p si y solo si $p \in \mathbb{R}^3 \setminus \epsilon(\Omega)$, 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. Toda función distancia con signo $\phi: \mathbb{R}^3 \rightarrow \mathbb{R}$ satisface la ecuación de la eikonal

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

en todo punto p donde sea diferenciable.

1.2. Cálculo del vector normal

En los siguientes capítulos veremos que el acceso al vector normal nos resultará indispensable para aplicar ciertas técnicas de renderizado. 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 de la función distancia con signo para obtenerlo [29].

Proposición 1.2. 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 $\nabla\phi(s) \perp \alpha$. Como $\alpha(t) \subset S_\phi$ la evaluación de ϕ en cualquier punto de la curva será cero, y por tanto

$$\frac{d}{dt}\phi(\alpha(t)) = 0.$$

Aplicando la regla de la cadena obtenemos

$$\frac{d\phi \circ \alpha}{dt} = \frac{\partial\phi}{\partial x} \Big|_s \frac{dx}{dt} \Big|_{t_0} + \frac{\partial\phi}{\partial y} \Big|_s \frac{dy}{dt} \Big|_{t_0} + \frac{\partial\phi}{\partial z} \Big|_s \frac{dz}{dt} \Big|_{t_0} = \nabla\phi(s) \cdot \frac{d\alpha}{dt} \Big|_{t_0} = 0.$$

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 , luego $\nabla\phi(s) \perp S_\phi$. \square

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 (de diferencias finitas) 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** [30], 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 1.3 (Método del tetraedro). *Dado $p \in S_\phi$, una aproximación de su vector normal N_p se obtiene normalizando el vector*

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

Demostración. Por la proposición [Proposición 1.2](#), basta comprobar que \hat{N}_p y $\nabla\phi(p)$ están cerca de ser colineales.

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

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

1.3. Funciones cota de distancia

Hemos visto que las funciones distancia con signo 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 las isosuperficies que generan 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 de *spheretracing* ([Subsección 2.1.3](#)), 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 función distancia con signo [\[22\]](#).

Definición 1.16. 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

$$|\gamma(p)| \leq |\phi(p)|, \quad \text{para todo } p \in \mathbb{R}^3,$$

y que tiene el mismo signo que ϕ en cada punto. En su artículo [\[21\]](#), C. Hart se refiere a ellas como SDB (*Signed Distance Bound*).

Observamos que una SDF es un caso especial de SDB en el que se cumple la igualdad de la definición anterior, y que si ambos tipos de funciones están asociadas a un mismo Ω tendrán los mismos ceros, generando por tanto la misma isosuperficie. Aunque trabajar con una función cota de distancia en *spheretracing* hará que la convergencia sea más lenta al proporcionar una cota más conservativa, en muchos casos será también más rápida de evaluar que una función distancia con signo por tener una expresión más simple, haciendo que sea deseable trabajar con ellas. Habrá otras ocasiones en las que incluso será imposible obtener

una función distancia signada para un cierto Ω . Por estos motivos en la literatura no se suele distinguir entre función distancia con signo y función cota de distancia, y como mucho se utilizan los términos función distancia exacta y aproximada respectivamente cuando se quiere realizar una distinción.

Veamos algunos ejemplos de funciones distancia aproximadas.

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

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

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

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

- **Elipsoide fijados $a, b, c \in \mathbb{R} \setminus \{0\}$.**

$$\Omega = \left\{ (x, y, z) \in \mathbb{R}^3 : \frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 0 \right\},$$

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

1.4. 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 [3], 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.

1.4.1. 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 las SDFs, estas operaciones se implementan fácilmente usando las funciones máximo y mínimo.

Definición 1.17 (Operaciones Booleanas). Sean A y B isosuperficies generadas por ϕ y γ respectivamente. La función μ define la isosuperficie para las siguientes operaciones.

1 Fundamentos matemáticos de las SDFs

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

Solo en el caso de la unión se obtiene una SDF exacta, ya que al aplicar la función máximo 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 *spheretracing* requiera de más iteraciones.

Un problema de usar estas transformaciones es que produce discontinuidades en la derivada de la función 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 funciones distancia de forma más natural. Usaremos una de las más extendidas, usada por programas de modelado 3D como Blender [4] o videojuegos como Dreams [8], y que ha sido estudiada por Íñigo Quílez en su web [31].

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

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 mínimo original para que cumpla ciertos requisitos. Por comodidad, definiremos

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

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 mínimo original será

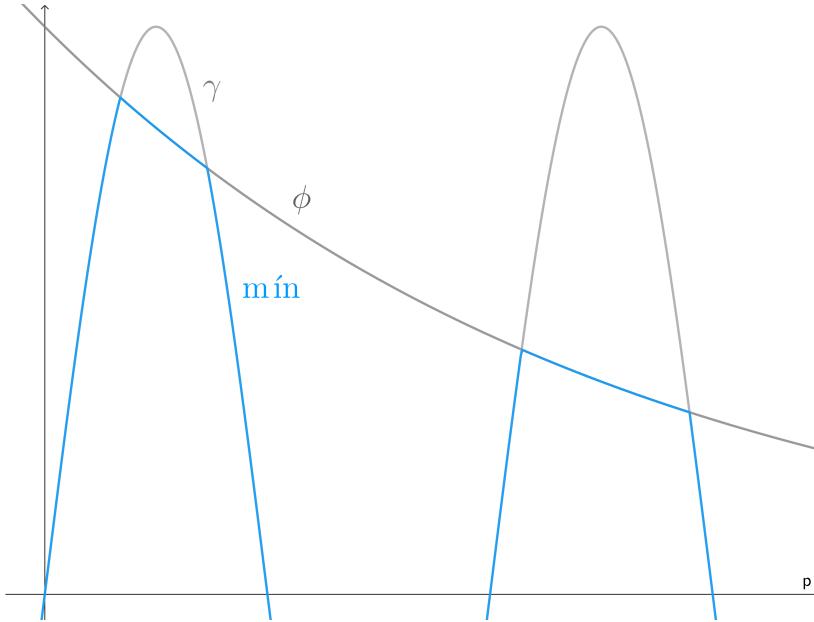
$$\begin{aligned} \text{smín}_{\phi, \gamma} : \mathbb{R}^3 &\rightarrow \mathbb{R}, \\ p &\mapsto \min_{\phi, \gamma}(p) - \omega_k(p). \end{aligned}$$

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

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

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

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

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

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

$$\omega_k(p) = 0, \text{ para todo } p \in \delta B_k.$$

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

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

El valor s es el que deberemos ajustar para que smín cumpla nuestros requisitos. Fijado un $p \in B_k$, consideramos una primera aproximación para ω_k :

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

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

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

$$\phi(p) = \gamma(p), \text{ luego } \frac{\phi(p) - \gamma(p)}{k} = 0, \text{ y por tanto } \omega_k(p) = s.$$

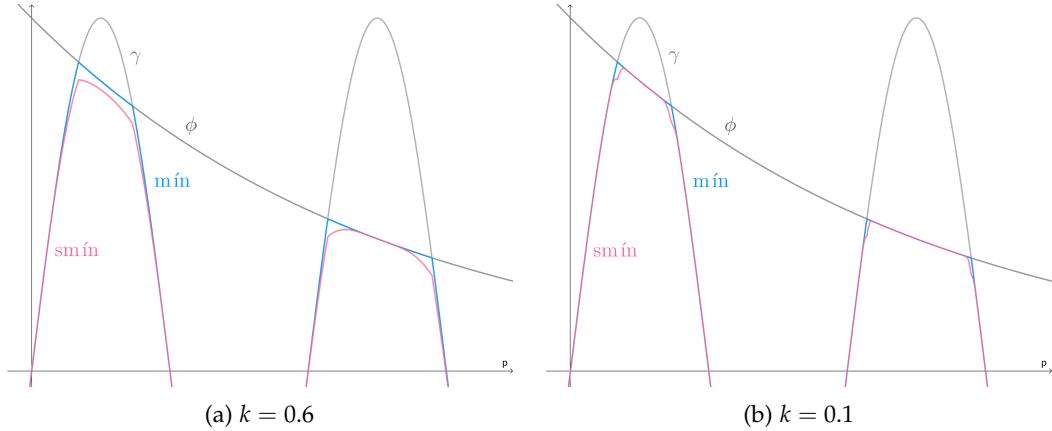


Figura 1.2: Primera aproximación de $smin(p)$ con $s = 0.05$ y $n = 2$

Otra condición necesaria es que sus derivadas parciales sean continuas. Para todo $i \in \{1, 2, 3\}$, estas son de la forma

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

Por tanto, para comprobar que las parciales son continuas cuando $\phi(p) = \gamma(p)$, para todo $i \in \{1, 2, 3\}$ 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 $c \in I$:

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

Hemos llegado a la expresión final

$$\begin{aligned}\omega_k(p) &= \begin{cases} \frac{k}{2n} \left(1 - \frac{|\phi(p) - \gamma(p)|}{k}\right)^n, & |\phi(p) - \gamma(p)| \leq k, \\ 0, & \text{otro caso,} \end{cases} \\ &= \frac{\max(k - |\phi(p) - \gamma(p)|, 0)^n}{2n \cdot k^{n-1}}, \quad s \in \mathbb{R}, n \in \mathbb{N}.\end{aligned}\quad (1.1)$$

Podemos observar los resultados en la [Figura 1.3](#). Finalmente, para obtener una versión suavizada del máximo, es fácil comprobar que

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

Con estos resultados, para que la transición de una superficie a otra en la [Def. 1.17](#) sea gradual basta con sustituir las versiones clásicas de las funciones máximo y mínimo por las que acabamos de obtener.

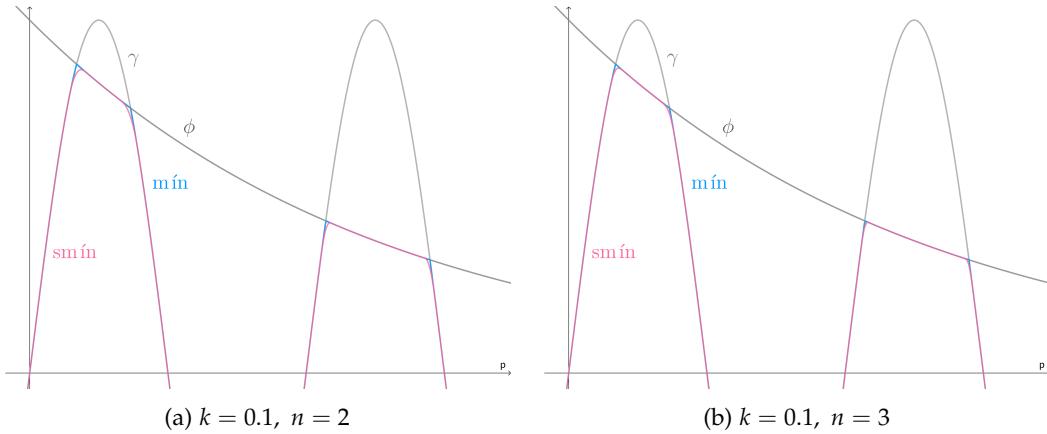


Figura 1.3: Resultado final de $\text{smín}(p)$

Definición 1.18 (Operaciones Booleanas Suavizadas). Sean A y B isosuperficies generadas por ϕ y γ respectivamente. La función μ define la isosuperficie para las siguientes operaciones.

- **Unión suavizada:** $\mu_{unionS}(p) = \min(\phi(p), \gamma(p)) - \frac{\max(k - |\phi(p) - \gamma(p)|, 0)^n}{2n \cdot k^{n-1}}$.
- **Intersección suavizada:** $\mu_{intersS}(p) = -\min(-\phi(p), -\gamma(p)) + \frac{\max(k - |\phi(p) - \gamma(p)|, 0)^n}{2n \cdot k^{n-1}}$.
- **Diferencia suavizada:** $\mu_{difsS}(p) = -\min(-\phi(p), \gamma(p)) + \frac{\max(k - |\phi(p) + \gamma(p)|, 0)^n}{2n \cdot k^{n-1}}$.

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

Observamos que los operadores definidos en la [Def. 1.17](#) no son más que un caso particular de estos últimos cuando k tiende a cero. Este método para obtener una versión suavizada de

las funciones mínimo y máximo no es el único. Hemos elegido debido a que las funciones obtenidas tienen asociado un coste computacional. Además, su deducción es bastante natural y el efecto que tiene el valor k sobre el resultado final es intuitivo para el usuario. En el artículo que hemos mencionado al inicio de la sección, Íñigo Quílez [31] presenta otras tres alternativas a esta versión, a la cual él se refiere como “mínimo suavizado polinomial”, y que también son compatibles con *raymarching*.

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

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,

$$\text{smín}_a, \text{smín}_b, \text{smín}_c = \text{smín}_b, \text{smín}_{a,c}.$$

En la [Figura 1.4](#) podemos ver un ejemplo de uso de estas versiones, y que no hay diferencias notables entre ellas, así que nos quedaremos con el método más eficiente: el polinómico. En la figura además hemos aprovechado los cálculos del valor w_k de la ecuación (1.1) para obtener un valor con el que interpolar la componente difusa (estudiaremos en detalle los atributos de un material en la [Sección 2.2](#)) de ambas primitivas cuando ocurre la transición. En concreto, y siguiendo la misma idea de lo que hacíamos con $\text{smín}_{\phi,\gamma}$, el valor que usaremos para interpolar es

$$\text{interp}(p) = \min_{\phi,\gamma}(p) - \frac{w_k(p) \cdot n}{k}.$$

La interpolación que se lleva a cabo es lineal, de forma que para combinar las componentes difusa de ambas superficies, que denotaremos como dif_ϕ y dif_γ , usaremos la expresión

$$\text{dif}_{\phi \cup \gamma}(p) = \text{dif}_\phi \times (1 - \text{interp}(p)) + \text{dif}_\gamma \times \text{interp}(p).$$

1.4.2. 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. Su funcionamiento se basará 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 la SDF original en $t^{-1}(p)$:

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

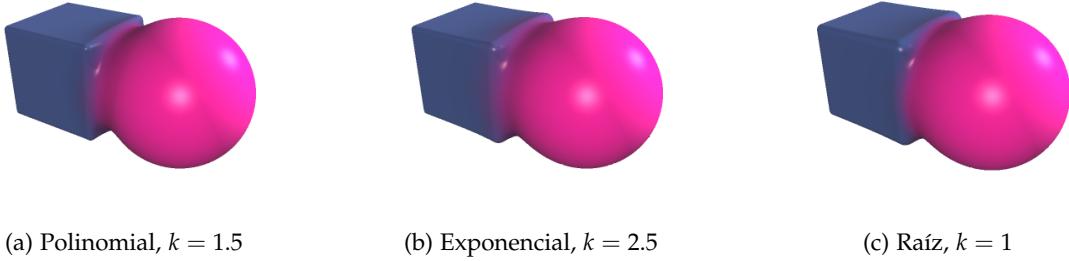


Figura 1.4: Diferentes versiones de la unión suavizada

Este razonamiento funciona bien para transformaciones como las traslaciones o rotaciones, 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 $s \in \mathbb{R}_0^+$

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

Como las distancias se escalan, deberemos hacer lo propio con la función que genere la nueva isosuperficie, aplicándole el mismo factor de escalado s como muestra la Def. 1.19.

Definición 1.19 (Operaciones afines). Sea A una isosuperficie. Definimos las SDFs para las siguientes operaciones.

- **Traslación de vector $v \in \mathbb{R}^3$:** $\text{sdf}_{\text{traslacion}}(p) = \text{sdf}_A(p - v)$.
- **Escalado uniforme de dimensiones $(s, s, s) \in \mathbb{R}^3$:** $\text{sdf}_{\text{escalado}}(p) = \text{sdf}_A(p/s) \cdot s$.
- **Rotaciones de ángulo $\alpha \in \mathbb{R}$ sobre los ejes x, y, z :**

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

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

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

Observamos que en este caso sí que obtenemos funciones distancia con signo como resultado, al contrario de lo que ocurría con las operaciones booleanas.

1.4.3. 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 función



Figura 1.5: Ejemplos de uso de los operadores afines

distancia con signo original. De esta forma podemos obtener operadores que de otra forma sería mucho más complicado implementar, como la torsión o el redondeo de los bordes de una primitiva [32].

Definición 1.20 (Operaciones Deformantes). Sea A una isosuperficie. La función μ define la isosuperficie para las siguientes operaciones.

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

En estas definiciones,

- $k \in \mathbb{R}_0^+$ 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. 1.19,
- $c: \mathbb{R}^3 \times \mathbb{R}^3 \times \mathbb{R}^3 \rightarrow \mathbb{R}^3$, $c(x, a, b)$ acota cada componente de x entre las de a y b .

Las únicas operaciones que nos proporcionan una función distancia con signo como resultado son el redondeo y la elongación. El resto es recomendable usarlas lo menos posible, pues las isosuperficies que generan pueden presentar fallos al ser renderizadas.

1.4.4. Operaciones de repetición

También podemos usar la técnica de cambiar el punto en el que evaluamos la función distancia para, en lugar de modificar la geometría original, añadir copias de la primitiva identificando varios puntos con uno que pertenezca a la isosuperficie. 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 1.21 (Operadores de Posicionamiento). Sea A una isosuperficie. La función μ define la isosuperficie para las siguientes operaciones.

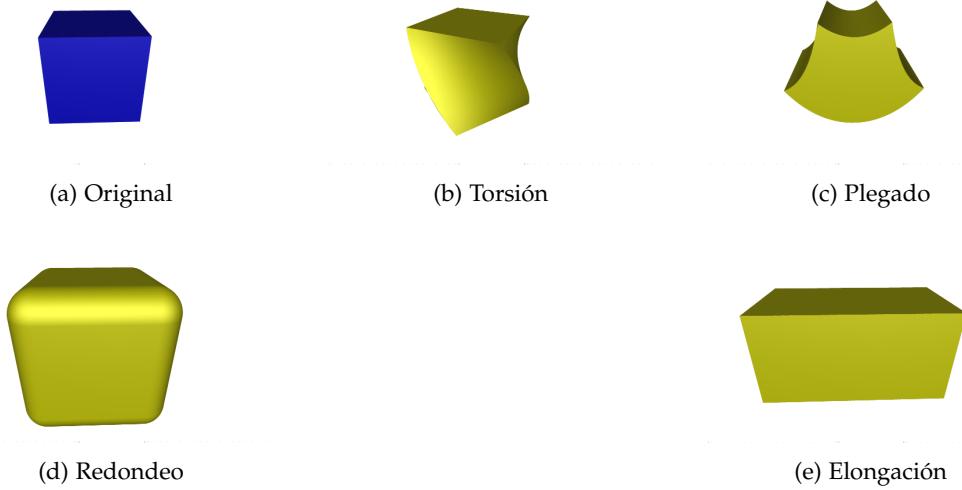


Figura 1.6: Ejemplos de uso de los operadores de deformación

- Simetrías sobre los ejes x, y, z :

$$\begin{aligned}\mu_{simX}(p) &= \text{sdf}_A(|x|, y, z), & \mu_{simY}(p) &= \text{sdf}_A(x, |y|, z), \\ \mu_{simZ}(p) &= \text{sdf}_A(x, y, |z|).\end{aligned}$$

- Simetrías sobre los planos xy, xz, yz :

$$\begin{aligned}\mu_{simXY}(p) &= \text{sdf}_A(|x|, |y|, z), & \mu_{simXZ}(p) &= \text{sdf}_A(|x|, y, |z|), \\ \mu_{simYZ}(p) &= \text{sdf}_A(x, |y|, |z|).\end{aligned}$$

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

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

- Repetición infinita:

$$\mu_{repInf}(p) = \text{sdf}_A\left((p + \frac{l}{2} \bmod l) - \frac{l}{2}\right).$$

En estas definiciones,

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

Las funciones obtenidas no son en general funciones distancia con signo. Esto ocurre en los siguientes casos.

- Al aplicar simetrías, cuando el objeto interseca el plano de simetría.

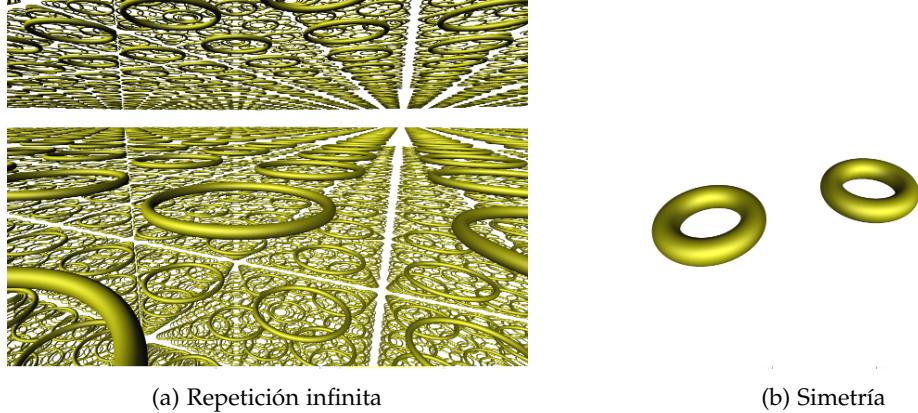


Figura 1.7: Ejemplos de uso de los operadores de repetición

- En el caso de la repetición infinita, cuando las dimensiones del objeto sean mayores o iguales a $l/2$.
- Siempre para la repetición finita, como consecuencia de usar la función máximo.

No obstante, este tipo de operaciones evidencia el potencial que tienen las funciones distancia con signo en cuanto a eficiencia a la hora de generar nuevas superficies, 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, o modelar solo una fracción de un objeto y generar el resto usando simetrías.

1.5. Obtención de una SDF a partir de ecuaciones implícitas

Empezábamos el capítulo diciendo que una de las representaciones más comunes de una superficie es a través de ecuaciones implícitas, pero hasta ahora nos hemos centrado en estudiar un subconjunto de esta familia. Si intentásemos aplicar el algoritmo de *raymarching* 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 ϕ cualquiera, obtener información aproximada de S_ϕ [33]. Esto nos será útil cuando no conocemos o no podamos calcular explícitamente la función distancia con signo de una superficie, pero sí su ecuación implícita.

Proposición 1.4. *Sea $\phi: \mathbb{R}^3 \rightarrow \mathbb{R}$ una función infinitamente diferenciable. Entonces*

$$|\text{sdf}_{S_\phi}(p)| \geq \frac{|\phi(p)|}{\|\nabla\phi(p)\|}.$$

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 $\|v\|$. Como ϕ es infinitamente diferenciable, podemos realizar el desarrollo de Taylor de ϕ centrado en p y evaluado en $q = p + v$:

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

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

$$0 = |\phi(p + v)| \approx |\phi(p) + \nabla\phi(p) \cdot v| \geq |\phi(p)| - |\nabla\phi(p) \cdot v| \geq |\phi(p)| - \|\nabla\phi(p)\| \cdot \|v\|,$$

donde hemos usado la desigualdad triangular y la linealidad del producto escalar. De esta expresión, finalmente deducimos que

$$\|v\| \geq \frac{|\phi(p)|}{\|\nabla\phi(p)\|}.$$

□

Este resultado solo nos proporciona una cota inferior de la función distancia (sin signo). En nuestro caso esto es suficiente, pues esta nos sigue permitiendo representar la frontera de S_ϕ , ya que proporciona una estimación conservadora de la distancia a ella. En su artículo [34], Pierre-Alain Fayolle describe un método para obtener una función distancia con signo asociada a una superficie implícita que representa de manera exacta su frontera. Para ello, la descompone

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

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

1.6. Implicitación de parametrizaciones racionales con bases de Gröbner

Ahora que sabemos representar las superficies generadas por una ecuación implícita cualquiera, nos proponemos ser capaces de representar también superficies definidas paramétricamente. Para ello fijaremos un cuerpo (anillo de división 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

$$\begin{cases} x_1 &= g_1(t_1, \dots, t_r), \\ &\vdots \\ x_n &= g_n(t_1, \dots, t_r), \end{cases} \tag{1.2}$$

donde g_i son polinomios de varias variables en A , obtener una ecuación implícita para V . En el caso que nos atañe $A^n = \mathbb{R}^3$, 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 [35], 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. Después veremos que este problema equivale a uno de pertenencia a un ideal y cómo resolverlo

usando la teoría de bases de Gröbner.

1.6.1. 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 1.22. Llamamos **monomio** en X a un producto de la forma

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

Lo denotaremos como X^α , y diremos que $\alpha = (\alpha_1, \dots, \alpha_n) \in \mathbb{N}^n$ es el **exponente** del monomio.

Definición 1.23. Definimos el **polinomio** con coeficientes en A a toda combinación lineal finita de monomios

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

Proposición 1.5. El conjunto de polinomios es un anillo conmutativo. En concreto, para

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

las operaciones internas del anillo son las siguientes.

- Suma inducida por la de A :

$$f + g = \sum_{\alpha \in \mathbb{N}^n} (a_\alpha + b_\alpha) X^\alpha.$$

- Producto inducido por la propiedad distributiva, siendo $X^\alpha X^\beta = X^{\alpha+\beta}$:

$$fg = \sum_{\alpha, \beta \in \mathbb{N}^n} (a_\alpha b_\beta) X^{\alpha+\beta}.$$

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

Proposición 1.6. $A[x_1, \dots, x_n] \cong A[x_1] \cdots A[x_n]$.

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

1. Reflexiva: $a \leq a$, para todo $a \in \Delta$.
2. Transitiva: si $a \leq b$ y $b \leq c$ entonces $a \leq c$, para todo $a, b, c \in \Delta$.
3. Antisimétrica: si $a \leq b$ y $b \leq a$ entonces $a = b$, para todo $a, b \in \Delta$.
4. Completitud: $a \leq b$ o $b \leq a$, para todo $a, b \in \Delta$.

Definición 1.25. Un **orden admisible** es un orden total \leq sobre \mathbb{N}^n cumpliendo

1. $(0, \dots, 0) \leq \alpha$, para todo $\alpha \in \mathbb{N}^n$,
2. si $\alpha \leq \beta$ entonces $\alpha + \gamma \leq \beta + \gamma$, para todo $\alpha, \beta, \gamma \in \mathbb{N}^n$.

Proposición 1.7. Todo orden admisible es un buen orden, esto es, todo subconjunto no vacío tiene un elemento mínimo. En particular satisface la condición de cadena descendente, y no existe ninguna secuencia infinita de monomios de $A[X]$ de la forma

$$m_1 \geq m_2 \geq \dots$$

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

Definición 1.26. Definimos el **orden lexicográfico** \leq_{lex} como

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

Definición 1.27. Dado $\omega \in \mathbb{N}^n$, un orden admisible \leq se dice **ω -graduado** cuando

$$\alpha \leq \beta \text{ implica que } \langle \alpha, \omega \rangle < \langle \beta, \omega \rangle,$$

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

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

Definición 1.28. Dado un orden admisible \leq , definimos el **orden ω -graduado asociado** como

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

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

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

Definición 1.29. Dado un monomio X^α , definimos su **grado** como

$$|\alpha| = \langle \alpha, (1, \dots, 1) \rangle = \alpha_1 + \dots + \alpha_n.$$

En el caso de un polinomio $f = \sum_{\alpha} a_{\alpha} X^{\alpha} \in A[X]$, diremos que su grado es el máximo $|\alpha|$ tal que el coeficiente a_{α} de f es distinto de cero, y lo notaremos como $\deg(f)$.

Definición 1.30. Definimos el **orden lexicográfico graduado inverso** $\leq_{\text{degrevlex}}$ como

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

Proposición 1.8. Sea \leq un orden admisible. Entonces \leq_{ω} es admisible.

Proposición 1.9. Los órdenes \leq_{lex} y $\leq_{\text{degrevlex}}$ 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 polinomios en varias variables.

Definición 1.31. 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:** $\text{lt}(f) = \text{lc}(f) \cdot \text{lm}(f)$.
- **Soporte:** $\text{supp}(f) = \{\alpha \in \mathbb{N}^n : a_{\alpha} \neq 0\}$.

Definición 1.32. Definimos el **exponente de un conjunto de polinomios** $F \subseteq A[X]$ al conjunto

$$\exp(F) = \{\exp(f) : f \in F\} \subseteq \mathbb{N}^n.$$

Proposición 1.10. Sea $\emptyset \neq I \leq A[X]$ y \leq un orden admisible. Entonces $\exp(I)$ es un ideal de \mathbb{N}^n .

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

Teorema 1.2 (Algoritmo de división). *Sea $F = \{f_1, \dots, f_s\} \subseteq A[X]$ y \leq un orden admisible. Todo polinomio $f \in A[X]$ se puede expresar como*

$$f = q_1 f_1 + \dots + q_s f_s + r,$$

donde $q_i, r \in A[X]$ y cumplen las siguientes condiciones.

- $\text{supp}(r) \cap (\exp(F) + \mathbb{N}^n) = \emptyset$.
- $r = 0$ o $\exp(r) \leq \exp(f)$.
- Para cada $1 \leq i \leq s$ se tiene que $q_i f_i = 0$ o $\exp(q_i f_i) \leq \exp(f)$.

Llamaremos a r el **resto** de dividir f por F , y lo notaremos $r = \text{rem}(f, F)$. Además, cuando $r = 0$ diremos que f reduce a 0, y escribiremos $f \xrightarrow{F} 0$.

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 hallar este polinomio simplemente se recorre el conjunto F hasta encontrar un f_i cumpliendo $\exp(f) \in \exp(f_i) + \mathbb{N}^n$. De esta forma podremos multiplicar f_i por cierto polinomio $g \in A[X]$ con $\exp(g) = \exp(f) - \exp(f_i)$ y coeficiente líder adecuado, para que $\exp(f_i g) = \exp(f)$ y al restárselo a f reduzcamos a cero su coeficiente líder. En el caso de que no se encuentre ningún f_i en estas condiciones, 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. La elección de un orden u otro también puede afectar al número de reducciones a cero necesarias para terminar el algoritmo. No obstante, en la siguiente sección veremos que la elección del orden no influye en el cálculo de bases de Gröbner.

Algorithm 1: División de polinomios en varias variables

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

```

 $p \leftarrow p$ 
 $[q_1, \dots, q_s] \leftarrow [0, \dots, 0]$ 
 $r \leftarrow 0$ 
while  $p \neq 0$  do
    divisorEncontrado  $\leftarrow \text{false}$ 
    for  $f_i \in F \text{ AND } \text{!divisorEncontrado}$  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  $\text{!divisorEncontrado}$  then
         $r \leftarrow r + \text{lt}(p)$ 
         $p \leftarrow p - \text{lt}(p)$ 
    end
end
return  $[r, q_1, \dots, q_s]$ 

```

1.6.2. Bases de Gröbner

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

Definición 1.33. Decimos que $\emptyset = I \subseteq A[X]$ es un **ideal** de $A[X]$ si

1. $a + b \in I$, para todo $a, b \in I$,

1 Fundamentos matemáticos de las SDFs

2. $af \in I$, para todo $a \in A$, para todo $f \in A[x]$.

En ese caso escribiremos $I \leq A$.

Proposición 1.11. *Dados los ideales $I, J \leq A[X]$, son también ideales de $A[X]$:*

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

Podemos calcular estos ideales usando sus conjuntos de generadores.

Definición 1.34. Dado $F = \{f_1, \dots, f_s\} \subseteq A[X]$, el **ideal generado** por F es

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

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

Proposición 1.12. *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. $I \cap J = \langle tF, (1-t)G \rangle \cap A[x_1, \dots, x_n]$, con t una variable auxiliar distinta a x_1, \dots, x_n .

Dado que, a priori, un ideal podría estar generado por un número infinito de polinomios, el hecho de conocer un conjunto de generadores nos permite trabajar y extraer propiedades de él de forma mucho más cómoda, más aún si este conjunto es finito. Este hecho nos hace preguntarnos si podemos extraer un conjunto de generadores finito de cualquier ideal. La respuesta a esta pregunta nos la proporciona el teorema de la base de Hilbert.

Definición 1.35. Diremos que $I \leq A[X]$ es un **ideal monomial** si existe un conjunto $B \subseteq \mathbb{N}^n$ tal que I está conformado por todos los polinomios que son sumas finitas de la forma

$$\sum_{\alpha \in B} h_\alpha X^\alpha, \text{ donde } h_\alpha \in A[X].$$

Usaremos la notación $I = \langle X^\alpha : \alpha \in B \rangle$.

Definición 1.36. Sea $I \leq A[X]$ y \leq un orden admisible. Denotamos el conjunto de los términos líder de I como

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

Lema 1.3. *Sea $\{0\} \neq I \leq A[X]$ y \leq un orden admisible. Entonces:*

1. $\text{lt}(I)$ es un ideal monomial.
2. Existen $g_1, \dots, g_t \in I$ tal que $\text{lt}(I) = \langle \text{lt}(g_1), \dots, \text{lt}(g_t) \rangle$.

Lema 1.4. *Sea $I = \langle X^\alpha : \alpha \in B \rangle$ un ideal monomial. Entonces*

$$X^\beta \in I \text{ si y solo si es divisible por } X^\alpha \text{ para algún } \alpha \in B.$$

Teorema 1.3 (Base de Hilbert). *Todo ideal $I \subseteq A[X]$ tiene un conjunto de generadores finito.*

Demostración. Si $I = \{0\}$, podemos tomar como conjunto generador $G = \{0\}$. En caso contrario fijamos un orden admisible, y usando el [Lema 1.3](#) obtenemos que existen $g_1, \dots, g_t \in I$ tal que $\text{lt}(I) = \langle \text{lt}(g_1), \dots, \text{lt}(g_t) \rangle$. Veamos que $I = \langle g_1, \dots, g_t \rangle$. La inclusión \supseteq es evidente, pues cada g_i pertenece a I . Para la otra, tomamos $f \in I$ y lo dividimos por $\{g_1, \dots, g_t\}$, obteniendo su expresión

$$f = q_1 g_1 + \dots + q_t g_t + r,$$

donde ningún término del polinomio r es divisible por ningún elemento de $\{\text{lt}(g_1), \dots, \text{lt}(g_t)\}$. Vamos a comprobar que de hecho $r = 0$. Por contradicción, supongamos que

$$r = f - q_1 g_1 - \dots - q_t g_t \neq 0.$$

De esta expresión obtenemos que $\text{lt}(r) \in \langle \text{lt}(I) \rangle = \langle \text{lt}(g_1), \dots, \text{lt}(g_t) \rangle$, de donde por el [Lema 1.4](#) tendríamos que $\text{lt}(r)$ es divisible por algún $\text{lt}(g_i)$, lo que contradice la definición de resto dada en el [Teorema 1.2](#). Confirmamos por tanto que $r = 0$, luego

$$f = q_1 g_1 + \dots + q_t g_t \in \langle g_1, \dots, g_t \rangle, \text{ de donde } I \subseteq \langle g_1, \dots, g_t \rangle. \quad \square$$

En la demostración anterior hemos visto que podemos obtener un conjunto generador finito $G = \{g_1, \dots, g_t\}$ para cualquier ideal $I \subseteq A[X]$ que además cumpla la propiedad $\text{lt}(I) = \langle \text{lt}(g_1), \dots, \text{lt}(g_t) \rangle$. Le daremos a este tipo especial de conjuntos generadores el siguiente nombre.

Definición 1.37. Dado $I \subseteq A[X]$ y \leq un orden admisible, diremos que $G = \{g_1, \dots, g_t\} \subseteq I$ es una **base de Gröbner** para I si

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

Proposición 1.13. *Sea $I \subseteq A[X]$, \leq un orden admisible y G una base de Gröbner suya. Entonces $\exp(I) = \exp(G) + \mathbb{N}^n$.*

Demostración. Sea $G = \{g_1, \dots, g_t\} \subseteq I$. Por definición de base de Gröbner, para todo $f \in I$ se tiene que

$$\text{lt}(f) \in \langle \text{lt}(g_1), \dots, \text{lt}(g_t) \rangle, \text{ luego } \text{lt}(f) = a_1 \text{lt}(g_1) + \dots + a_t \text{lt}(g_t), \text{ donde } a_1, \dots, a_t \in A.$$

De esta expresión obtenemos que $\exp(\text{lt}(g_i)) \leq \exp(\text{lt}(f))$ para cada $i \in \{1, \dots, t\}$, luego $\exp(g_i) \leq \exp(f)$, obteniendo el resultado. \square

Proposición 1.14. *Sea $I \subseteq A[X]$ y \leq un orden admisible. Para cualquier $f \in A[X]$ existe un único $r \in A[X]$ tal que*

1. $\text{supp}(r) \cap \exp(I) = \emptyset$,
2. $f - r \in I$.

Demostración. Sea $I \subseteq A[X]$ y $G = \{g_1, \dots, g_t\}$ una base de Gröbner suya. Como consecuencia del [Teorema 1.2](#) tenemos que el resto $r = \text{rem}(f, G)$ cumple:

1. $\text{supp}(r) \cap (\exp(G) + \mathbb{N}^n) = \emptyset$. Como $\exp(I) = \exp(G) + \mathbb{N}^n$ por la [Proposición 1.13](#), esto equivale a que $\text{supp}(r) \cap \exp(I) = \emptyset$.

2. $f - r = q_1g_1 + \cdots + q_tg_t$, luego pertenece a I .

Solo queda por comprobar la unicidad. Sean $r, r' \in A[X]$ en las condiciones del enunciado, y $g, g' \in I$ tales que

$$f = g + r = g' + r', \text{ de donde } r - r' = r - f + f - r' = -g + g' \in I.$$

Supongamos que $r \neq r'$. Entonces,

$$\exp(r - r') \in \exp(I) \text{ y } \exp(r - r') \in \text{supp}(r - r') \subseteq \text{supp}(r) \cup \text{supp}(r'),$$

de donde

$$\emptyset \neq (\text{supp}(r) \cup \text{supp}(r')) \cap \exp(I) = (\text{supp}(r) \cap \exp(I)) \cup (\text{supp}(r') \cap \exp(I)) = \emptyset \cup \emptyset,$$

donde en el último paso hemos usado la primera condición del lema. \square

Observación 1.3. Dado que $\exp(I)$ depende del orden elegido, también lo hará el polinomio r .

Corolario 1.2. Sea $I \leq A[X]$, \leq un orden admisible y G una base de Gröbner suya. Entonces,

$$f \in I \text{ si y solo si } \text{rem}(f, G) = 0.$$

El concepto de base de Gröbner es quizás el más importante tratado en este trabajo, pues permite abordar gran variedad de problemas sobre ideales de forma computacional, como el de pertenencia a un ideal o el de implicación, como veremos próximamente. Como es de esperar, a la hora de trabajar con bases de Gröbner será deseable que estas tengan el menor número de elementos posible. Veamos que no sólo siempre podemos encontrar alguna base de Gröbner asociada al ideal, sino que esta será reducida y única, sin depender del orden elegido.

Definición 1.38. Sea $I \leq A[X]$ y \leq un orden admisible. Diremos que G es una **base de Gröbner reducida** para I si es una base de Gröbner suya y para todo $g \in G$ se cumple

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

Es decir, una base de Gröbner $G = \{g_1, \dots, g_t\}$ será reducida si ningún monomio de $\text{supp}(g_i)$ es divisible por $\text{lm}(g_j)$ para todo $i \neq j$. Esto equivale a que

$$\text{rem}(g, G \setminus \{g\}) = g, \text{ para todo } g \in G.$$

Definición 1.39. Diremos que un subconjunto $\emptyset \neq M \subseteq \mathbb{N}^n$ es un **ideal de \mathbb{N}^n** si $M = M + \mathbb{N}^n$, y lo notaremos $M \leq \mathbb{N}^n$. Cuando para algún $F \subseteq M$ se tenga que $M = F + \mathbb{N}^n$ diremos que M está generado por F .

Definición 1.40. Sea $M \leq \mathbb{N}^n$. Decimos que A es un **conjunto generador minimal** de M si

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

Lema 1.5. Todo ideal $M \leq \mathbb{N}^n$ tiene un único conjunto generador minimal.

Teorema 1.4. Todo ideal I admite una única base de Gröbner reducida para un orden admisible dado.

Demostración. **Existencia** Sea $G \subseteq I$ cumpliendo que $\exp(G)$ es un conjunto generador minimal de $\exp(I)$, que sabemos que existe por el [Lema 1.5](#). Sea $g \in G$ y $r = \text{rem}(g, G \setminus \{g\})$. Tenemos que

$$\exp(g) \notin \exp(G \setminus \{g\}) + \mathbb{N}^n, \text{ luego } \exp(g) = \exp(r).$$

Por otro lado, ya que $g - r \in \langle G \setminus \{g\} \rangle \subseteq I$, se tiene que $r \in I$. Como además $\exp(G) = \exp((G \setminus \{g\}) \cup \{r\})$, obtenemos que $G' = (G \setminus \{g\}) \cup \{r\}$ es una nueva base de Gröbner de I cumpliendo que $\text{supp}(r) \cap (\exp(G' \setminus \{r\}) + \mathbb{N}^n) = \emptyset$. Aplicando este procedimiento a cada elemento de G y dividiéndolo por su coeficiente líder obtenemos una base reducida de I .

Unicidad Sean G_1, G_2 dos bases reducidas de I . Por el [Lema 1.5](#) sabemos que $\exp(G_1) = \exp(G_2)$, luego dado cualquier $g_1 \in G_1$, existe un único $g_2 \in G_2$ tal que $\exp(g_1) = \exp(g_2)$. Por otro lado, teniendo en cuenta que $g_2 - g_1 \in I$ por ser ambos elementos del ideal y $\text{rem}(g_1 - g_2, G_1) = 0$ por el [Corolario 1.2](#), se cumple

1. $\text{supp}(g_1 - g_2) \subseteq (\text{supp}(g_1) \cup \text{supp}(g_2)) \setminus \{\exp(g_1)\},$
2. $\text{supp}(g_i) \setminus \{\exp(g_i)\} \cap (\exp(G_i) + \mathbb{N}^n) = \emptyset, \text{ para cada } i \in \{1, 2\},$

de donde

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

Concluimos entonces usando la [Proposición 1.14](#) que $\text{rem}(g_1 - g_2, G_1) = g_1 - g_2$, de forma que $g_1 = g_2$. Hemos demostrado que cada elemento de una base está en la otra, luego $G_1 = G_2$. \square

Terminamos la sección obteniendo un algoritmo para calcular la base de Gröbner reducida para un ideal dado un conjunto de generadores G suyo. Aplicar de forma directa la definición de base de Gröbner sería inviable, así que en su lugar introduciremos una caracterización de las bases de Gröbner más abordable desde el punto de vista computacional.

Definición 1.41. Dados $\alpha, \beta \in \mathbb{N}^n$, definimos los siguientes términos.

- **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 1.42. 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 1.5 (Primer Criterio de Buchberger). Sean $I \leq A[X]$, \leq un orden admisible y $G = \{g_1, \dots, g_t\}$ un conjunto de generadores de I . Entonces:

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

El algoritmo que usaremos para el cálculo de la base de Gröbner se basará en este criterio. Sin embargo, antes de presentarlo estudiamos dos criterios adicionales [36, 37] que lo harán más eficiente descartando S-polinomios antes de tener que comprobar si reducen a cero, ahorrando el cómputo de numerosas divisiones.

Definición 1.43. Sea $f = \sum_{\alpha} a_{\alpha} X^{\alpha}$ un polinomio cuyo monomio líder es $X^{\alpha^{(k)}}$. Definimos el **segundo monomio líder** de f como el monomio $X^{\alpha^{(i)}}$ de f tal que

$$X^{\alpha^{(i)}} \geq X^{\alpha^{(j)}}, \text{ para todo } j \neq k.$$

Lo denotaremos como $\text{sm}(f)$.

Teorema 1.6 (Criterios de Buchberger). *Sean $I \leq A[X]$, \leq un orden admisible, $G \subseteq A[X]$ 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. Existe un $f \in G$ tal que $\text{lm}(f) \mid \text{lcm}(g_1, g_2)$ y además
 - a) algún $S(g_i, f) \xrightarrow{G} 0$ o
 - b) $\text{lm}(f) \mid \frac{\text{lm}(g_j)}{\text{gcd}(\text{exp}(g_1), \text{exp}(g_2))}$ 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 todos los criterios anteriores obtenemos el [Algoritmo 2](#) para calcular la base de Gröbner de cualquier ideal. La salida de este no es una base reducida, pero la demostración del [Teorema 1.4](#) nos proporciona un método para reducir una base cualquiera a la minimal asociada. En el [Algoritmo 3](#) mostramos este procedimiento.

Algorithm 2: Algoritmo de Buchberger optimizado

Data: conjunto de generadores $F = [f_1, \dots, f_s]$ del ideal I

Result: base de Gröbner de I

$G \leftarrow F;$

repeat

```

     $G' \leftarrow G;$ 
    for each pair  $\{f, g\} \subseteq G'$  do
        if !Criterio 1( $f, g$ ) AND !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  $G$ 

```

Ya somos capaces de obtener una base de Gröbner reducida de cualquier ideal dado un conjunto de generadores suyo, pero en este proceso se toma una decisión que aún no hemos

Algorithm 3: Reducción de base de Gröbner

Data: G base de Gröbner de un ideal I **Result:** base de Gröbner reducida de I $G \leftarrow F;$ $G' \leftarrow \emptyset;$ **foreach** $g \in G$ **do** $g \leftarrow g / \text{lc}(g);$ $r \leftarrow \text{rem}(g, G \setminus \{g\});$ **if** $r \neq 0$ **then** $g \leftarrow r;$ $G' \leftarrow G' \cup \{r\};$ **end****end****return** G'

discutido: cómo se eligen las parejas $\{f, g\}$ del [Algoritmo 2](#). Uno de los métodos más usados es la conocida como **estrategia normal**, debido a su simpleza y haber probado ser de las que completan más rápido el algoritmo. Esta consiste en tomar el par f, g cuyo $\text{lcm}(f, g)$ sea del menor grado posible según el orden admisible usado. Vemos que la elección de un orden u otro puede involucrar un cambio en el número de reducciones a cero, y en consecuencia el aumento de operaciones realizadas, pero en ningún caso afectará al resultado final.

1.6.3. Teorema de implicitación

Empezábamos la sección diciendo que el problema de implicitación equivalía al de pertenencia a un ideal, como veremos a continuación, pero nuestro objetivo en definitiva es el de resolver el sistema de ecuaciones (1.5). Es decir, estamos interesados en conocer el conjunto de ceros comunes a todos los polinomios del sistema, lo que motiva la siguiente definición.

Definición 1.44. Dado $F \subseteq A[X]$, llamamos **variedad afín** asociada a F al conjunto

$$\mathbb{V}(F) = \{(a_1, \dots, a_n) \in A^n : f(a_1, \dots, a_n) = 0 \text{ para todo } f \in F\}.$$

Proposición 1.15. Sean $F, G \subseteq A[X]$ y $\mathbb{V}(F), \mathbb{V}(G)$ las variedades afines asociadas. 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 1.16. Sean los ideales $I, J \leq A[X]$. Entonces $\mathbb{V}(I \cap J) = \mathbb{V}(I) \cup \mathbb{V}(J)$.

Definición 1.45. Sea $B \subseteq A^n$. Definimos el **ideal asociado** a B como

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

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

Definición 1.46. Dado $I \leq A[x_1, \dots, x_n]$, definimos su **ideal de l -eliminación** como

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

Definición 1.47. Decimos que un orden admisible \leq en \mathbb{N}^n es un **orden de l -eliminación** si

$$\beta \leq \alpha \text{ implica } \beta \in \mathbb{N}_l^n \text{ para todo } \alpha \in \mathbb{N}_l^n \text{ y para todo } \beta \in \mathbb{N}^n,$$

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

Teorema 1.7 (Eliminación). *Sea $I \leq A[x_1, \dots, x_n]$ y G una base de Gröbner suya respecto a un orden \leq de l -eliminación. Entonces, una base de Gröbner para I_l viene dada por*

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

Volviendo al problema de implicación, recordamos que los polinomios de los cuales tenemos que obtener la variedad afín asociada son las ecuaciones paramétricas del sistema

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

Si escribimos $g_i = f_i/q_i$ con $f_i, q_i \in A[t_1, \dots, t_r]$ para $i = 1, \dots, n$, 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 en el caso de que $q_i = 1$ para cada $i \in \{1, \dots, r\}$.

Teorema 1.8 (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 1.9 (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 la aplicación

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

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 1.4. 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 este será una potencia de la ecuación implícita buscada, 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 del teorema anterior. El hecho de que no obtengamos la potencia exacta de la ecuación implícita no es problema, pues nos basta conocer donde se anula para poder representar la frontera de la superficie que genera. Sin embargo, no tenemos asegurado que vaya a haber un único generador del ideal, en cuyo caso contrario 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 1.48. 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 1.17. Sea $I \leq A[X]$. Entonces \sqrt{I} es un ideal y contiene a I .

Definición 1.49. Decimos que un ideal $I \leq A[X]$ es **radical** si $\sqrt{I} = I$.

Proposición 1.18. Sea $B \subseteq A^n$. Entonces $\mathbb{I}(B)$ es un ideal radical.

Teorema 1.10 (Nullstellensatz fuerte). Si A es algebraicamente cerrado, dado un ideal $I \leq A[X]$ se cumple

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

Proposición 1.19. Sea $I \leq A[X]$ y $f \in A[X]$. Entonces

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

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. Además, calculando el radical obtendríamos directamente la potencia exacta de la ecuación implícita que nos da el Teorema 1.9. Sin embargo, el cálculo del radical o su número de elementos es en general muy complicado, luego lo que haremos en la práctica será simplemente aplicar el método mostrado en el Teorema 1.9 y comprobar

si efectivamente se obtiene un único generador, en cuyo caso bastará obtener la función distancia con signo aproximada de dicho generador como se estudió en la [Sección 1.5](#) para así poder representar la superficie mediante *raymarching*. En caso de que se obtenga más de un generador concluiremos que no podemos realizar la implicitación.

Hay otros métodos que permiten abordar el problema de eliminación de variables, y en consecuencia el de implicitación. Uno especialmente interesante es el uso de resultantes, pues en casos específicos puede simplificar mucho la obtención de la ecuación implícita. En la siguiente sección estudiaremos como solucionar el problema de implicitación usando resultantes de forma básica.

1.7. Implicitación de parametrizaciones racionales con resultantes

En la sección anterior, para resolver el problema de implicitación hemos tratado de resolver el sistema de ecuaciones (1.6.3) buscando los puntos que satisfacen cada una de las ecuaciones, pero existen vías alternativas. En esta sección veremos cómo usar resultantes para la implicitación de superficies paramétricas racionales. Para ello, empezaremos introduciendo el concepto de resultado y estudiando sus versiones más sencillas [35], para posteriormente abordar el problema de implicitación en un ambiente mucho más general basándonos en el trabajo de Eng-Wee Chionh y Ronald N. Oldman [38].

1.7.1. Conceptos básicos y caso de una variable

Una de las primeras ideas que se pueden ocurrir para resolver el sistema (1.6.3) de forma alternativa es que si todos sus polinomios tienen algún factor común $h \in A[t_1, \dots, t_r]$ no trivial, esto es, cumpliendo $\deg(h) > 1$, entonces

$$h(t_1, \dots, t_r) = 0 \text{ implica } g_i(t_1, \dots, t_r) = 0 \text{ para cada } i \in \{1, \dots, n\}.$$

En concreto, las raíces que coinciden y se anulan simultáneamente con las de los polinomios del sistema son las del polinomio h , luego si $h = \gcd(g_1, \dots, g_n)$ la implicación anterior se convierte en una equivalencia. Esta idea motiva la siguiente definición [39].

Definición 1.50. Sean $f_1, \dots, f_n \in A[t_1, \dots, t_r]$. Definimos el **resultado** de f_1, \dots, f_n como el polinomio $\text{Res}_{f_1, \dots, f_n} \in A[t_1, \dots, t_r]$ tal que

$$\text{Res}_{f_1, \dots, f_n}(t_1, \dots, t_r) = 0 \text{ si y solo si } f_1(t_1, \dots, t_r) = \dots = f_n(t_1, \dots, t_r) = 0.$$

Cuando no haya lugar a confusión omitiremos el subíndice.

Hemos visto que la forma más directa de obtener el resultado de un sistema de ecuaciones es a través del máximo común divisor, pero para a medida que el número de polinomios crece, eso conllevaría realizar un gran número divisiones, que ya hemos visto que son caras computacionalmente. El cálculo del resultado en varias variables no es en general nada simple, pero bajo ciertas condiciones puede llegar a serlo, como muestra el siguiente resultado en el caso de que se tenga el mismo número de variables que de polinomios y estos tengan una determinada forma.

Teorema 1.11 (Teorema fundamental de eliminación). *Sean $f_1, \dots, f_n \in A[t_1, \dots, t_n]$ lineales, esto es, de la forma*

$$f_i(t_1, \dots, t_n) = a_1^{(i)}t_1 + \dots + a_r^{(i)}t_n + b, \text{ para cada } i \in \{1, \dots, n\}.$$

Entonces su resultante es

$$\text{Res}(t_1, \dots, t_n) = \det \begin{pmatrix} a_1^{(1)} & a_2^{(1)} & \dots & a_r^{(1)} \\ \vdots & \vdots & & \vdots \\ a_1^{(n)} & a_2^{(n)} & \dots & a_n^{(n)} \end{pmatrix}.$$

A continuación estudiamos en detalle un método para obtener el resultado en el caso más sencillo posible, el de una única variable y dos polinomios [35]. En este entorno, el método más común es el de la matriz de Sylvester, la cual deduciremos a partir de la búsqueda de un divisor común de estos polinomios.

Lema 1.6. *Sean $f, g \in A[x]$ polinomios tal que $\deg(f) = l$ y $\deg(g) = m$, con $l, m \in \mathbb{N} \setminus \{0\}$. Entonces f y g tienen un factor común no trivial si y solo si existen otros polinomios $A, B \in A[x]$ tal que*

1. A y B son no nulos,
2. $\deg(A) \leq m - 1$ y $\deg(B) \leq l - 1$,
3. $Af + Bg = 0$.

La cuestión a resolver ahora es la de la existencia de los polinomios A y B . Para ello escribimos

$$\begin{aligned} A &= \sum_{i=0}^{m-1} u_{m-i-1}x^i, & B &= \sum_{j=0}^{l-1} v_{l-j-1}x^j, \\ f &= \sum_{k=0}^l c_{l-k}x^k, & g &= \sum_{h=0}^m d_{m-h}x^h, \text{ con } c_l, d_h \neq 0, \end{aligned} \tag{1.3}$$

donde trataremos a los coeficientes u_i y v_j en A como incógnitas para $i \in \{0, \dots, m-1\}$ y $j \in \{0, \dots, l-1\}$. Para encontrarlos impondremos que se cumpla la tercera condición del Lema 1.6:

$$Af + Bg = 0. \tag{1.4}$$

Sustituyendo las expresiones de (1.3) en la igualdad (1.4) y comparando los coeficientes de cada potencia de x , obtenemos el siguiente sistema de incógnitas u_i, v_j y coeficientes c_k, d_h :

$$\begin{array}{lll} c_0u_0 & +d_0v_0 & = 0, \\ c_1u_0 + c_0u_1 & +d_1v_0 + d_0v_1 & = 0, \\ \ddots & \ddots & \vdots \\ c_lu_{m-1} + d_mv_{l-1} & & = 0. \end{array}$$

Como tenemos $l + m$ incógnitas y ecuaciones, sabemos por álgebra lineal que el sistema tendrá alguna solución no nula si y solo si la matriz de coeficientes asociada tiene determinante

igual a cero.

Definición 1.51. Dados dos polinomios $f, g \in A[x]$ no nulos de la forma

$$f = \sum_{k=0}^l c_{l-k}x^k, \quad g = \sum_{h=0}^m d_{m-h}x^h, \quad \text{con } c_l, d_h \neq 0 \text{ y } l, m > 0,$$

definimos la **matriz de Sylvester** de f y g respecto a x como

$$\text{Syl}(f, g, x) = \begin{pmatrix} c_0 & & & d_0 & & \\ \vdots & \ddots & & \vdots & \ddots & \\ c_l & & c_0 & d_m & & d_0 \\ & \ddots & \vdots & & \ddots & \vdots \\ & & c_l & & & d_m \end{pmatrix},$$

donde el resto de posiciones son cero.

Proposición 1.20. En el contexto de la definición anterior, se tiene que

$$\text{Res}_{f,g}(x) = \det(\text{Syl}(f, g, x)).$$

Podemos concluir por tanto que siempre podemos encontrar los polinomios A y B en las condiciones del [Lema 1.6](#). Además, estos cumplen el siguiente hecho.

Proposición 1.21. Dados $f, g \in A[x]$ no nulos, existen polinomios $A, B \in A[x]$ tal que

$$Af + Bg = \text{Res}_{f,g}(x).$$

De hecho, si $\deg(f) > 0$ o $\deg(g) > 0$, entonces A, B son polinomios con coeficientes enteros en los coeficientes de f y g .

La demostración de este resultado se puede consultar en [35], y de ella se puede obtener también la siguiente propiedad interesante.

Proposición 1.22. Dados $f, g \in A[x]$ no nulos, existen polinomios $\tilde{A}, \tilde{B} \in A[x]$ de la forma

$$\tilde{A} = \frac{A}{\text{Res}(f, g, x)} \quad \text{y} \quad \tilde{B} = \frac{B}{\text{Res}_{f,g}(x)},$$

que cumplen

$$\tilde{A}f + \tilde{B}g = 1.$$

1.7.2. Resultante auxiliar

Ahora que tenemos una idea básica de como trabajar con resultantes, veamos como podemos usarlos para realizar la implicación de una superficie parametrizada racionalmente [38]. Sabemos ya por el [Teorema 1.8](#) que todo parametrización racional satisface un conjunto de ecuaciones implícitas. Sin embargo, a diferencia de los resultados enunciados en este teorema, en el que trabajábamos con un número arbitrario de variables, en esta ocasión vamos a ceñirnos al caso de tres variables. Hacemos esto debido a que este será el ambiente en el que

trabajaremos en la práctica y, como ya hemos mencionado, a medida que se incrementa el número de variables y ecuaciones, trabajar con resultantes es cada vez más complicado. Así, supondremos que tenemos una parametrización de la forma

$$x = \frac{\xi(r, s, t)}{\omega(r, s, t)}, \quad y = \frac{\eta(r, s, t)}{\omega(r, s, t)}, \quad z = \frac{\zeta(r, s, t)}{\omega(r, s, t)}, \quad (1.5)$$

donde $\omega(r, s, t) \neq 0$, y supondremos además que el máximo común divisor de estos polinomios es constante, de forma que trabajamos con una **parametrización propia**. Normalmente las superficies en \mathbb{R}^3 vienen parametrizadas por los parámetros s y t , pero vamos a realizar un paso previo en el que homogeneizamos cada polinomio, es decir, hacemos que todos los monomios de cada polinomio tengan el mismo grado. Esto es sencillo, ya que para todo polinomio $f \in A[x_1, \dots, x_n]$ siempre puede obtenerse su homogeneizado ${}^h f$ añadiendo una variable adicional x_0 [4o] de la forma

$${}^h f(x_0, x_1, \dots, x_n) = x_0^{\deg(f)} f\left(\frac{x_1}{x_0}, \dots, \frac{x_n}{x_0}\right).$$

Este es el motivo de que aparezca la variable adicional r . Una vez homogeneizados los polinomios de la parametrización, podemos obtener el **sistema de ecuaciones auxiliar**

$$\begin{cases} \xi(r, s, t) - x\omega(r, s, t) = 0, \\ \eta(r, s, t) - y\omega(r, s, t) = 0, \\ \zeta(r, s, t) - z\omega(r, s, t) = 0, \end{cases} \quad (1.6)$$

cuya existencia del resultado asociado tenemos asegurada por el [Teorema 1.11](#).

Definición 1.52. Llamamos **resultado auxiliar** al resultado asociado a los polinomios del sistema (1.6), y lo denotaremos $\text{Res}_{\text{aux}}(x, y, z)$.

Proposición 1.23. *El resultado auxiliar es o bien un polinomio no nulo o bien idénticamente cero.*

Los siguientes resultados muestran la relación que tiene el resultado auxiliar con el problema de implicitación.

Proposición 1.24. *Toda parametrización racional está contenida en una única superficie dada por un único polinomio irreducible.*

Proposición 1.25. *Si el resultado auxiliar no es idénticamente nulo, entonces la superficie generada por la ecuación $\text{Res}_{\text{aux}}(x, y, z) = 0$ y la dada por la parametrización (1.5) representan el mismo conjunto de puntos.*

Teorema 1.12. *Si $\text{Res}_{\text{aux}}(x, y, z)$ no es idénticamente nulo, entonces*

$$\text{Res}_{\text{aux}}(x, y, z) = f^l(x, y, z), \text{ para algún } f \in A[x, y, z] \text{ irreducible y } l \in \mathbb{N} \setminus \{0\}.$$

Demostración. Tomamos $A = \mathbb{C}$. Dado que $\mathbb{C}[x, y, z]$ es un dominio de factorización única, debe darse

$$\text{Res}_{\text{aux}}(x, y, z) = f_1^{l_1}(x, y, z) \cdots f_r^{l_r}(x, y, z),$$

donde todo f_i es irreducible y distinto a los demás para cada $i \in \{1, \dots, r\}$. Sea f el polinomio dado por la [Proposición 1.24](#) que genera la superficie racional que contiene a la dada por la

parametrización. Por la [Proposición 1.25](#), sabemos entonces que

$$\{(a, b, c) : f_1^{l_1}(a, b, c) \cdots f_r^{l_r}(a, b, c) = 0\} \subseteq \{(a, b, c) : f(a, b, c) = 0\}.$$

Dado que f es irreducible, debe darse que $r = 1$ y $f = f_1$, concluyendo la demostración. \square

Al igual que ocurría al usar bases de Gröbner, solo hemos obtenido una potencia de la ecuación implícita, lo cual nos es suficiente. Sin embargo, en este caso es mucho más fácil obtener de forma precisa la ecuación implícita de la superficie dada por la parametrización racional.

Teorema 1.13. *Si $\text{Res}_{\text{aux}}(x, y, z)$ no es idénticamente nulo, entonces la ecuación implícita asociada a la parametrización racional es*

$$f(x, y, z) = \frac{\text{Res}_{\text{aux}}(x, y, z)}{\gcd\left(\text{Res}_{\text{aux}}(x, y, z) \cdot \frac{\partial \text{Res}_{\text{aux}}}{\partial u}(x, y, z)\right)},$$

donde u es cualquiera de las variables x, y, z que aparezcan en Res_{aux} .

Demostración. Usando el [Teorema 1.12](#) sabemos que $\text{Res}_{\text{aux}} = f^l$, de donde

$$\frac{\partial \text{Res}_{\text{aux}}}{\partial u}(x, y, z) = l f^{l-1} \frac{\partial f}{\partial u}(x, y, z).$$

Dado que f es irreducible y de grado mayor que $\frac{\partial f}{\partial u}(x, y, z)$, estos no tienen ningún factor común, y por consiguiente

$$\gcd\left(\text{Res}_{\text{aux}}, \frac{\partial \text{Res}_{\text{aux}}}{\partial u}\right) = f^{l-1},$$

de donde se obtiene el resultado. \square

Observamos que aunque tengamos que calcular un máximo común divisor, solo hay dos polinomios involucrados, lo cual simplifica los cálculos notablemente en comparación al caso más polinomios. Terminamos la sección viendo bajo qué circunstancias podemos asegurar que $\text{Res}_{\text{aux}} \neq 0$, de forma que estaremos seguros de que la parametrización es representada por una única ecuación implícita. Para ello necesitamos presentar el concepto de punto base.

Definición 1.53. Llamamos **puntos base** de una parametrización racional de la forma (1.5) a la intersección de las curvas planas

$$\xi(r, s, t) = 0, \quad \eta(r, s, t) = 0, \quad \zeta(r, s, t) = 0, \quad \omega(r, s, t) = 0.$$

Teorema 1.14. *El resultante auxiliar es idénticamente nulo si y solo si la parametrización tiene puntos base.*

Todos los resultados que hemos presentado tenían como hipótesis la no nulidad del resultante auxiliar, de forma que solo hemos aprendido a trabajar en la ausencia de puntos base. Para el estudio del cálculo de resultantes cuando existan puntos base se pueden consultar [41, 42]. No obstante, los resultados vistos siguen siendo potentes, pudiendo llegar a agilizar de forma muy notable los cálculos en función del problema concreto con el que estemos

trabajando. En particular, el algoritmo de cálculo y reducción de una base de Gröbner va añadiendo sobre la marcha nuevos polinomios con los que realizar operaciones, sin saber exactamente cuándo acabará. Es decir, ese algoritmo se basa en buscar y comprobar, lo que requiere un mayor número de operaciones que el algoritmo propuesto por Eng-Wee Chionh y Ronald N. Goldman [38], que conoce de antemano el número de polinomios con los que va a trabajar y produce en general un número de operaciones mucho menor. Además, en caso de que necesitemos obtener la ecuación implícita exacta, el tener que calcular un radical traerá muchas complicaciones, en caso de que sea posible. Por estos motivos el uso de resultantes es la vía más común para la resolución del problema de implicitación. Acabamos el capítulo con un ejemplo práctico de uso de bases de Gröbner y resultantes para un problema de implicitación sencillo.

Ejemplo 1.3. El plano $\Pi = \{(x, y, z) \in \mathbb{R}^3 : -x + y + z = 0\}$ tiene la siguiente parametrización racional:

$$\begin{cases} x = s + t, \\ y = s, \\ z = t. \end{cases}$$

Usando bases de Gröbner y aplicando el [Teorema 1.8](#), tenemos que calcular una base del ideal

$$I = \langle x - (s + t), y - (s), z - (t) \rangle \leq \mathbb{Q}[s, t, x, y, z].$$

La versión más básica del algoritmo de Buchberger de SageMath implementada en el paquete `sage.rings.polynomial.toy_buchberger` realiza ocho reducciones a cero, mientras que la versión presentada en el [Algoritmo 2](#) realiza seis, reduciendo este número a tan solo una si además se usan los criterios del [Teorema 1.6](#). En cualquier caso, obtenemos la base

$$B = \{-x + y + z, -s + y, -s - t + x, -t + z\},$$

de donde el ideal de 2-eliminación de I sería

$$J = I \cap \mathbb{Q}[x, y, z] = \langle -x + y + z \rangle,$$

que en este caso es radical, obteniendo la ecuación implícita exacta del plano.

Si por el contrario hacemos uso de resultantes, tras homogeneizar, el sistema auxiliar de la parametrización sería

$$\begin{cases} s + t - rx = 0, \\ s - ry = 0, \\ t - rz = 0. \end{cases}$$

Aplicando el [Teorema 1.11](#), podemos obtener el resultado auxiliar como el determinante de la matriz formada por los coeficientes de las variables r, s, t de las ecuaciones del sistema auxiliar:

$$\text{Res}_{\text{aux}}(x, y, z) = \det \begin{pmatrix} -x & 1 & 1 \\ -y & 1 & 0 \\ -z & 0 & 1 \end{pmatrix} = -x + y + z.$$

Obtenemos de nuevo la que ya sabemos que es la ecuación exacta. Podemos comprobarlo

1 Fundamentos matemáticos de las SDFs

usando el [Teorema 1.13](#) como sigue:

$$f(x, y, z) = \frac{\text{Res}_{\text{aux}}(x, y, z)}{\gcd\left(\text{Res}_{\text{aux}}(x, y, z) \cdot \frac{\partial \text{Res}_{\text{aux}}}{\partial x}(x, y, z)\right)} = \frac{-x + y + z}{\gcd(-x + y + z, -1)} = \frac{-x + y + z}{1}.$$

2 Algoritmos de visualización de SDFs

Una vez estudiadas las técnicas a través de las cuales podemos crear y manipular primitivas, podemos usar estas para formar la escena que queremos representar. Podemos optar por dos enfoques. El primero de ellos es hacer *spheretracing* una vez por cada primitiva que conforme la escena, lo cual permitiría tener un control más preciso sobre las propiedades de cada objeto de la escena de forma individual, como por ejemplo su apariencia (material) o distancia de dibujado. El otro enfoque consiste en combinar todas las primitivas de la escena en una sola mediante el operador booleano de unión. La principal ventaja en este caso sería el tener toda la escena definida a partir de una única SDF, de forma que tenemos la información más condensada y el renderizado es más sencillo al no tener que combinar el resultado de varias ejecuciones del algoritmo de *spheretracing*. El precio a pagar por esta simplicidad es que perdemos el control más granular que sí teníamos antes. No obstante, optaremos por este último enfoque, tanto por simplicidad como porque nuestro objetivo final es crear nuevas superficies, y lo que más sentido tiene es que usemos una única SDF para representarlas.

Ahora que tenemos definida la escena a partir de una función distancia con signo, necesitamos una forma de visualizarla. En IG se utilizan diferentes técnicas y algoritmos que transforman datos geométricos en otros que nuestras pantallas puedan representar. Dos de los métodos más utilizados son la rasterización y el trazado de rayos o *raytracing*. En general, la rasterización es el método más usado para aplicaciones interactivas, ya que las GPUs fueron originalmente diseñadas para realizar rasterización de forma eficiente. Por otro lado, el *raytracing* ofrece resultados más realistas, sobretodo en los aspectos relacionados con la iluminación, a precio de ser más lento. En los últimos años son cada vez más comunes las GPUs con soporte hardware para *raytracing*, haciendo que se extienda su uso a aplicaciones interactivas, como los videojuegos.

A la hora de trabajar con estos algoritmos es importante la forma en la que se representa la información de la geometría. La más común en rasterización y *raytracing* es a través de mallas de polígonos, un conjunto de puntos de un espacio afín que forman caras planas. En ambos algoritmos necesitamos hacer uso del concepto de primitiva como los elementos más pequeños que pueden ser visualizados, y típicamente se trata de triángulos cuando se trabaja con mallas de polígonos. En nuestro caso no usamos mallas de polígonos, y tenemos una representación no discreta de la geometría de la superficie. Si bien la rasterización se puede adaptar para trabajar con objetos diferentes a mallas de polígonos, esto no es lo común, y el *raytracing* es mucho más adaptable en estos casos, pues permite trabajar con cualquier tipo de objeto con el que se pueda calcular la intersección con un rayo, como ocurre con las SDFs.

La **rasterización** recorre cada primitiva P del modelo, comprobando para cada una qué conjunto S de pixels de la pantalla la cubren. Una vez obtenidos los pixels, se ejecutará un programa escrito por el programador (*fragment shader*) para cada uno de ellos, que calculará el color final del píxel. El funcionamiento del *raytracing* es similar al de la rasterización, pero intercambiando los dos bucles. El procedimiento consiste por tanto en recorrer los pixels de

la pantalla y comprobar qué primitivas del modelo cubren cada uno. Para ello se traza un rayo por el centro de cada píxel y se calcula la intersección con el objeto, razón del nombre “trazado de rayos”. En la [Figura 2.1](#) y la [Figura 2.2](#) podemos apreciar las diferencias en el procedimiento de ambos algoritmos. Ambos métodos tienen complejidad algorítmica $\mathcal{O}(pn)$, siendo n el número de primitivas y p el de pixels, aunque en el caso del *raytracing* esta se puede mejorar usando indexación espacial para la obtención del conjunto de primitivas que cubren cada píxel.

: Rasterización	: Raytracing
<pre> inicializar color de todos los pixels for cada primitiva P en el modelo do $S \leftarrow$ pixels cubiertos por P for cada píxel q en S do calcular color de P en q actualizar color de q end end </pre>	<pre> inicializar color de todos los pixels for cada píxel q de la pantalla do $T \leftarrow$ primitivas que cubren q for cada primitiva P en T do calcular color de P en q actualizar color de q end end </pre>

Figura 2.1: Algoritmos de visualización

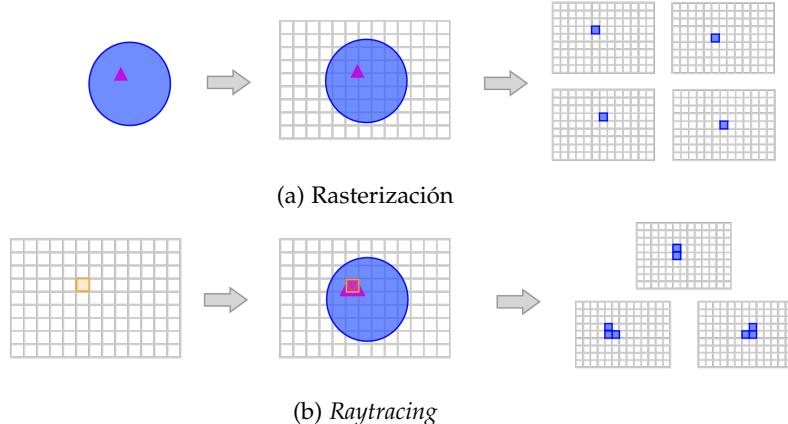


Figura 2.2: Funcionamiento de los métodos de rasterización y *raytracing*

Teniendo en cuenta las consideraciones anteriores, para representar superficies generadas por funciones distancia con signo utilizaremos *spheretracing*, un método basado en *raytracing*. No obstante, para ello haremos uso de las APIs de rasterización en GPU, lo cual puede parecer contradictorio, pues como hemos visto ambos algoritmos tienen una estructura opuesta. El motivo fundamental de esta aparente contradicción es que las APIs de *raytracing* solo están disponibles en GPUs modernas y avanzadas, y nosotros queremos poder realizar esta tarea en el mayor número de dispositivos posible. Podríamos conseguir esto realizando los cálculos en la CPU recorriendo secuencialmente los pixels, pero incluso parallelizándolos en varias hebras asignando a cada una un conjunto de pixels no podríamos conseguir el grado de interactividad que buscamos. La única solución es por tanto usar la GPU para realizar los

2.1 Renderizado por raytracing en GPU usando un *fragment shader*

cálculos de la intersección rayo-escena, pues son mucho más rápidas en este tipo de cálculos. Además, hoy en día prácticamente todos los dispositivos cuentan con GPUs, incluso los dispositivos móviles, y las APIs de rasterización son muy portables y conocidas.

Para lograr nuestro objetivo de trazar rayos usando las APIs de rasterización usaremos que, como hemos visto, estas permiten ejecutar para cada píxel donde se proyecte una primitiva un código definido por el programador llamado *fragment shader* y que produce el color del píxel. Así, en lugar de hacer *raytracing* sobre una escena 3D, visualizaremos por rasterización un plano formado por dos triángulos cubriendo toda la imagen, lo que provocará la ejecución de una instancia del *fragment shader* en cada píxel de la imagen, y será en él donde se implemente el algoritmo de intersección rayo-escena. A este plano lo llamaremos **lienzo** o *canvas*, pues efectivamente estaremos pintando la escena encima suya píxel a píxel a través del *fragment shader*.

2.1. Renderizado por *raytracing* en GPU usando un *fragment shader*

En esta sección estudiaremos en detalle el proceso de creación del lienzo usando una API cualquiera de rasterización y el desarrollo de los cálculos necesarios para ello, incluyendo la intersección del rayo con la escena, simulación avanzada de iluminación y técnicas de suavizado de la imagen.

2.1.1. Creación del lienzo

Cuando introdujimos el método de rasterización dijimos que este suele ser usado con mallas de polígonos, que a su vez estaban compuestas por puntos en un espacio afín que se unen formando caras. Lo cierto es que en IG se hace uso de múltiples espacios de coordenadas, los cuales es imprescindible conocer para entender el proceso de definición de geometría y 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.
- **Coordenadas de dispositivo:** 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.

Dado que para usar rasterización el lienzo deberá estar definido como una malla de polígonos, deberemos declarar los vértices que la conforman y cómo estos se unen formando primitivas, en este caso triángulos. Este lienzo, como toda geometría, tendrá asignado dos *shaders* o procesadores, que son programas que se ejecutan en la GPU. Estos programas pueden recibir parámetros, pero son independientes entre sí, siendo la única forma en la que

2 Algoritmos de visualización de SDFs

pueden comunicarse entre ellos mediante el paso de atributos de entrada y salida. Hay dos tipos de *shaders*: de vértices (*vertex shader*) y de fragmentos o pixels (*fragment shader*), cada uno con atributos específicos de entrada y salida.

En el *vertex shader* utilizaremos los siguientes atributos.

- v_{loc} : vector de cuatro flotantes que contiene las coordenadas homogéneas locales del vértice a procesar. La cuarta componente es la componente homogénea, que es necesaria para realizar el cambio a coordenadas recortadas. Se trata de un parámetro de entrada enviado por la aplicación al visualizar una secuencia de vértices.
- v_{wc} : vector de cuatro flotantes con la posición transformada del vértice actual. El objetivo del *vertex shader* es el de calcular su valor, luego es un parámetro de salida.

Por otro lado, en el *fragment shader* usaremos los que siguen.

- v_{frag} : vector de cuatro flotantes con las coordenadas de dispositivo para el centro del píxel actual. Es un atributo de entrada, los cuales interpolan su valor automáticamente en cada vértice en los *fragment shaders*. La cuarta componente es la inversa de la componente homogénea de v_{rec} , y se utiliza en el cálculo de la profundidad de los pixels y en las operaciones de corrección de perspectiva.
- v_{col} : terna RGBA de flotantes que contendrá el color del píxel actual. Es un parámetro de salida, y la función del *fragment shader* es otorgarle un valor.

En primer lugar se ejecuta una instancia del **procesador de vértices o vertex shader** para cada vértice de la geometría. Su finalidad es 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 v_{loc} , para cambiar de un sistema de coordenadas a otro se utilizan matrices de transformación [43] [44]. Todas ellas son de flotantes con dimensión 4×4 , y haremos uso de las siguientes.

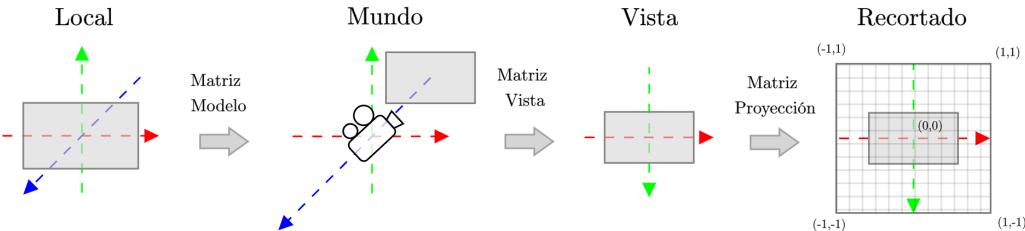


Figura 2.3: Coordenadas locales a recortadas

- **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 la matriz identidad de dimensión cuatro

$$M = Id_{4 \times 4}.$$

- **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 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 en función de las siguientes características del *view-frustum*, la región del espacio de la escena que es visible por pantalla.

- Apertura vertical del campo de visión β indicada como un ángulo entre 0 y 180 grados.
- Relación de aspecto a entre el ancho w y el alto h del *view-frustrum*.
- Límites cercano y lejano en el eje Z cambiados de signo n y f del *view-frustrum*.

La matriz se construye como

$$M_P = \begin{pmatrix} \frac{c}{a} & 0 & 0 & 0 \\ 0 & c & 0 & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2nf}{n-f} \\ 0 & 0 & -1 & 0 \end{pmatrix}, \text{ donde } c = \coth\left(\frac{\beta}{2}\right).$$

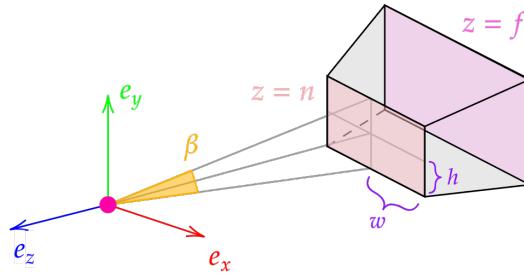


Figura 2.4: Parámetros del *view frustum*

Con esta información ya podemos escribir nuestro *vertex shader*, mostrado en la Figura 2.5.

Tras la ejecución del *vertex shader* las coordenadas de recortado se transforman a coordenadas de dispositivo, que el **procesador de fragmentos o fragment shader** recibirá como entrada. De él se ejecutará una instancia para cada píxel de la pantalla, y su objetivo es asignar a la variable v_{loc} el color que el píxel tendrá como una terna RGBA, y será aquí donde hagamos todos los cálculos necesarios para renderizar la superficie con *spheretracing*. El primer paso para esto será definir un sistema de coordenadas dentro del propio lienzo con el que sea más

Algorithm 4: Vertex Shader

Data: matriz de proyección M_P , matriz de vista M_V , matriz de modelo M_M , y coordenadas locales homogéneas del vértice v_{loc}

Result: posición transformada del vértice actual en coordenadas homogéneas de mundo v_{wc}

$$v_{wc} \leftarrow M_P \cdot M_V \cdot M_M \cdot v_{loc}$$

Figura 2.5: Cuerpo del método `main` del *vertex shader*

cómo trabajar que con el que ya disponemos a través de v_{frag} .

Para obtener estas coordenadas, primero desplazamos el origen que nos proporciona v_{frag} al centro de la pantalla usando el número de columnas y filas de pixels en la imagen, que deberemos pasar como parámetro al *shader* y llamaremos `u_resolution`, para posteriormente normalizar respecto a alguno de los ejes. Hacemos esto porque si intentamos normalizar sobre ambos ejes obtendremos coordenadas en el rango $[-0.5, 0.5]^2$, y al no ser (en general) el lienzo cuadrado, la imagen se verá 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, el resultado de dividir el ancho de la imagen por el alto, ambos medidos en pixels. Finalmente, para que la coordenada Y tome siempre valores en $[-1, 1]$ multiplicamos por dos. Con esto conseguimos las coordenadas del centro del pixel en coordenadas normalizadas de dispositivo, similares a las de recortado pero divididas por la componente homogénea.

$$uv = \frac{2 \cdot (v_{frag} - 0.5 \cdot u_resolution)}{u_resolution_y}.$$

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, ya que las vamos a usar para pintar sobre el lienzo como si de aplicar una textura se tratase. Cuando se trabaja con *fragment shaders* la forma de debugear es dando colores a los pixels. En nuestro caso, podemos ver la diferencia entre ambos sistemas de coordenadas si usamos uv como los canales rojo y verde del parámetro de salida v_{col} , superponiendo una rejilla construida usando la función módulo para apreciar si existe deformación, tal y como se muestra en la Figura 2.6. En ella podemos observar que si normalizamos sobre el eje horizontal, los verdes y amarillos no llegan a ser del todo intensos, ya que la componente del verde (la vertical) no llega a su valor máximo. Normalizando ambos ejes obtenemos el rango de colores completo, pero existe deformación en la rejilla debido a que el lienzo no es cuadrado. Finalmente, normalizando sobre el eje vertical obtenemos también el rango completo de colores, ya que los valores del eje horizontal que se salen del rango $[-1, 1]$ son visualizados como si hubieran sido acotados en dicho intervalo, pero la rejilla se dibuja correctamente. En las siguientes secciones veremos cómo usar estas coordenadas para dibujar nuestra superficie sobre el lienzo.

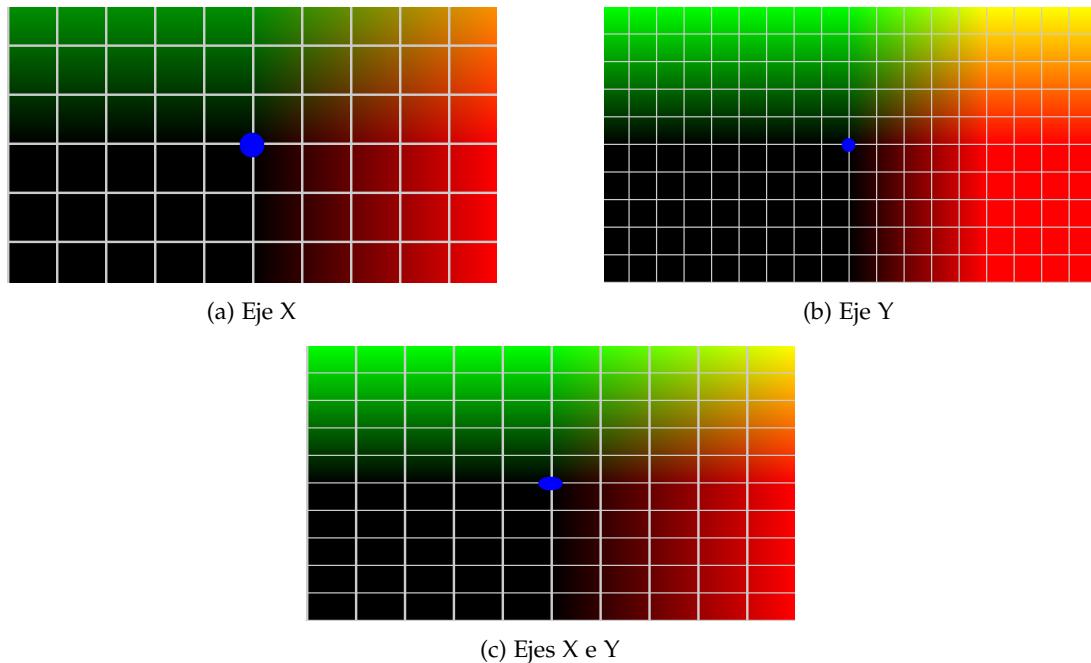


Figura 2.6: Normalización de coordenadas sobre distintos ejes

2.1.2. Generación de rayos primarios

A partir de ahora, pensamos en nuestra escena no como en la que hemos definido el lienzo, sino aquella que queremos dibujar usando *spheretracing* dada una función distancia con signo ϕ . Esta escena estará definida en el espacio de coordenadas de mundo, el cual tiene un marco de referencia formado por tres vectores unitarios $B = \{e_x, e_y, e_z\}$ y un origen o , y colocaremos en ella los siguientes elementos (a partir de ahora supondremos que todas las tuplas de coordenadas son en coordenadas de mundo):

- 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 $l = o = (0, 0, 0)$.

La idea para conseguir representar la superficie consiste en trazar rayos a partir de c_o hacia el centro de cada uno de los cuadrados del plano de visión, cada uno representando un píxel de la pantalla, de forma que si el rayo interseca con S_ϕ significa que ese píxel corresponde a un punto de la superficie, y será coloreado como tal.

Cada uno de estos rayos estará definido por un origen r_o y una dirección r_d . El origen será siempre la posición de la cámara c_o , pero obtener la dirección requiere más trabajo. En el escenario descrito en la [Figura 2.7](#), donde S_ϕ es una esfera centrada en el origen

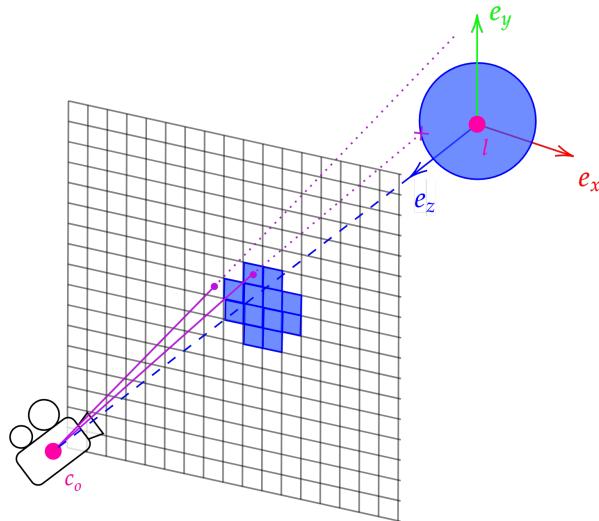


Figura 2.7: Trazado de rayos a través del plano de visión

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

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

Una opción sería tomar $c_0 = (0, 0, d)$, donde el valor d es la distancia desde el punto del observador (foco de la proyección) al plano de visión. Modificar el valor de d cambiaría los ángulos de apertura de visión horizontal y vertical, cambiando el tamaño aparente de los objetos proyectados. Es decir, d actuaría como un control del campo de visión, de forma que cuanto mayor sea su valor mayores serán los ángulos de visión, y por tanto los objetos se verán más pequeños. Lo fijaremos a un valor de 1. Sin embargo este escenario es el más sencillo posible, y si queremos poder mover la cámara manteniendo el punto de atención l en el centro de la pantalla tendremos que poder trabajar con una orientación arbitraria suya. Para ello deberemos construir un marco de coordenadas de vista relativo a la cámara, que definiremos por una base ortonormal $\{f_1, f_2, f_3\}$ y el origen c_0 .

Para obtener el valor de los elementos de f_1, f_2 y f_3 empezamos obteniendo los siguientes vectores ortonormales.

- **Vector director c_d :** indica la dirección hacia la que mirará la cámara, que será paralela al eje Z del marco de coordenadas de vista. Su valor vendrá dado por $c_d = l - c_0$.
- **Right vector c_r :** paralelo al eje X del marco de coordenadas de vista. Debe ser perpendicular tanto a c_d como al vector *view-up*. Este último, que denotaremos como u , es un vector libre que indica la dirección que el observador verá proyectada en vertical y apuntando hacia arriba en la imagen, y normalmente se toma el valor $u = (0, 1, 0)$

para que coincida con el eje Y. Al ser c_r perpendicular al vector director y a *view-up*, podemos obtenerlo como el producto vectorial de ambos: $c_r = u \times c_d$.

- **Up vector c_u :** paralelo al eje Y del marco de coordenadas de vista. Como debe ser ortogonal a los otros dos, lo calculamos 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 que tenemos el nuevo marco cartesiano, queda transformar el vector director original $r_d = (u, v, -1)$ 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 $\{e_x, e_y, e_z\}$, que al ser la base del marco de coordenadas de mundo y al tratarse de una matriz de cambio de base entre dos marcos cartesianos, coincidirá con escribir por columnas $\{f_1, f_2, f_3\}$, de forma que

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

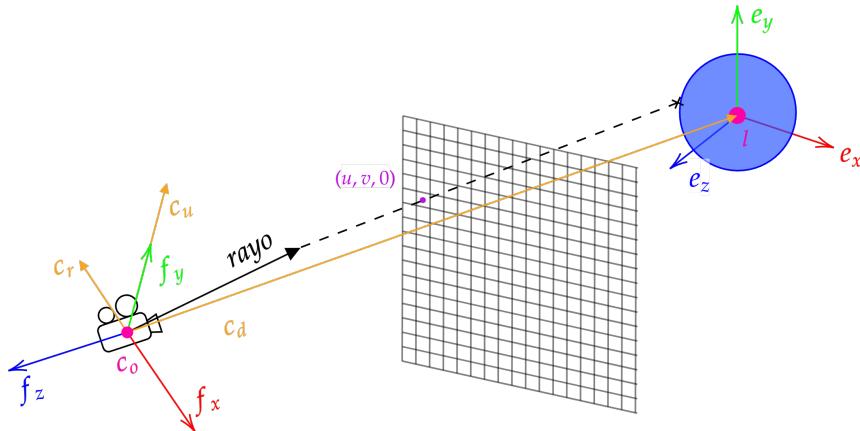


Figura 2.8: Obtención de la dirección del rayo

2.1.3. Algoritmos de intersección rayo-escena: *raymarching* y *spheretracing*

Una vez que conocemos toda la información del rayo ya estamos en disposición de comprobar si este interseca con S_ϕ . Para esto se pueden utilizar varios algoritmos, de entre los que estudiaremos el *raymarching* y el *spheretracing*. El método del *spheretracing* es una variante del *raymarching*, así que estudiaremos este primero.

El *raymarching* es un método iterativo: a partir de c_o , en cada iteración avanzamos en la dirección del rayo una distancia fija δ . Evaluamos entonces nuestra 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 2.10 ilustra este procedimiento, donde `DibujarSuperficie()` y `DibujarFondo()` devuelven ternas RGBA que serán asignadas al píxel actual dependiendo de si hay intersección o no.

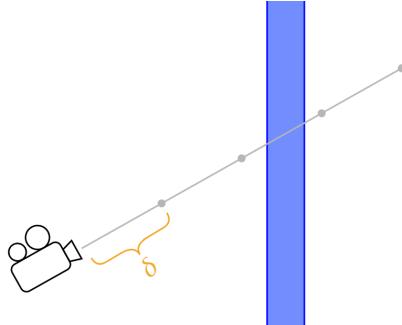


Figura 2.9: Pérdida de intersección en *raymarching* para valores elevados de δ

: Raymarching

Data: origen del rayo c_o , dirección del rayo v
Result: terna RGB con el color del píxel actual
 $d \leftarrow 0 //$ distancia total
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
 $d \leftarrow d + \delta;$
 if $d > \text{MAX_DISTANCIA}$ **then**
 | **return** `DibujarFondo()`
 end
end

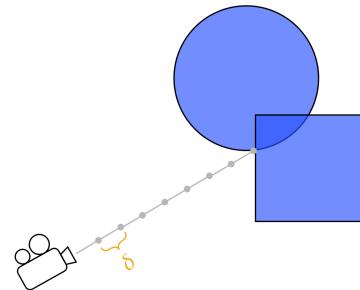


Figura 2.10: Algoritmo de *raymarching*

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 además será bastante alto debido a que el valor de incremento δ debe ser pequeño si no queremos perder ninguna intersección, como ocurre en la Figura 2.9.

Como solución a este problema aparece el *spheretracing*, que reduce drásticamente el número de iteraciones, y por tanto de 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 perdernos una intersección. 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 pixels de la pantalla corresponden a la superficie S_ϕ , y se encuentra descrito con detalle en la [Figura 2.11](#). Con esto al fin podemos describir la forma que tendrá nuestro *fragment shader* en la [Figura 2.12](#). 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 2.13](#) para una esfera centrada en el origen. Veremos como mejorar esto en la próxima sección.

:Spheretracing

Data: origen del rayo c_o , dirección del rayo v
Result: terna RGB con el color del píxel actual
 $d \leftarrow 0 //$ distancia actual

```

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
     $d \leftarrow d + sdf$ 
    if  $d > MAX\_DISTANCIA$  then
        return DibujarFondo()
    end
end
```

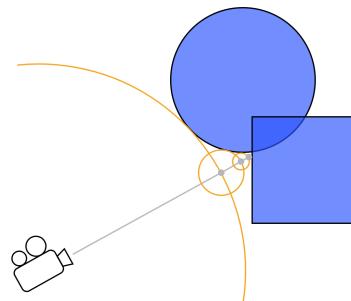


Figura 2.11: Algoritmo de *spheretracing*

2.2. Modelos de iluminación y sombras

Ya sabemos qué pixels 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.

Algorithm 5: Fragment Shader

Data: coordenadas de dispositivo v_{frag} del píxel actual
Result: color del píxel actual como terna RGBA v_{col}

$$c_0 \leftarrow \text{posición de cámara en función de la entrada del ratón}$$

$$l \leftarrow \text{punto de atención elegido por el usuario}$$

$$uv \leftarrow 2 \cdot \frac{v_{frag} - 0.5 \cdot u_resolution}{u_resolution_y}$$

$$f_1, f_2, f_3 \leftarrow \text{CalcularMarcoCartesiano}(c_0, l)$$

$$r_d \leftarrow (f_1 | f_2 | f_3) \cdot \text{normalizar}((uv_x, uv_y, -1))$$

$$\text{color} \leftarrow \text{spheretracing}(c_0, r_d)$$

$$v_{col} \leftarrow (\text{color}_x, \text{color}_y, \text{color}_z, 1)$$

Figura 2.12: Cuerpo del método `main` del *fragment shader*

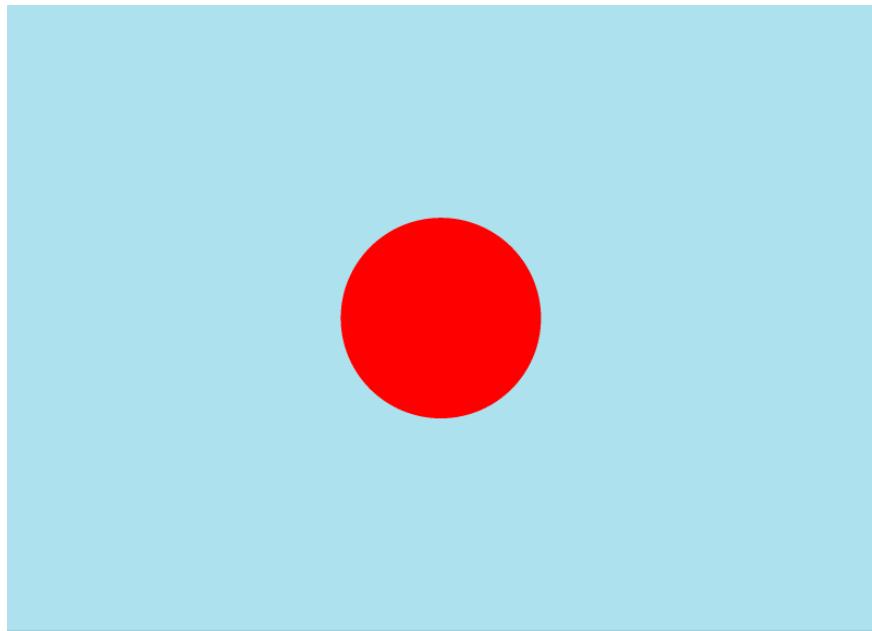


Figura 2.13: Resultado de *spheretracing* asignando colores planos

2.2.1. Modelos de Phong 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 [45], también popular pero mucho más simple y computacionalmente menos costoso. A su vez este modelo se basa en el de Phong, presentado en 1975 [46], y 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 $p \in \mathbb{R}^3$.
- Un número finito n de **fuentes de luz**. Llamaremos l_i con $i \in \{1, \dots, n\}$ 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 direcionales, de forma que no serán visibles en la escena. Además supondremos que emiten una radiancia constante S_i para $i \in \{1, \dots, n\}$.
- 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

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

2 Algoritmos de visualización de SDFs

Como L_A y L_E son constantes solo 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 $p \in S_\phi$ ya que, si bien la radiancia en un punto $q \notin S_\phi$ en la dirección v depende de la radiancia saliente del primer punto visible desde q en la dirección $-v$, nosotros solo evaluaremos el modelo de iluminación cuando detectemos una intersección con la superficie. En caso de no hallar intersección alguna tomaremos simplemente

$$L(p, v) = L_A, \quad p \notin S_\phi.$$

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 la m -ésima fuente de luz visible desde p en la dirección 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 es

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

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 reflexión, 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 . De esta forma, tenemos que

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

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 un factor R_E que codifique la fracción de radiancia reflejada de esta forma, de modo que podemos expresar

la fracción de radiancia especular reflejada como

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

$$(p, v, l_m) \mapsto R_E \cdot \max(0, r_m \cdot v)^\alpha, \quad \alpha \in \mathbb{R}.$$

El valor α se llama **exponente de brillo**, y nos proporciona control sobre el tamaño e intensidad de las zonas brillantes del objeto, siendo más pequeños e intensos los brillos generados cuanto mayor sea su valor. Un ejemplo de por qué esto nos resulta útil es comparar materiales especulares, como el mármol y el metal. Ambos 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, luego tomaríamos $\alpha_{\text{metal}} > \alpha_{\text{mármol}}$.

- **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. Un ejemplo de material con estas propiedades sería el yeso, para el cual tomaremos $\|R_D\| \gg \|R_E\|$. Al igual que antes, solo queremos que 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

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

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

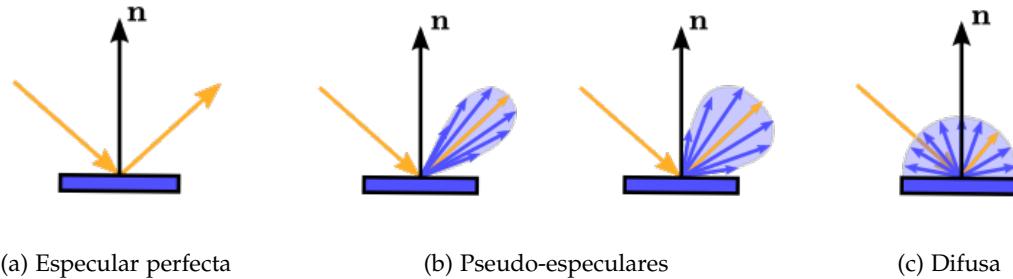


Figura 2.14: Tipos de función de distribución de reflectividad bidireccional [47]

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.

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 :

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

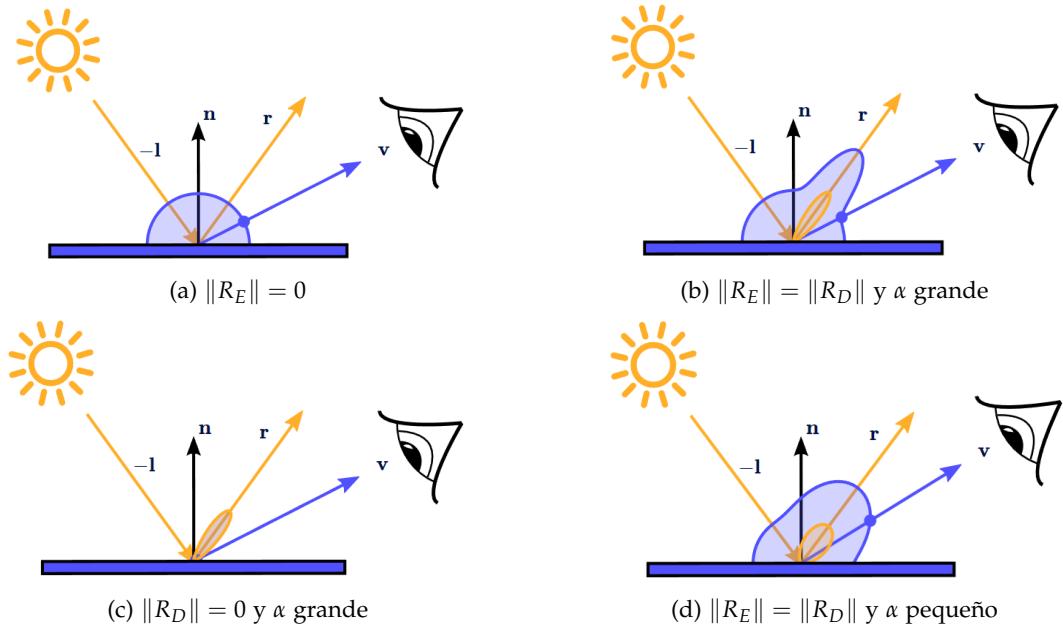


Figura 2.15: Ejemplo de distintos valores para R_E , R_D y α

Recogemos los resultados obtenidos en la siguiente definición.

Definición 2.3 (Modelo de Phong). 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 Phong viene dada por

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

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 acotadas entre $[0, 1]$ 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 1977 James F. Blinn [48] introdujo una variante a este modelo que hoy conocemos como **modelo de Blinn-Phong**. Su única diferencia con el de Phong consiste en el uso del llamado

halfway vector

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

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 2.16 podemos ver el comportamiento de h_m para distintas configuraciones de l_m y v .

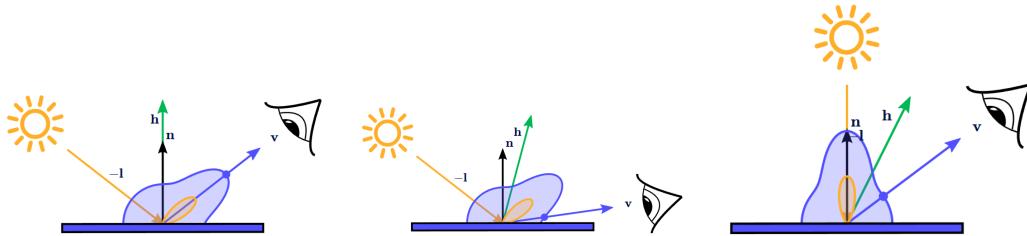


Figura 2.16: Comportamiento de h_m con $\|R_S\| = \|R_D\|$

Aunque esta nueva versión se trate de una simplificación del modelo de Phong, 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 2.17.

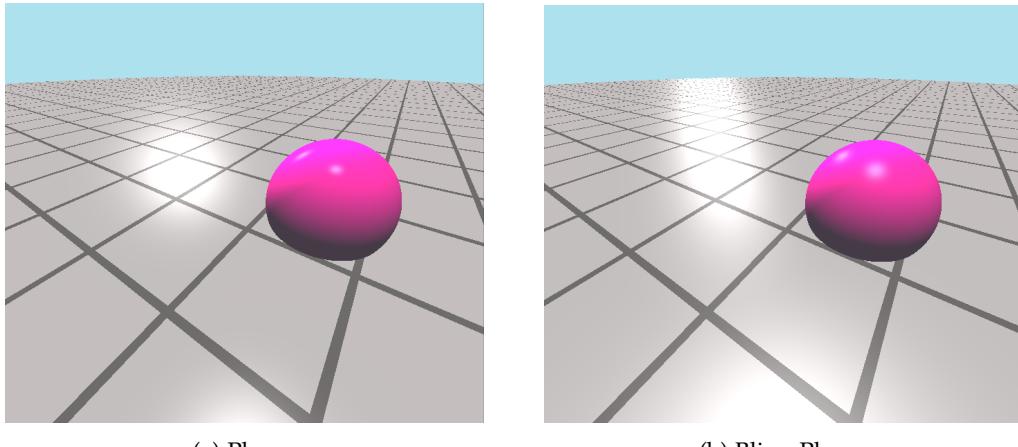


Figura 2.17: Zonas brillantes en modelos de Phong y Blinn-Phong

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

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

Ya podemos darle forma a las funciones DibujarSuperficie y DibujarFondo usadas en la Figura 2.11, suponiendo que pasamos como *uniforms* los parámetros del material y los valores l_i y S_i para cada $i \in \{1, \dots, n\}$. Solo queda un asunto por tratar. A la vista de la

Algorithm 6: DibujarSuperficie

Data: punto p , dirección del rayo v , distancia $\phi(p)$
Result: terna RGB con la radiancia percibida en el punto p

```

 $L \leftarrow L_A // Radiancia final$ 
for  $i \in \{1, \dots, n\}$  do
     $h \leftarrow \text{normalizar}(L_i - v) // \text{Observador en dirección opuesta a la del rayo}$ 
     $N_p \leftarrow \text{CalcularNormal}(p)$ 
     $NLi \leftarrow \max(0, N_p \cdot l_i)$ 
     $NH \leftarrow \max(0, N_p \cdot h)$ 

     $f_{ra} = R_A$ 
     $f_{rd} = NLi \cdot R_D$ 
     $f_{re} = NLi \cdot R_E \cdot NH^\alpha$ 

     $L \leftarrow L + S_i \cdot (f_{ra} + f_{rd} + f_{re})$ 
end
return  $L$ 

```

Algorithm 7: DibujarFondo

Result: terna RGB con el color de fondo de la escena
return L_A

Figura 2.18: Implementación de las funciones DibujarSuperficie y DibujarFondo

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. En la siguiente sección veremos presentamos una técnica para calcularlo de forma aproximada.

2.2.2. Sombras

Los resultados obtenidos en la Figura 2.17 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 las SDFs 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 2.19 para añadir este comprobación.

Algorithm 8: DibujarSupercicie

Data: punto p , dirección del rayo v , distancia $\phi(p)$
Result: terna RGB con la radiancia percibida en el punto p

```

 $L \leftarrow L_A + L_E // \text{Radiancia final}$ 
for  $i \in \{1, \dots, n\}$  do
     $// \dots$ 
     $\text{sombras} \leftarrow \text{CalcularSombras}(p, l_i)$ 
     $L \leftarrow L + S_i \cdot (f_{ra} + f_{rd} + f_{re}) \cdot \text{sombras}$ 
end
return  $L$ 
```

: CalcularSombras

Data: punto p_0 , dirección de luz l_i
Result: factor de sombra en p_0 en el rango $[0, 1]$

```

 $d \leftarrow \delta // \text{distancia actual}$ 
for  $i \in \text{MAX\_ITERACIONES}$  do
     $p \leftarrow p_0 + d \cdot v$ 
     $sdf \leftarrow \phi(p)$ 
    if  $sdf < \varepsilon$  then
         $\mid \text{return } 0;$ 
    end
     $d \leftarrow d + sdf$ 
    if  $d > \text{MAX\_DISTANCIA}$  then
         $\mid \text{return } 1$ 
    end
end
return  $1$ 
```

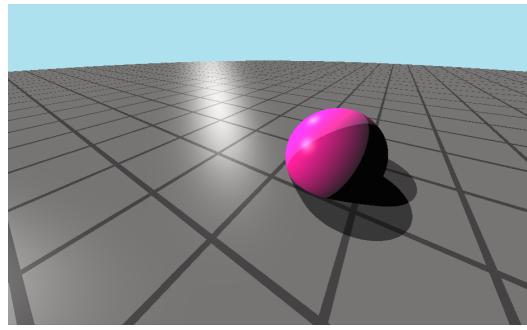


Figura 2.19: Cálculo básico de sombras

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 la Figura 2.11 no podemos inicializar $d = 0$, ya que entonces se detectaría una intersección en el mismo punto p . En su lugar tomamos un valor inicial $\delta \in]0, \varepsilon[$, donde ε es el umbral que se utiliza como parada en el algoritmo de *spheretracing*.

Estudiando los resultados obtenidos vemos que al añadir sombras obtenemos una imagen mucho más realista y cohesiva, otorgando 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* tomando como origen del rayo el punto del que queremos evaluar la sombra y como dirección la de la fuente de luz. Representaremos la proporción de luz que alcanza el punto p_0 como el valor $sombra \in [0, 1]$, donde 0 significa que el punto no recibe nada de luz, y por tanto aparecerá totalmente oscuro, y 1 que ningún obstáculo hace sombra sobre el punto y está totalmente iluminado por la fuente de luz. El caso de que el punto esté totalmente en sombra ya lo hemos visto en la versión anterior del algoritmo. Ahora estudiamos que ocurre cuando el punto no haya sido alcanzado por el rayo pero haya estado muy cerca de serlo. En este caso, dejaremos que reciba algo de luz, formando una región de penumbra. Calcularemos esta cantidad en base a las siguientes dos condiciones.

1. La cantidad de luz que permitiremos que reciba será menor cuanto más haya faltado para que se produzca la intersección. Si casi se produce la intersección entonces significa que hay un obstáculo muy cerca de la trayectoria del rayo, y por tanto la evaluación de la SDF en ese punto tendrá un valor muy pequeño, y lo contrario ocurre si la intersección estuvo lejos de ocurrir:

$$sombra \propto sdf.$$

2. Tiene sentido que la distancia al obstáculo también influya en la medida en la que este ocluye al punto. Esta distancia es también accesible a través del valor d del algoritmo, y heuristicamente se ha comprobado la utilidad de la expresión

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

Esta nueva versión de CalcularSombras se encuentra descrita en la [Figura 2.20](#). 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.

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 2.21](#). 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 la SDF. Hay varias formas de solventar esto, como la propuesta por Sebastian Aaltonen [7] 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 [49] en Shadertoy y estudiada posteriormente por Íñigo Quílez [50].

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.

: CalcularSombras

Data: punto p_0 , dirección de luz l_i , tamaño de luz k

Result: factor de sombra en p_0 en el rango $[0, 1]$

```

sombra ← 1
d ← δ // distancia actual
for i ∈ MAX_ITERACIONES do
    p ←  $p_0 + d \cdot v$ 
    sdf ←  $\phi(p)$ 
    if  $sdf < \varepsilon$  then
        | return 0;
    end
    sombra ← min(sombra,  $k \cdot \frac{sdf}{d}$ )
    d ←  $d + sdf$ 
    if  $d > MAX_DISTANCIA$  then
        | return 1
    end
end
return sombra

```

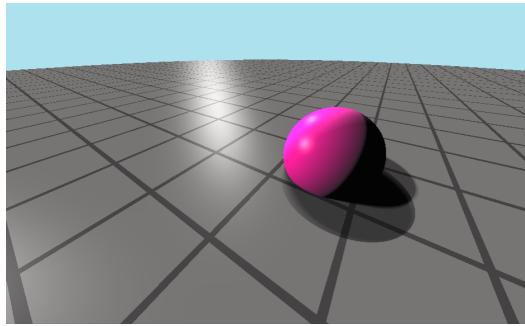


Figura 2.20: Cálculo de sombras suavizadas



Figura 2.21: Detalle de las fugas de luz al calcular sombras

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 2.23 y los resultados que consigue en la Figura 2.27.

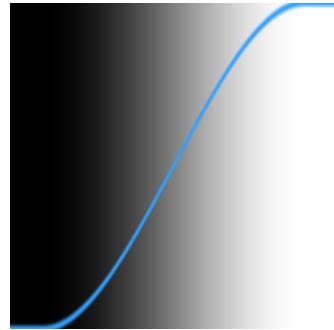


Figura 2.22: Visualización de smoothstep [51]

Algorithm 9: CalcularSombras

```

Data: punto  $p_0$ , dirección de luz  $l_i$ , tamaño de luz  $k$ 
Result: valor en el rango  $[0, 1]$  representando la cantidad de sombra recibida en  $p_0$ 
     $sombra \leftarrow 1$ 
     $d \leftarrow \delta //$  distancia actual
    for  $i \in \text{MAX\_ITERACIONES}$  do
         $p \leftarrow p_0 + d \cdot v$ 
         $sombra \leftarrow \min(res, k \cdot \frac{sdf}{d})$ 
         $sdf \leftarrow \phi(p)$ 
         $d \leftarrow d + |sdf|$ 
        if  $sombra < -1$  OR  $d > \text{MAX\_DISTANCIA}$  then
            break
        end
    end
return  $\text{smoothstep}(-1, 1, sombra)$ 

```

Figura 2.23: Cálculo de sombras suavizadas mejorado

2.2.3. Oclusión ambiental

Al añadir sombras los objetos están mucho más integrados en la escena, pero todavía podemos conseguir un mayo 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 **ocultación ambiental**, y de nuevo gracias a que estamos usando SDFs nos resultará muy fácil y computacionalmente barato simularlo.

Cuando se trabaja con geometría de polígonos, una de las técnicas más comunes es la ocultación 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 pixels cercanos. Con esta información

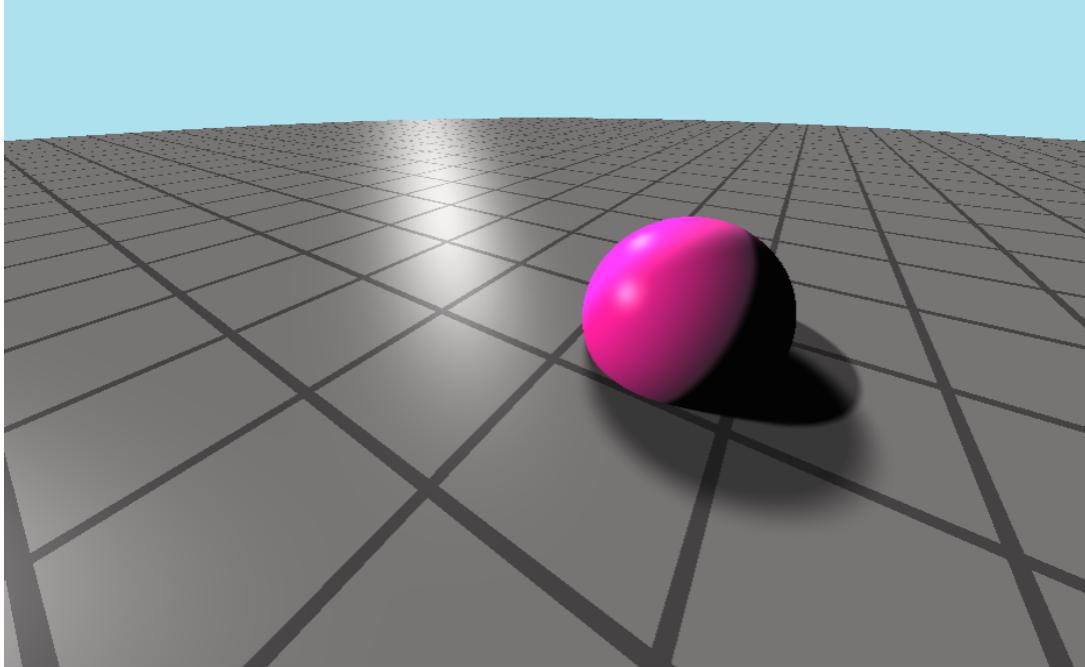


Figura 2.24: Resultado final del cálculo de sombras

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 pixels 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 2.25, o que si un objeto no aparece en pantalla directamente no sea reflejado, como se representa en la Figura 2.26.

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 podemos 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 de la SDF que el algoritmo de *spheretracing*. La técnica que vamos a usar fue ideada por Alex Evans en 2006 [53], y se conoce como **occlusión ambiental muestreada por la normal**.

El método se basa en dado un punto $p \in S_\phi$ evaluar la SDF en varios puntos del vector normal N_p a diferentes 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

2 Algoritmos de visualización de SDFs



Figura 2.25: Reflejos en el videojuego Ratchet & Clank: Una dimensión aparte usando SSR y raytracing [52]

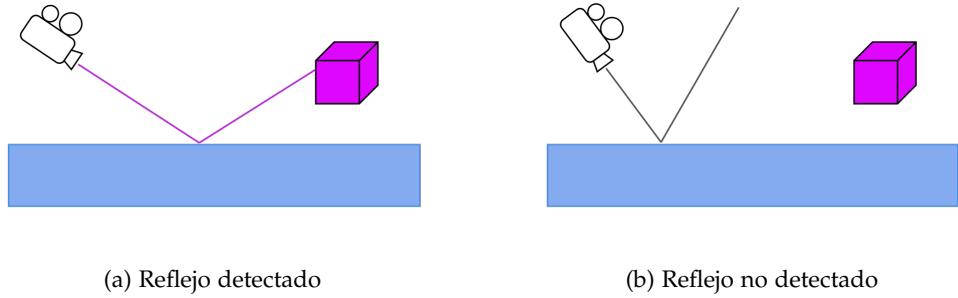


Figura 2.26: Reflejos en agua con SSR

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 a lo largo de N_p hacemos M evaluaciones igualmente espaciadas, es decir con $d_i = \frac{i}{M}$ e $i \in \{1, \dots, M\}$, consideraremos que el punto p no está ocluido si

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

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

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

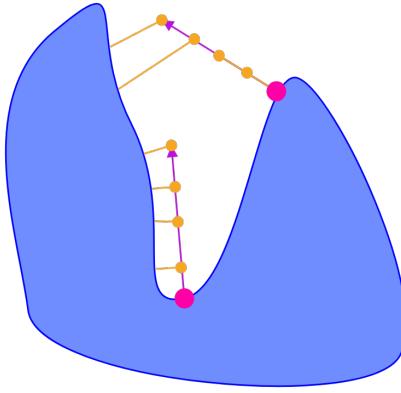


Figura 2.27: Cálculo de oclusión ambiental muestreada por la normal

ya que siempre habrá algún punto a lo largo de N_p que esté a una unidad o menos de 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 2.28. En ella hemos introducido una pequeña optimización [54] 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 2.29, 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 2.33a). En el mercado actual existen multitud de alternativas como solución a este problema. Algunos ejemplos son FXAA (*Fast approximate anti-aliasing*) [55], basado en espacio de pantalla, o SSAA (*Super-sampling anti-aliasing*) [56] y MSAA (*Multisample anti-aliasing*) [57], que usan la técnica del **supermuestreo** o **supersampling** junto con filtros de suavizado. Nosotros implementaremos una versión de SSAA, 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

Algorithm 10: CalcularAO

Data: punto p , vector normal N_p
Result: factor de oclusión ambiental en el rango $[0, 1]$

```

 $ao \leftarrow 1$ 
 $increment \leftarrow 1/M$ 
 $i \leftarrow increment$ 
while  $i < 1$  do
     $sdf \leftarrow \phi(p + iN_p)$ 
     $ao \leftarrow ao - 2^{-iM} \cdot (i - sdf)$ 
     $i \leftarrow i + increment$ 
end
return  $ao$ 
```

Algorithm 11: DibujarSuperficie

Data: punto p , dirección del rayo v
Result: terna RGB con la radiancia percibida en el punto 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  $L$ 
```

Figura 2.28: Cálculo y aplicación de oclusión ambiental

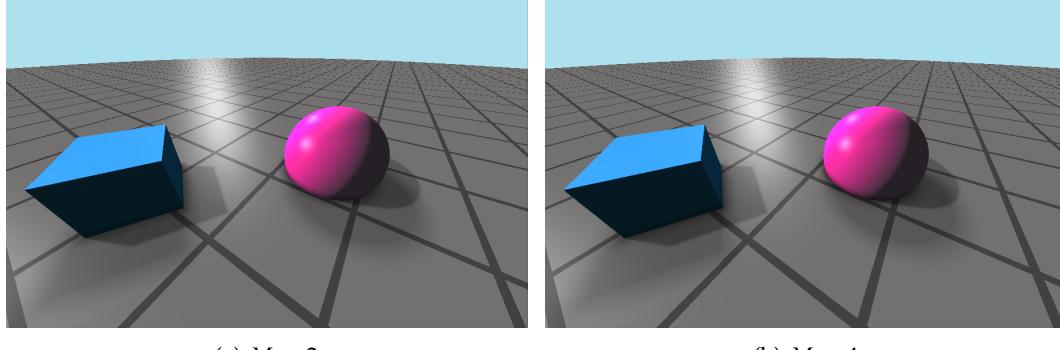
(a) $M = 2$ (b) $M = 4$

Figura 2.29: Resultado del cálculo de oclusión ambiental

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 pixels 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 pixels 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 2.30 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.

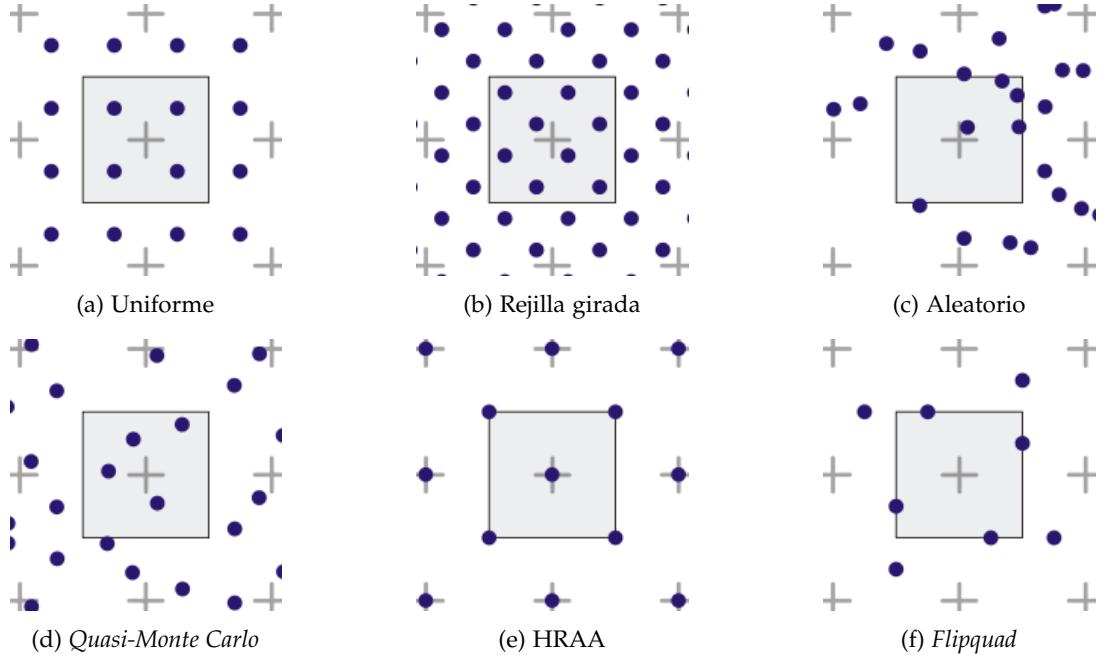


Figura 2.30: Patrones de supermuestreo [58]

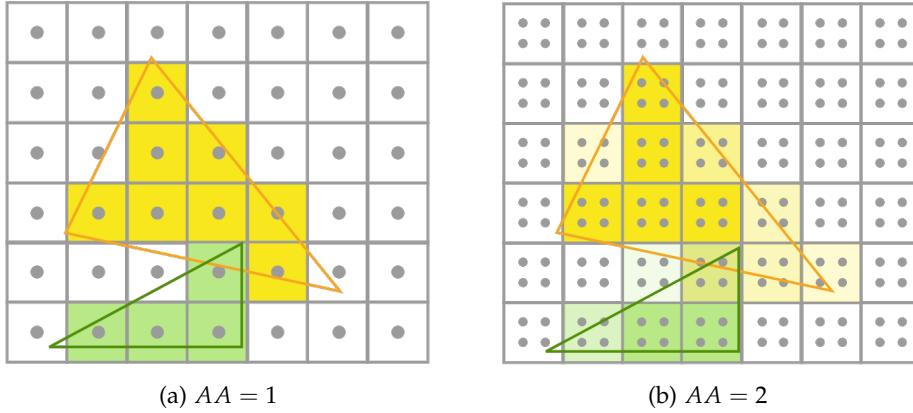
Modificar por dónde pasan los rayos dentro de cada píxel equivale a cambiar cómo calculamos los uv. Llamaremos a partir de ahora AA al nivel de *antialiasing* de la imagen, de forma que por cada píxel haremos pasar AA^2 rayos. Para hallar los nuevos puntos de muestra bastará con subdividir el píxel en AA^2 cuadrantes y quedarnos con el centro de cada uno. Si recordamos que v_{frag} 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 Figura 2.32.

En la Figura 2.33 podemos ver que la pérdida de rendimiento no es en vano y obtenemos una imagen mucho más suave que la original. Sin embargo, como veremos en el Capítulo 4, 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 la Figura 2.34 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 2.31: Antialiasing para diferentes valores de AA

: Fragment Shader

Data: coordenadas de dispositivo v_{frag} del píxel actual
Result: color del píxel actual como terna RGBA v_{col}
 $c_0 \leftarrow$ posición de cámara en función de la entrada del ratón
 $l \leftarrow$ punto de atención elegido por el usuario
 $color \leftarrow (0, 0, 0)$
for $m \in \{0, \dots, AA - 1\}$ **do**
 for $n \in \{0, \dots, AA - 1\}$ **do**
 $offset \leftarrow \frac{(m, n)}{AA} - (0.25, 0.25)$
 $uv \leftarrow 2 \cdot \frac{(v_{frag} + offset) - 0.5 \cdot u_resolution.xy}{u_resolution.y}$
 $r_d \leftarrow (f_1 | f_2 | f_3) \cdot \text{normalizar}((uv_x, uv_y, -1))$
 $color \leftarrow color + \text{spheretracing}(c_0, r_d)$
 end
end
 $color \leftarrow color / AA^2$
 $v_{col} \leftarrow (color_x, color_y, color_z, 1)$

$n \setminus m$	0	1	2
0	•	•	•
1	•	•	•
2	•	•	•

Figura 2.32: Cuerpo del método `main` del *fragment shader*

2 Algoritmos de visualización de SDFs

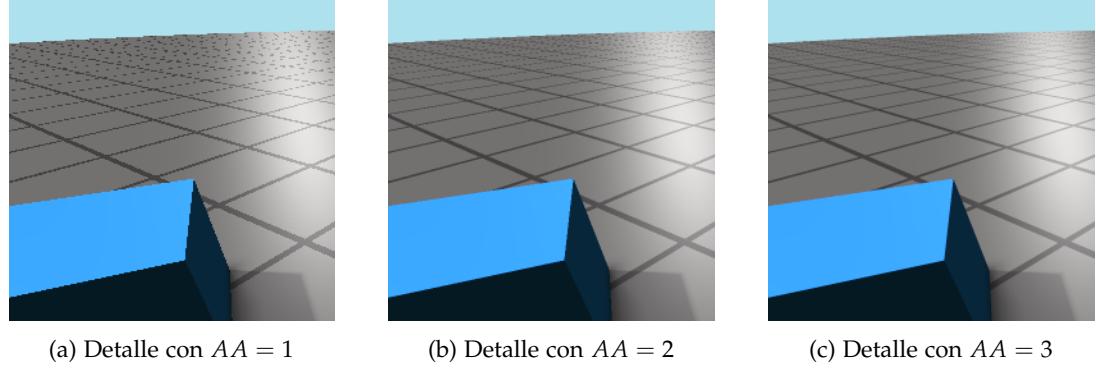


Figura 2.33: Resultados de añadir *antialiasing*

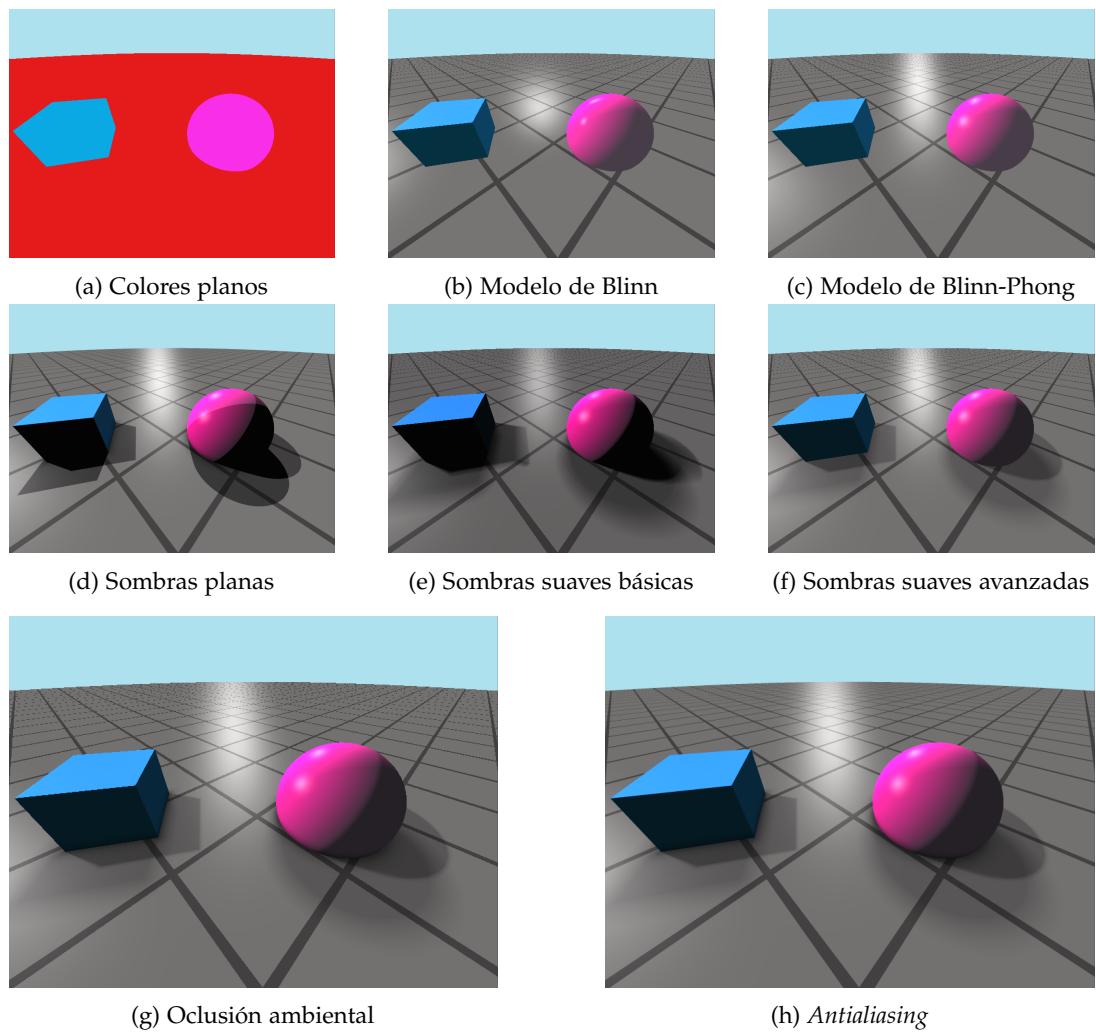


Figura 2.34: Renderizado de SDF añadiendo técnicas de iluminación y renderizado avanzadas

3 Diseño, desarrollo e implementación

En este capítulo veremos el proceso de desarrollo de este proyecto y cómo se han usado las técnicas y conceptos presentados en los capítulos anteriores para ello. El resultado final se puede probar en <https://github.com/Daniel2000815/multivariate-polynomial>.

3.1. Requerimientos

En la introducción se hizo una introducción de los objetivos que se pretenden alcanzar con el desarrollo de este trabajo. A continuación presentamos la especificación de requisitos de nuestro proyecto: una aplicación web para definir superficies mediante SDFs, ecuaciones implícitas o paramétricas, y manipularlas de forma sencilla para el usuario mediante un modelo en estructura de árbol.

3.1.1. Requerimientos funcionales

Podemos separar estos requisitos en tres categorías principales.

Creación de primitivas

- RF1.** Leer entrada del usuario en lenguaje matemático.
- RF2.** Detectar de errores de sintaxis en la entrada de ecuaciones.
- RF3.** Leer entrada del usuario en lenguaje GLSL.
- RF4.** Permitir al usuario usar las primitivas y operadores existentes directamente en su expresión GLSL.
- RF5.** Detectar de errores de sintaxis en la entrada de expresiones GLSL.
- RF6.** Permitir definir parámetros que permitan manipular el aspecto de cada superficie.
- RF7.** Permitir la modificación de las propiedades del material de la superficie.
- RF8.** Implicitar superficies paramétricas.
- RF9.** Obtener SDF aproximada de una ecuación implícita.
- RF10.** Guardar superficies en el almacenamiento local mediante un gestor de primitivas.
- RF11.** Mostrar tabla que enumere las primitivas existentes con sus propiedades básicas.

Visualización de superficies

- RF12.** Representar superficies definidas por SDFs mediante *spheretracing* usando las técnicas del [Capítulo 2](#).
- RF13.** Reflejar en tiempo real los cambios sobre cualquiera de las entradas del *fragment shader*, incluyendo los parámetros introducidos por el usuario o el material.
- RF14.** Girar cámara usando el ratón.
- RF15.** Hacer zoom usando la rueda del ratón.

Manipulación de primitivas

- RF16.** Crear un tipo nodo que permita seleccionar una primitiva de las definidas por el usuario y modificar sus parámetros en tiempo real.
- RF17.** Crear un tipo de nodo que implemente operadores booleanos suavizados y permita modificar tanto el valor como el exponente de suavizado (véase la [Subsección 1.4.1](#)).
- RF18.** Crear un tipo de nodo que implemente operadores afines y permita controlar los valores aplicados sobre cada eje.
- RF19.** Crear un tipo de nodo que implemente operadores deformantes y permita controlar la intensidad del efecto.
- RF20.** Crear un tipo de nodo que implemente los operadores de repetición y permita controlar los valores del número de copias y separación.
- RF21.** Permitir conectar los nodos con el ratón.
- RF22.** Aplicación de operadores en tiempo real según las conexiones realizadas.
- RF23.** Transmisión de los parámetros de un nodo a sus hijos en tiempo real.
- RF24.** Guardar la superficie resultado en el almacenamiento local mediante un gestor de primitivas.
- RF25.** Persistencia del estado de los nodos y conexiones existentes mediante un gestor de nodos.
- RF26.** Los nodos se pueden mover y colapsar para que ocupen menos espacio.
- RF27.** Se puede hacer zoom y moverse por el área en la que se encuentran los nodos.

3.1.2. Requerimientos no funcionales

Estos requisitos se enfocan principalmente en la escalabilidad, usabilidad y experiencia de usuario.

- RNF1.** Se debe soportar el mayor número posible de dispositivos, de forma que el *hardware* usado no suponga una limitación para el usuario.
- RNF2.** La aplicación debe estar correctamente optimizada, y presentar los cambios en tiempo real de la forma más fluida posible.

- RNF3.** La interfaz tiene que ser intuitiva y responsive, realizando el mayor número de operaciones de manera transparente al usuario y guiándolo a través de mensajes de consejo y error.
- RNF4.** La arquitectura de la aplicación debe ser modular y basada en componentes independientes que puedan ser fácilmente actualizables.

3.2. Diseño del interfaz

La aplicación está conformada por tres secciones independientes, entre las que se puede navegar usando el menú superior. En este menú también se encuentra un botón de ayuda que proporciona instrucciones en función de la sección en la que se encuentra el usuario.

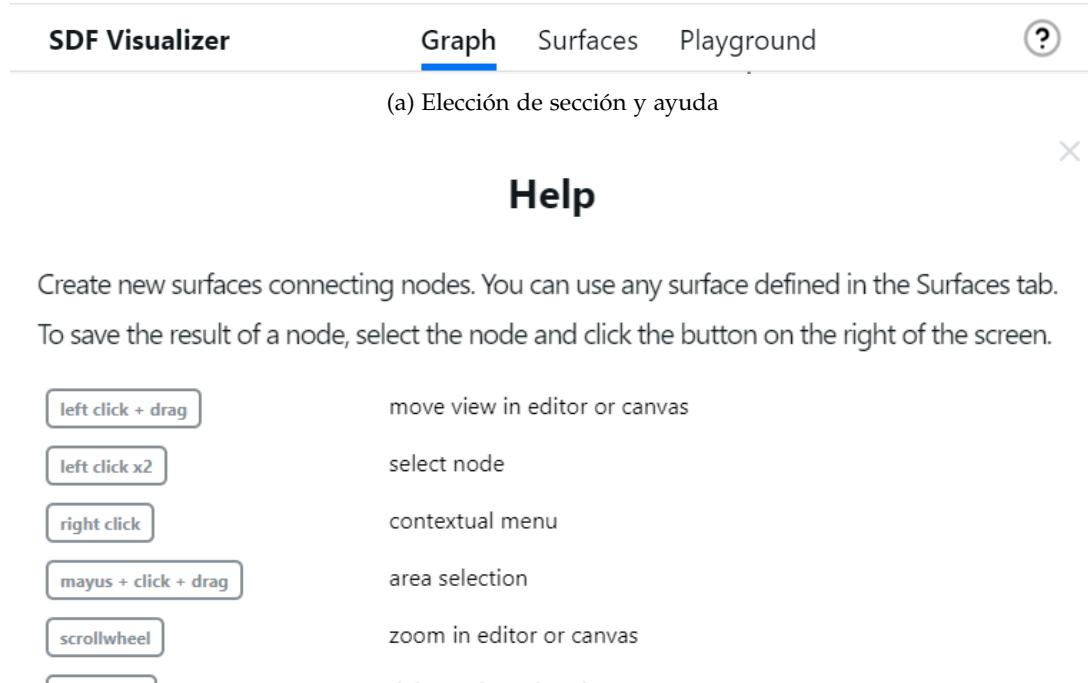


Figura 3.1: Menú superior de la aplicación

En la **sección Graph** se encuentra el editor de nodos, en el cual podremos crear nuevos a través de un menú contextual activado con el click derecho. Conectando a través de sus puertos de entrada y salida distintos tipos de nodos y modificando sus parámetros seremos capaces de crear nuevas superficies, que podremos guardar en el almacenamiento local mediante el panel lateral tras introducir un nombre válido. Existirán dos grandes tipos de nodos: los de primitiva, que permiten seleccionar una primitiva entre las existentes para ser conectada a uno o varios nodos de operaciones, que implementan las operaciones explicadas

3 Diseño, desarrollo e implementación

en la [Sección 1.4](#). Dentro de los nodos de operaciones habrá un tipo de nodo por cada familia de operación vista en esta sección: booleanos, de deformación, transformación y repetición, En la [Figura 3.3](#) se muestra un uso avanzado del editor de nodos mediante un ejemplo clásico de geometría de sólidos constructiva. También podemos mover la vista del área en la que se encuentran los nodos arrastrando con el ratón, hacer zoom con la rueda, y seleccionar varios nodos simultáneamente usando la tecla Mayus y el ratón.

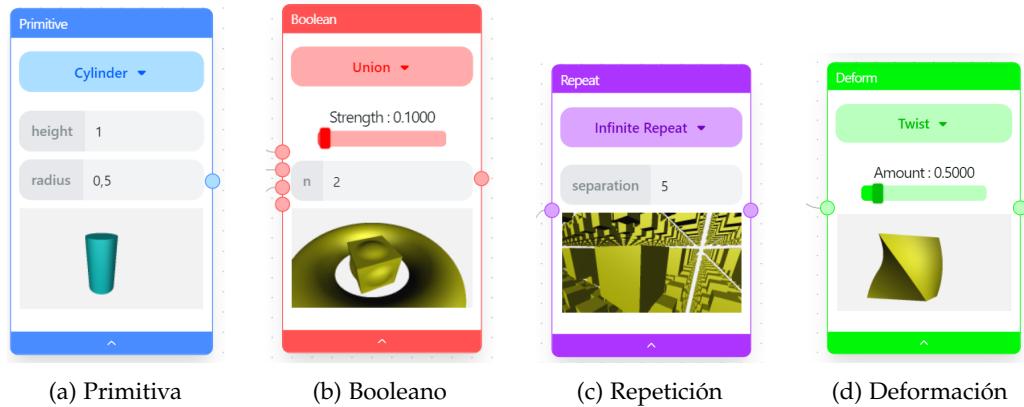


Figura 3.2: Ejemplo de tipos de nodos



Figura 3.3: Uso de la sección Graph

En el **apartado Surfaces** podemos ver un panel con las primitivas existentes en el sistema junto con información relevante sobre ellas, como sus parámetros o como fue definida (a través de ecuaciones paramétricas, implícitas o una SDF). Por defecto se incluyen varias primitivas de ejemplo, que el usuario puede modificar o eliminar pulsando los botones correspondientes. Desde esta sección el usuario será capaz de visualizar y modificar las primitivas existentes, así como crear nuevas, cuya pulsación del botón asociado abrirá un cuadro de diálogo en el que introducir toda la información necesaria.

NAME	TYPE	PARAMETERS	INPUT
Crear / Restaurar			
Sphere	Implicit	r	$x^2 + y^2 + z^2 - r$
Sphere Parametric	Parametric		$x = \frac{2 \cdot s}{s^2 + t^2 + 1}, y = \frac{2 \cdot t}{s^2 + t^2 + 1}, z = \frac{s^2 + t^2 - 1}{s^2 + t^2 + 1}$
Editar			
Plane	Parametric		$s + t, s, t$
Eliminar			
Hyperbolic Paraboloid	Parametric		$t, s, t^2 - s^2$
Torus	SDF	R, r	$\text{length}(\text{vec2}(\text{length}(p.xz)-R,p.y)) - r$
Cube	SDF	l	$\text{length}(\max(\text{abs}(p) - \text{vec3}(l), 0.0)) + \min(\max(\text{abs}(p.x) - l, \max(\text{abs}(p.y) - l, \text{abs}(p.z) - l)), 0.0)$
Ellipsoid	Parametric		$s, t, s^2 + t^2$

Figura 3.4: Tabla de primitivas

En el cuadro de diálogo veremos podremos introducir el nombre de la superficie e introducir las ecuaciones en función del tipo de entrada seleccionado en el menú superior:

- En el modo SDF se introduce una única expresión en sintaxis GLSL. Además se puede usar cualquier primitiva u operador ya existente mediante llamadas a funciones, como muestra la [Figura 3.5](#).
- En el modo Implicit ([Figura 3.6](#)) se vuelve a introducir una única expresión, que será una ecuación implícita en las variables x, y, z .
- En el caso de ecuaciones paramétricas hay seis campos de texto, pues se introducen por separado el numerador y denominador de las parametrizaciones, como ocurre en el ejemplo de la [Figura 3.7](#). Ahora se usan las variables s, t como parámetros de las ecuaciones.

En la parte inferior se encuentran además dos secciones que podemos modificar de manera opcional. La primera nos permite añadir parámetros a la expresión de la superficie que podremos modificar a través de un nodo de primitiva en la sección Graph. Para crear o modificar un parámetro se deberá indicar el símbolo que tiene este en la expresión de la función distancia, una etiqueta para mostrarlo en el editor de nodos, un valor por defecto y el método de entrada que tendrá el usuario para indicar su valor en el editor de nodos. Si se elige la entrada por valor el usuario introducirá su valor en un campo de entrada numérica convencional. Si por el contrario se elige la entrada en un rango, en el editor de nodos aparecerá un deslizador, cuyos límites serán indicados por el usuario al momento de definir el parámetro. Cada vez que se realice una modificación válida de estos campos o se elimine un parámetro, y solo entonces, se podrá volver a intentar obtener la SDF según el resto de entradas. La otra sección incluye controles para modificar los atributos del material a través de selectores de color. Todos los cambios realizados en cualquier campo se ven reflejados en tiempo real en la previsualización de la parte derecha. Si en cualquiera de los campos se detecta algún error, este será mostrado de la forma más precisa y comprensible al usuario, como se muestra en la [Figura 3.8](#). Una vez toda la información introducida es válida, se le

3 Diseño, desarrollo e implementación

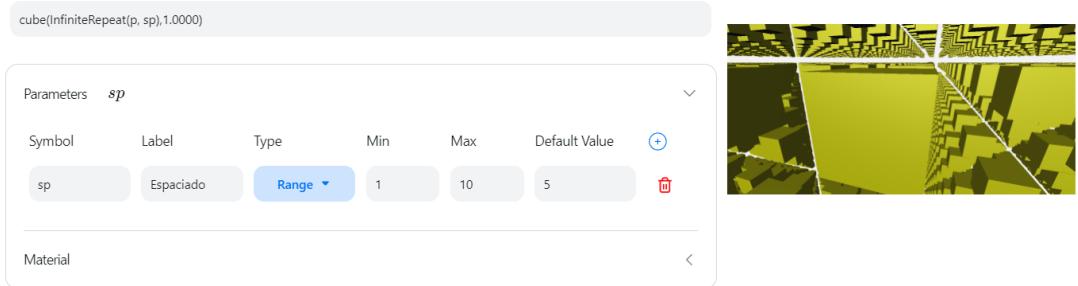


Figura 3.5: Ejemplo de creación en el modo SDF usando operadores y primitivas predefinidos

NAME	TYPE	PARAMETERS	INPUT
Sphere	Implicit	<code>r</code>	$x^2 + y^2 + z^2 - r$
Sphere Parametric	Parametric		$x = \frac{2 \cdot s}{s^2 + t^2 + 1}, y = \frac{2 \cdot t}{s^2 + t^2 + 1}, z = \frac{s^2 + t^2 - 1}{s^2 + t^2 + 1}$
Plane	Parametric		$s + t, s, t$
Hyperbolic Paraboloid	Parametric		$t, s, t^2 - s^2$
Torus	SDF	<code>R, r</code>	$\text{length}(\text{vec2}(\text{length}(p.xz) - R, p.y)) - r$
Cube	SDF	<code>l</code>	$\text{length}(\max(\text{abs}(p) - \text{vec3}(l), 0.0)) + \min(\max(\text{abs}(p.x) - l, \max(\text{abs}(p.y) - l, \text{abs}(p.z) - l)), 0.0)$
Ellipsoid	Parametric		$s, t, s^2 + t^2$

Figura 3.6: Ejemplo de creación de una esfera de radio r definida implícitamente

permite al usuario pulsar el botón de guardar. Esta acción hará que se llame a la función del contenedor de primitivas correspondiente, haciendo que se sobreescriba la primitiva original con la calculada para la previsualización (sin realizar la evaluación de los parámetros) y se reflejen los cambios tanto en el editor de nodos como en la tabla de primitivas. Finalmente, observar que en caso de que el usuario quiera crear una primitiva en lugar de editar una existente, el funcionamiento interno será el mismo, con la única excepción de que el cuadro de diálogo aparecerá completamente vacío.

La última es la llamada **sección Playground**, ya que se trata de una zona en la que el usuario puede experimentar el efecto que tiene aplicar o no las técnicas explicadas en este trabajo en tiempo real en la escena dibujada a la izquierda. Se incluyen controles para modificar el comportamiento de las sombras, la oclusión ambiental y el *antialiasing*. Se pueden añadir hasta cuatro luces a la escena, configurando la dirección, tamaño y color de cada una de ellas. También se pueden modificar las propiedades del material de cada elemento de la escena. Todas estas entradas se realizan mediante deslizadores, controles circulares o selectores de color para que sean lo más intuitivas posible (Figura 3.9). Dado que el usuario introduce la radiancia emitida de las luces mediante un color, pero en realidad este debería ser una terna RGB no acotada (parámetro S_i de la Def. 2.4), se incluyen deslizadores que multiplicarán las

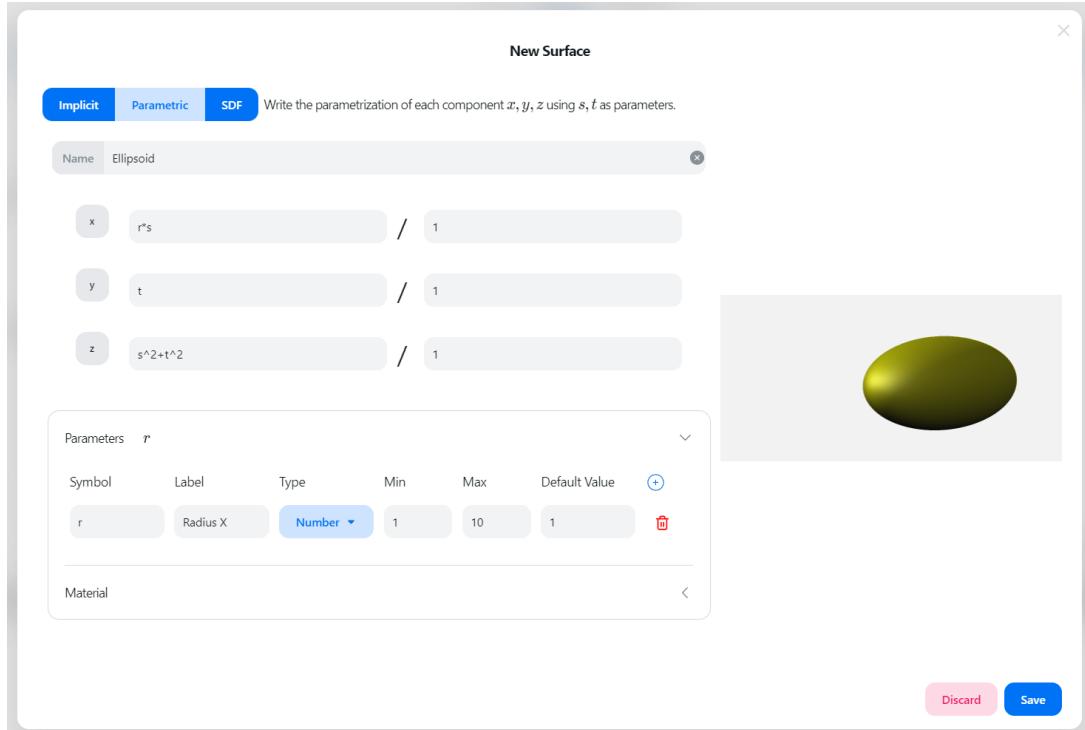


Figura 3.7: Ejemplo de creación de un elipsoide definido paramétricamente

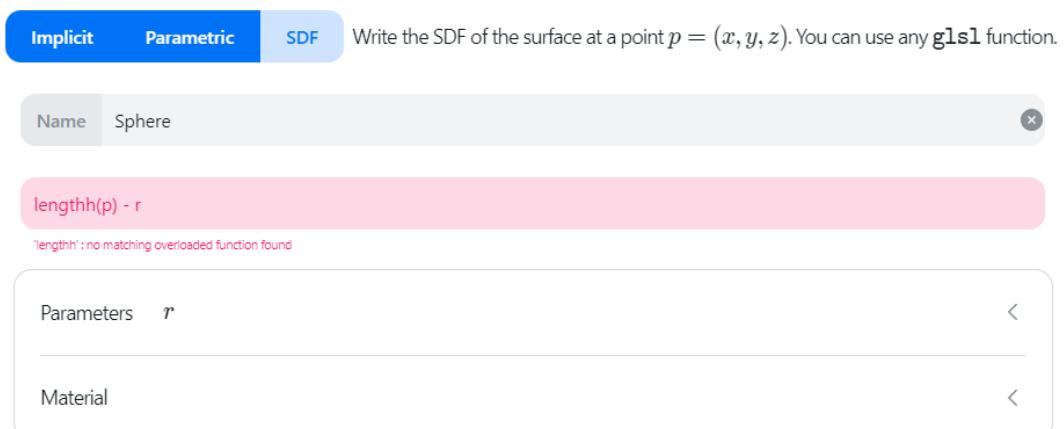


Figura 3.8: Ejemplo de error al usar un parámetro no definido

ternas acotadas introducidas como color por el usuario.

3 Diseño, desarrollo e implementación

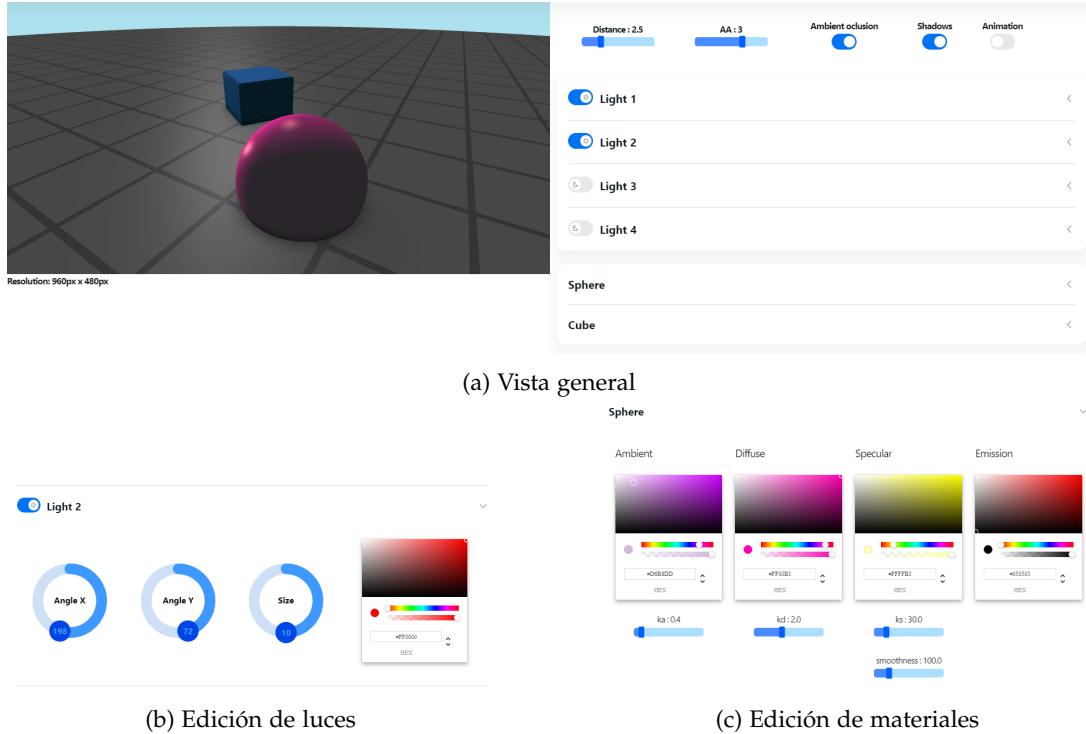


Figura 3.9: Sección Playground

3.3. Herramientas software

Entre los requerimientos especificados anteriormente se indicaba que la aplicación debía ser web para así ser accesible al mayor número de usuarios. Teniendo esto en cuenta, la elección del lenguaje para su desarrollo no podía ser otro que **JavaScript** (en tandem con **TypeScript**), pues todos los navegadores actuales son compatibles con él y es el más extendido entre la comunidad. En este punto podemos elegir si usar HTML y CSS o un *framework* de interfaces de usuario, como React, Angular o Vue. En nuestro caso se ha decidido usar **React** [59] para esta tarea, ya que es de las más populares, y por tanto cuenta con muy buen soporte, documentación y muchos paquetes de la comunidad a nuestra disposición. 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 los llamados componente funcionales, los cuales 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 desencadenen 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 desea reaccionar a cambios de ciertos *hooks*, este *hook* permite indicar sus dependencias entre corchetes.

Ejemplo 3.1. Un ejemplo de uso básico de JSX, componentes funcionales y manejo de estado en React es el siguiente.

```

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

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

  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>
  );
}

```

La principal ventaja de React sobre Angular es su uso del DOM virtual, lo que lo hace más eficiente, así como su flexibilidad y menor curva de aprendizaje. En cuanto a sus desventajas, React no posee un gestor de estado por defecto, mientras que Angular sí lo hace, siendo además una biblioteca de menor tamaño y necesitar una menor configuración inicial que React. Comparando ahora Vue con React encontramos menos diferencias, pero es cierto que Vue, con su flexibilidad a la hora de definir la estructura de la aplicación, menor tamaño y curva de aprendizaje más suave, está pensado para proyectos más pequeños y tiene un menor soporte de la comunidad.

Como acabamos de mencionar, React no cuenta con un gestor de estado por defecto. Esto significa que no contamos con una solución específica o una biblioteca integrada para administrar y mantener el estado de nuestra aplicación de manera organizada. En nuestro caso esta funcionalidad es fundamental, pues trabajamos con dos grandes categorías de datos estructurados: las primitivas que define el usuario y los nodos del editor de nodos. Para paliar esta carencia tendremos que recurrir a alguna librería externa. La más popular es Redux, cuyas principales virtudes son que tiene un patrón de flujo de datos unidireccional muy claro, lo que facilita el rastreo del flujo de datos en la aplicación, y que permite la incorporación de *middleware* personalizado. Sin embargo, cuando trabajamos con datos tan sencillos como los nuestros, el uso de Redux añadiría complejidad innecesaria y código redundante, y en su lugar se ha optado por el uso de **Zustand** [60], una librería de código abierto mucho más ligera que Redux, que requiere de menos código redundante para su uso y que se basa en el uso de *hooks* para leer el estado, más acorde con la filosofía de React, de forma que se actualizarán únicamente los componentes que dependan de los cambios relevantes en el estado, haciendo la aplicación más eficiente.

Como veremos en la siguiente sección, Zustand está basado en el uso de contenedores que contienen datos y métodos para realizar operaciones sobre ellos. Cada uno de estos contenedores es autocontenido, siendo la única forma de acceder a los datos a través de los métodos del contenedor. Esto hace que sea más fácil realizar depuración, así como evitar que haya modificaciones inesperadas desde algún componente externo. Además, es muy sencillo almacenar la información de un contenedor cualquiera en el almacenamiento local a través del método *persist* incluido, de forma que el usuario pueda disponer de los datos guardados en sesiones anteriores.

Para que el usuario pueda usar y crear primitivas usando las operaciones vistas en la [Sección 1.4](#) tenemos que implementar el editor de nodos en forma de árbol. Este editor será uno de los pilares de la aplicación, y que proporcione la mejor experiencia posible al usuario final debe ser nuestra prioridad. Es por esto que hemos decidido que el editor se base en **React Flow** [61], una librería de código abierto que permite la implementación de diagramas interactivos basados en nodos. Se trata con diferencia de la librería más completa en la fecha actual, ya que incluye multitud de funcionalidades que hacen más cómodo su uso tanto al desarrollador como al usuario final, es muy extensible, permitiendo al desarrollador definir sus propios tipos de nodos, personalizable, y está en constante evolución.

Tanto en el editor de nodos como en el panel de primitivas, el poder mostrar al usuario el resultado de sus acciones sobre la forma de la superficie en tiempo real es un gran añadido, y cambia por completo la forma en la que este interacciona con la aplicación. Para esto, necesitaremos poder visualizar una función distancia con signo mediante *spheretracing* aplicando los algoritmos de iluminación y renderizado vistos en el [Capítulo 2](#). Como se explicó en este capítulo, los requisitos para ello son un lienzo junto con un *vertex shader* y un *fragment shader*. Para la creación de la geometría del lienzo haremos uso de librería de código abierto **gl-react** [62]. Esta librería está completamente integrada con React, de forma que permite la creación de componentes reutilizables en función de nuestras necesidades, y pone su foco en la eficiencia, no por ello aportando menos funcionalidades que sus rivales. De hecho ocurre todo lo contrario, y a fecha de creación de este trabajo esta es la única que proporciona una API para el manejo de fallos de compilación del *shader* (entre otros), la cual nosotros necesitamos para poder comunicar al usuario sus errores. Otra muy buena alternativa a

gl-react es shadertoy-react [63], que es la versión adaptada a React de Shadertoy y nos proporciona un lienzo muy adaptable, fácil de usar y optimizado para dispositivos móviles. Finalmente su uso fue descartado justamente por no incluir una forma de manejar los errores de compilación de los *shaders*.

Tanto gl-react como shadertoy-react se encargan de la creación del *vertex shader*, que como vimos en la Subsección 2.1.1 es muy sencillo, y nosotros solo tenemos que encargarnos de escribir el *fragment shader*. El lenguaje en el que deberemos escribir este *shader* es **GLSL**, pues es el utilizado por la API de WebGL [64], que está a su vez basada en OpenGL y es la estándar en el ámbito de renderización de gráficos en navegadores web.

Por último, hemos visto que podemos obtener una SDF a partir de ecuaciones paramétricas e implícitas, y para ello se necesitan intensos cálculos. En particular, para permitir al usuario definir superficies mediante ecuaciones paramétricas debemos de tener una forma de trabajar con polinomios en varias variables, y así poder aplicar el Teorema 1.9. Si bien tenemos a nuestra disposición un gran número de librerías externas de cálculo simbólico, en el momento de realización de la aplicación no se encontró ninguna opción viable para trabajar con polinomios en varias variables en JavaScript de forma nativa. Como alternativas se barajó el uso de la API de Geogebra [65] o realizar llamadas a código Python que usara SageMath [66]. Sin embargo, por motivos de rendimiento y completitud de este trabajo, se decidió desarrollar una librería nativa en TypeScript para el manejo de polinomios en varias variables, cálculo de bases de Gröbner e implicitación bajo el nombre de **multivariate-polynomial**. No obstante, en determinadas circunstancias nos será conveniente apoyarnos en una librería externa. En particular, necesitaremos que la librería elegida sea capaz de proporcionarnos el árbol de expresión de un polinomio cualquiera. De entre las que incorporan esta funcionalidad, nosotros hemos decidido usar Nerdamer [67] por ser de código abierto y de las más ligeras y flexibles a la hora de trabajar con polinomios. Otra opción habría sido mathjs, pero esta no tiene soporte nativo para TypeScript, lenguaje que usaremos para escribir la librería debido a que, gracias a su tipado estático, tanto su desarrollo como su uso es menos propenso a fallos.

3.4. Implementación de la aplicación web

Ahora que tenemos una visión general de la estructura deseada para la aplicación, hagamos un estudio detallado de como los principales componentes de esta funcionan e interaccionan entre sí de manera interna.

3.4.1. Gestor de estado

El funcionamiento de Zustand consiste en el uso de contenedores, formados por atributos y métodos para trabajar con ellos. Cuando un componente quiere acceder a un contenedor, basta con que se suscriba a sus cambios a través del *hook* que proporciona Zustand: useStore. Como ya habíamos adelantado, nosotrosaremos uso de dos contenedores, uno para las primitivas definidas por el usuario 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 panel de primitivas, ya que ambos necesitan leer de él para saber cuales son las primitivas existentes, así como escribir para crear nuevas o modificar las existentes.

Este contenedor tiene la estructura mostrada en la [Figura 3.10](#). En particular, la información se almacena a través del tipo Primitive, el cual tiene los siguientes atributos.

- name: nombre que otorga a la primitiva el usuario y será mostrada por pantalla.
- id: identificador único obtenido aplicando varias modificaciones a name. Primero se le realiza normalización NFD (Forma Normal de Descomposición Canónica, por sus siglas en inglés) debido a que en Unicode algunos caracteres pueden representarse de varias maneras, ya sea como un solo carácter o como una secuencia de caracteres combinados. Usando normalización NFD garantizamos que los caracteres se almacenen y manipulen de manera consistente. Por ejemplo, el carácter “á” se descompone en los caracteres “a” y “'”. Después se revisa que se siga un formato adecuado, eliminando espacios y caracteres especiales, así como evitando que el identificador empiece por un número.
- inputMode: indica el método que usó el usuario para definir la primitiva, ya sea introducir directamente la SDF con sintaxis GLSL, la ecuación implícita o la parametrización racional.
- input: cadena de texto con la ecuación introducida por el usuario. Dado que al usar parametrizaciones se requiere de seis ecuaciones (tres numeradores y tres denominadores), su valor se guarda en un array.
- sdf: función distancia con signo obtenida tras procesar la entrada del usuario.
- parameters: parámetros asociados a la primitiva y presentes en la función distancia con signo, que podrán ser usados en el editor de nodos para interactuar con ella. Cada parámetro viene representado por la estructura Parameter, que contiene el símbolo del parámetro presente en la expresión, una etiqueta para ser mostrado en el editor de nodos, y si su valor debe estar acotado en cierto rango.
- fHeader: cabecera de la función GLSL asociada a la primitiva. La función tendrá como nombre el id, y recibe como argumentos el punto donde se quiera evaluar la SDF y cada uno de los parámetros en parameters.
- material: contiene las componentes especular, difusa y ambiental, así como el coeficiente de brillo del material asociado a la primitiva.

En secciones próximas veremos la utilidad de todos estos atributos.

Ejemplo 3.2. Para definir una esfera centrada en el origen mediante su ecuación implícita y poder controlar su radio en el editor de nodos, se crearía la siguiente primitiva.

```
{
  id: "sphere",
  name: "Sphere",
  inputMode: InputMode.Implicit,
  input: ["x^2 + y^2 + z^2 - r", "1", "", "1", "", "1"],
  parsedInput: "length(p)-r",
  parameters: [
    { symbol: "r", label: "Radius", defaultVal: 1.0, type: "range", range: [0, 100] }
  ],
}
```

```

fHeader: "sphere(vec3 p, float r)",
material: {
    specular: [1.0,1.0,1.0],
    diffuse: [0.0,1.0,0.0],
    ambient: [0.2,0.2,0.2],
    smoothness: 10.0
}
}

```

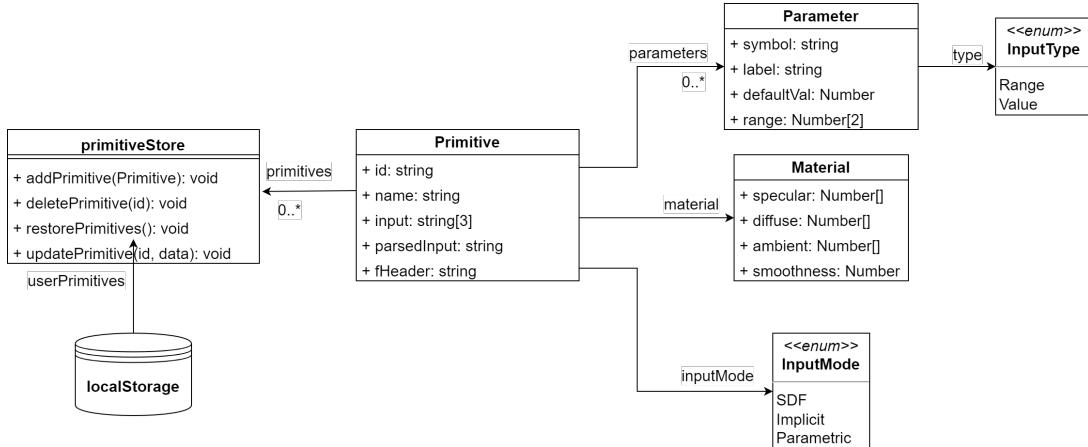


Figura 3.10: Diagrama de clases del contenedor de primitivas

A modo de ejemplo, el esqueleto del contenedor de primitivas se muestra en la Figura 3.11.

3.4.2. Lienzo

Como comentamos en la sección anterior, la creación tanto del lienzo como del *vertex shader* corre a cuenta de gl-react, y nosotros solo tendremos que preocuparnos de pasarle como una cadena de texto el *fragment shader* al componente de gl-react. Sin embargo, que no tengamos que encargarnos nosotros de definir estos elementos no significa que no debamos entender su proceso. A continuación estudiaremos los conceptos básicos de como realizar la creación del lienzo y *vertex shader* [68] con WebGL, así como nuestra implementación del *fragment shader* para realizar *spheretracing*, todo esto apoyándonos en la teoría explicada en la Subsección 2.1.1.

Empezamos estudiando el *vertex shader* usando WebGL directamente con HTML y JavaScript. Sabemos que se ejecuta una instancia de este *shader* para cada vértice de la malla, y que recibirá las coordenadas del vértice adecuado a través de un atributo de entrada. Para que esto sea posible primero tendremos que haber definido nosotros cada uno de los cuatro vértices que conforman el lienzo. Cuando se trabaja con un gran número de vértices, lo común es trabajar con tablas (arrays) de vértices e índices. En la tabla de vértices se declaran las coordenadas de los vértices, y en la de índices se indica como estos se conectan para formar los triángulos que conforman la malla. Esto permite que no haya vértices repetidos, y

```
const defaultPrimitives =[  
{  
  id: "torus",  
  name: "Torus",  
  inputMode: InputMode.SDF,  
  input: ["length(vec2(length(p.xz)-R,p.y)) - r", "1", "", "1", "", "1"],  
  // ...  
},  
// ...  
];  
  
export const usePrimitiveStore =create()(  
  persist(  
    (set, get) =>{  
      primitives: defaultPrimitives,  
  
      updatePrimitive(id, data) { // ... },  
  
      deletePrimitive(id) { // ... },  
  
      addPrimitive(prim) { // ... },  
  
      restorePrimitives() { // ... },  
    }),  
  {  
    name: 'user-primitives', // ID del contenedor en almacenamiento local  
  }  
)  
)
```

Figura 3.11: Estructura básica del contenedor de primitivas de Zustand

por tanto que se reduzca la cantidad de datos que el sistema debe transmitir y procesar. En nuestro caso sin embargo solo tenemos cuatro vértices, y por simpleza podemos permitirnos prescindir de la tabla de índices.

Para conseguir esto empezamos enviando los datos que usaremos de la CPU a la GPU usando el método `gl.bufferData`, que recibe los siguientes argumentos.

- El tipo de búfer que se creará, en este caso `gl.ARRAY_BUFFER`, pues es un array de coordenadas.
 - Puntero con los datos que se cargarán en el búfer, en este caso los vértices del lienzo.
 - Indicación de como se utilizarán los datos del búfer. Con `gl.STATIC_DRAW` podemos indicar que los datos no cambiarán una vez hayan sido cargados, siendo estos de solo lectura.

Observamos en la [Figura 3.12](#) que como necesitamos dos triángulos, y por tanto dos grupos de tres vértices, hemos tenido que repetir dos vértices. Para formar los triángulos hacemos uso del método `gl.drawArrays`, el cual renderiza primitivas a partir de un array de datos.

```

function setGeometry(gl) {
    gl.bufferData(
        gl.ARRAY_BUFFER,
        new Float32Array([
            -2, -1,
            -2, 1,
            2, 1,
            -2, -1,
            2, 1,
            2, -1]),
        gl.STATIC_DRAW);
}

```

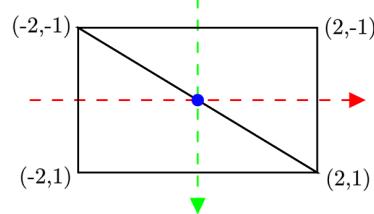


Figura 3.12: Declaración de vértices del lienzo

Como queremos dibujar triángulos, la primitiva será `GL_TRIANGLES`. Al indicar esta primitiva, se tomarán los vértices del búfer de tres en tres y se dibujará un triángulo con ellos, así que es imprescindible que el orden en el que los hayamos definido sea el correcto. Adicionalmente, indicamos que el índice de inicio es el primero y que use los seis vértices definidos con los parámetros `offset` y `count` respectivamente.

```

function drawScene() {
    var primitiveType =gl.TRIANGLES;
    var offset =0;
    var count =6;
    gl.drawArrays(primitiveType, offset, count);
}

```

Una vez definida la geometría, quedaría escribir el *vertex shader* [69], para el cual recordemos que teníamos que calcular las matrices de transformación necesarias. Sin embargo, dado que WebGL es una API en 2D, solo necesitamos pasar directamente de coordenadas de objeto a recortadas, y en nuestro caso podemos no preocuparnos de realizar la proyección de la geometría ni operación sobre los vértices alguna. Para pasar atributos del *shader* se utiliza la palabra reservada `uniform`, cuyo nombre viene de que se trata de una variable global de la cual su valor no cambia de una ejecución a otra. Estas variables solo pueden ser de ciertos tipos definidos por GLSL, siendo los más usados los siguientes.

- Escalares: `int`, `float`, `bool`.
- Vectores de n flotantes: `vec n` , donde $n \in \{2, 3, 4\}$. Este tipo de datos tiene la peculiaridad de que permiten acceder a sus valores como si se tratases de un struct de C de varias formas, por ejemplo como `.r`, `.g`, `.b`, `.a`, haciendo referencia a que una de las funciones de este tipo de datos es asignar el color del píxel. A continuación se muestran todas las formas de acceder a las entradas de los vectores.

```

vec4 vector;
vector[0] =vector.r =vector.x =vector.s;
vector[1] =vector.g =vector.y =vector.t;

```

```
vector[2] =vector.b =vector.z =vector.p;
vector[3] =vector.a =vector.w =vector.q;
```

Otra característica muy útil de los vectores es la conocida como *swizzling*, que permite acceder a varias entradas simultáneamente usando una sintaxis similar a la anterior.

```
vec3 v =vec4(1.0, 0.0,0.0);
v.yz   = vec2(0.5, 0.5); // v = [1.0, 0.5, 0.5]
v.br   = vec2(0.0)      // v = [0.0, 0.5, 0.0]
vec2 zero =v.sp
```

Dado que las coordenadas que pasamos como *uniform* eran bidimensionales, usaremos el tipo *vec2*.

- Matrices de flotantes cuadradas y dimensión $n \times n$: *matn*, donde $n \in \{2, 3, 4\}$.
- Tipos opacos: no tienen un valor como tal, sino que hacen referencia a un objeto externo, como una textura.

En la [Subsección 2.1.1](#) decíamos que la única forma de comunicación entre *shaders* es a través de paso de atributos de entrada y salida, y que el objetivo del *vertex shader* era el de calcular la posición transformada del vértice como un atributo de salida. La sintaxis de estos atributos es diferente a la de los pasados por el usuario, y en lugar de *uniform* se escribe *in* o *out* en función de si se trata de un atributo de entrada o salida. Sin embargo, OpenGL define por defecto ciertos atributos con una funcionalidad específica en cada tipo de *shader*, y no será necesario declararlos para usarlos. En el caso del *fragment shader*, esto ocurre con el atributo *vec4 gl_Position*, que es el que debemos utilizar para almacenar las coordenadas recortadas transformadas. Así, el *vertex shader* más básico y que es suficiente para nosotros es el siguiente.

```
<script id="vertex-shader" type="x-shader/x-vertex">
    in vec4 gl_Position; // no necesario
    uniform vec2 v_loc;

    void main() {
        gl_Position =vec4(v_loc, 0,1);
    }
</script>
```

Es importante que el tipo del script sea *x-shader/x-vertex* para que cuando posteriormente indiquemos a WebGL la ubicación del *shader* a través del método *createShaderFromScriptElement* sepa como tratar su contenido.

Con toda esta información y a modo de resumen, presentamos a continuación un escenario mínimo de uso de WebGL para la creación del lienzo.

```
// Crear contexto WebGL
var canvas =document.getElementById("canvas");
```

```

var gl = canvas.getContext("experimental-webgl");

// Crear y asignar shaders
var vertexShader = createShaderFromScriptElement(gl, "2d-vertex-shader");
var fragmentShader = createShaderFromScriptElement(gl, "2d-fragment-shader");
var program = createProgram(gl, [vertexShader, fragmentShader]);
gl.useProgram(program);

// Vincular atributos con variables de atributos en shaders:
var positionLocation = gl.getAttribLocation(program, "v_loc");

// Declaracion de vertices
var buffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
gl.bufferData(
    gl.ARRAY_BUFFER,
    new Float32Array([
        -2, -1,
        -2, 1,
        2, 1,
        -2, -1,
        2, 1,
        2, -1]),
    gl.STATIC_DRAW);

gl.enableVertexAttribArray(positionLocation);
gl.vertexAttribPointer(positionLocation, 2, gl.FLOAT, false, 0, 0);

// Dibujar lienzo
gl.drawArrays(gl.TRIANGLES, 0, 6);

```

En este código estamos suponiendo que hemos creado también un *fragment shader* de forma similar al *vertex shader*. Un ejemplo básico en el que todo el lienzo sería blanco sería:

```

<script id="2d-fragment-shader" type="x-shader/x-fragment">
    in vec4 gl_FragColor; // no necesario

    void main() {
        gl_FragColor = vec4(1.0);
    }
</script>

```

En nuestro caso sabemos que tendremos que trabajar más que esto, pues es en este *shader* donde implementaremos el algoritmo de *spheretracing*.

Pasamos por tanto ahora a estudiar el *fragment shader*. Dado que el código es el mismo usemos directamente WebGL o gl-react, vamos a ver cómo lo hemos implementado en el contexto de nuestra aplicación de React, ya que este tiene la particularidad de que tendremos que pasárselo a gl-react en forma de *string*. Gran parte de su contenido es estático, incluyendo las constantes usadas en el algoritmo de *spheretracing* (iteraciones máximas, precisión,

distancia dibujado), el código de los operadores vistos en la [Sección 1.4](#) y los algoritmos de cálculo de iluminación, oclusión ambiental y *antialiasing*. Esto hace que el *shader* puede estar definido en un archivo independiente como una función que recibe una serie de parámetros y devuelve una cadena de texto. Sin embargo, hay elementos del *shader* que son dinámicos debido a que el usuario puede modificarlos de una forma u otra. Estos serán pasados como *props* al componente de lienzo, que a su vez los pasará al *fragment shader* cuando detecte un cambio en alguno de ellos.

Listing 3.1 Definición del procesador de fragmentos

```
export const fs =(sdf, primitives, aa, ao, shadow) =>{
  return `
    // fragment shader ...
  `
}
```

El más importante de los argumentos recibidos por el componente es la propia función distancia con signo en forma de cadena de texto, cuyo valor cambia cada vez que se cambia una primitiva o se modifica el número de operadores aplicados en el árbol actual. Lo ideal sería pasar todos los parámetros al *shader* a través de *uniforms*, pero estos no pueden ser del tipo *string*, lo que hace imposible usarlos para la función distancia, y aunque se pudiera, sería complejo e ineficiente implementar en GLSL un intérprete de estas cadenas. En su lugar, no tenemos más remedio que recompilar el *shader* cada vez que se modifica la función distancia. No obstante, el valor de los parámetros para las primitivas y operadores sí que pueden ser pasados como *uniforms*, pues son siempre flotantes, y será de vital importancia para la fluidez de la aplicación. Esto es debido a que muchos de estos parámetros pueden ser modificados mediante deslizadores, que pueden cambiar su valor varias veces por segundo. Si además este nodo tiene hijos, esto provocaría la recompilación de múltiples *shaders* varias veces por segundo, haciendo que la interacción del usuario con la aplicación sea muy tosca y poco satisfactoria. Para conseguir esto, cada vez que se añade una primitiva u operación con parámetros a la expresión de un nodo, se define un nuevo *uniform* por cada parámetro. Dado que el nombre de los parámetros puede coincidir entre varios nodos, el nombre del *uniform* tiene como prefijo el identificador del nodo, seguido del propio nombre del parámetro. Observamos que añadir las definiciones de estos *uniforms* no supone ninguna compilación adicional del *shader*, ya que esta ocurría igualmente al tener que actualizar la expresión de la SDF. Finalmente, el valor de los parámetros, cuyo valor se gestiona en el contenedor del editor de nodos, es pasado al *shader* a través del *prop uniforms* de gl-react.

Para modificar la menor cantidad de código, existe una función predefinida *sdf* vacía en el *shader*, que recibe un punto *p* y devolverá la expresión evaluada por GLSL de la función distancia en dicho punto, siendo esta la equivalente a la función distancia con signo ϕ del [Algoritmo 11](#). Además, dado que al usuario se le permitirá usar directamente las variables *x, y, z* al introducir una función distancia con signo en lugar de tener que escribir *p.x* o *p[0]* para acceder a las componentes del punto *p*, estas se declaran previamente con los nombres *x, y, z*. Sin embargo, no será esta función *sdf* la que evalúen todos los algoritmos de renderizado para sus cálculos, sino la función *map*, que devuelve una estructura del tipo *Surface* conteniendo la distancia con signo a la superficie desde el punto actual y el material que tiene asignado la superficie. A su vez, este material viene representado por la estructura *Material*,

que encapsula los atributos de reflexión difusa, especular, ambiental, etc. La razón de usar esta función es que se pueda trabajar con varias superficies al mismo tiempo, cada una con un material, lo cual permite un mayor grado de control sobre el aspecto de la escena y, por ejemplo, ser capaz de interpolar los materiales de dos superficies al realizar operaciones booleanas, como se mostró en la [Figura 1.4](#).

Otro factor cambiante en el *fragment shader* son las primitivas disponibles, ya que el usuario podrá crear nuevas superficies a partir de ellas, y por tanto deberán estar declaradas en el *shader*. Al igual que antes, deberemos recompilar el *shader* cada vez que se realice un cambio sobre las primitivas almacenadas, del cual nos notificará el *hook* del contenedor de Zustand, ya que tendremos que modificar el código fuente del *shader*. Cada primitiva tendrá asignada una función en el *shader*, recibiendo el punto *p* en el que ser evaluada y un argumento por cada parámetro de la primitiva. Así, cuando se detecte un cambio se generará una cadena de texto con la definición de todas las primitivas, que la función *fs* recibirá como argumento e insertará en el código del *shader*. Ocurre lo mismo con los parámetros que define el usuario, pues deberemos declarar un *uniform* para posteriormente poder asignarles valores a través de *gl-react*. El valor de estos puede provenir tanto del contenedor de nodos como del de primitivas. Cuando el lienzo sea usado dentro de un nodo del editor, el valor de los parámetros es introducido por el usuario en cada uno, y se almacena en el contenedor de nodos. En el resto de casos, el usuario no introduce de manera explícita estos valores, así que se evaluarán en el valor por defecto almacenado en el contenedor de primitivas (cuando el usuario modifica un parámetro en el diálogo de edición, lo que está modificando en realidad es su valor por defecto).

Listing 3.2 Ejemplo de declaración de primitiva

```
float Sphere(vec3 p, float radius){
    return length(p) -r;
}
```

Resulta también interesante permitir al usuario elegir qué técnicas y algoritmos quiere utilizar para el renderizado, y que así pueda apreciar las diferencias en tiempo real. Para ello usaremos *uniforms* asociados a los algoritmos de sombras y oclusión ambiental, así como al factor de escalado del *antialiasing*, y en función de su valor aplicaremos los algoritmos de una forma u otra.

El resto de parámetros variables del *shader* se pueden pasar mediante *uniforms*, pues son flotantes y vectores, incluyendo los que siguen.

- Material de la primitiva con la estructura mostrada en la [Figura 3.10](#) a través de varios *vec3* y *float*.
- Resolución del lienzo en pixels como un *vec2*, pasada desde el componente de React.
- 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 a través del ratón y se utilizan para obtener la matriz de cámara presentada en la ecuación (2.1.2).

Varios de estos parámetros son los que se usan en el apartado Playground de la aplicación.

Una vez se ha hecho el cómputo del color del píxel actual, este valor debe ser almacenado en el atributo de entrada `gl_FragColor`, que al igual que ocurría con `gl_Position` es pasado automáticamente. A continuación se presenta el contenido básico del *fragment shader* coloreado según la sintaxis GLSL, pero hay que recordar que en realidad lo que se devuelve es una cadena de texto. En él se realizan todos los cálculos mostrados en el Capítulo 2 en GLSL, pero aquí mostramos los más importantes, tales como el cálculo del marco cartesiano de la cámara, del sistema de coordenadas del lienzo, *antialiasing*, sombras, etc.

Listing 3.3 Procesador de fragmentos

```
// Constants
const int MAX_MARCHING_STEPS=255;
const float MIN_DIST=0.;
const float MAX_DIST=100.;
const float PRECISION=.0001;
const float EPSILON=.0005;
const float PI=3.14159265359;

// Uniforms
uniform vec2 u_resolution;
uniform vec3 u_ambient;
uniform int AO;
uniform int SHADOWS;
uniform int AA;
// ...

float sdf(vec3 p){
    float x =p.x;
    float y =p.y;
    float z =p.z;

    return ${sdf};
}

struct Material
{
    vec3 ambient; float ka;
    vec3 diffuse; float kd;
    vec3 specular; float ks;
    vec3 emission;
    float smoothness;
};

struct Surface{
```

```

float sd;
Material mat;
};

// == OPERATORS ==
float Union(float a,float b,float k,float n,out float interp){
    if(k==0.)
        return min(a,b);

    float h=max(k-abs(a-b),0.)/k;
    float m=pow(h,n)*.5;
    float s=m*k/n;

    interp=a<b?m:1.-m;
    return(a<b)?a-s:b-s;
}

vec3 Twist(in vec3 p,in float k){
    float c=cos(k*p.y);
    float s=sin(k*p.y);
    mat2 m=mat2(c,-s,s,c);
    vec3 q=vec3(m*p.xz,p.y);
    return q;
}

vec3 InfiniteRepeat(in vec3 p,in float s){
    return mod(p+.5*s,s)-.5*s;
}

// ...

// == USER PRIMITIVES ==
${primitives}$

// == SDF ==
float sdf(vec3 p){
    float x =p.x;
    float y =p.y;
    float z =p.z;
    float interp;

    return ${sdf};
}

Surface map(vec3 p){
    Material mat =Material(
        u_ambient, // ambient
        u_ka, // ka
        u_diffuse, // diffuse
        u_kd, // kd
        u_specular, // specular
        u_ks, // ks

```

3 Diseño, desarrollo e implementación

```
        u_emission, // emission
        u_smoothness // smoothness
    );
    float d =sdf(p);

    return Surface(d, mat);
}

Surface rayMarch(vec3 ro,vec3 rd, float start, float end){
    float depth=start;
    Surface co;

    for(int i=0;i<MAX_MARCHING_STEPS;i++){
        vec3 p =ro + depth*rd;
        co=map(p);
        depth +=co.sd;
        if(co.sd<PRECISION||depth>end)
            break;
    }

    co.sd=depth;

    return co;
}

// == ILUMINATION ==
vec3 lighting(vec3 pos, vec3 rd, vec3 nor, Surface s){
    vec3 result =u_ambientEnv +s.mat.emission;
    float occ =AO>0 ?calcAO(pos,nor) :1.0;

    for(int i=0; i<4; i++){ // Fijamos el maximo de luces en 4
        if(u_lightsEnable[i] >0){
            vec3 Li =normalize(vec3(u_lightsPos[3*i], u_lightsPos[3*i+1],
                u_lightsPos[3*i+2]));
            vec3 lColor =vec3(u_lightsColor[3*i], u_lightsColor[3*i+1],
                u_lightsColor[3*i+2]);
            vec3 h =normalize(Li-rd);
            float NLi =max(0.,dot(nor,Li));
            float NH =max(0.,dot(nor,h));

            float shadow =SHADOWS>0 ?calcSoftshadow(pos,Li,.02,2.5,u_lightsSize[i]) :1.0;

            vec3 amb =s.mat.ka*s.mat.ambient;
            vec3 dif =s.mat.kd*s.mat.diffuse*NLi;
            vec3 spe =s.mat.ks*s.mat.specular*pow(NH,s.mat.smoothness);

            spe*=.04+.96*pow(clamp(1.-dot(h,Li),0.,1.),5.);
            result +=lColor*(amb+dif+spe)*occ*shadow;
        }
    }

    return result;
}
```

```

}

// == CARTESIAN SYSTEM FOR THE CAMERA ==
mat3 camera(vec3 cameraPos,vec3 lookAtPoint){
    vec3 cd=normalize(lookAtPoint-cameraPos);
    vec3 cr=normalize(cross(vec3(0.,1.,0.),cd));
    vec3 cu=normalize(cross(cd,cr));

    return mat3(-cr,cu,-cd);
}

// == MAIN ==
void main()
{
    const vec3 backgroundColor=vec3(0.95);
    const vec3 lookAt=vec3(0.);

    vec3 col=vec3(0.);
    vec3 eye=vec3(3.,3.,5.);
    mat3 rot=(RotateY(u_cameraAng.x)*RotateX(u_cameraAng.y));
    eye=rot*eye*u_zoom;
    mat3 cam =camera(eye,lookAt);

    for( int m=0; m<AA; m++ ){
        for( int n=0; n<AA; n++ )
        {
            vec2 o =vec2(float(m) +0.5,float(n) +0.5) / float(AA) -vec2(0.5);
            vec2 uv=((gl_FragCoord.xy+o) -0.5*u_resolution.xy) / u_resolution.y;

            vec3 rayDir=cam*normalize(vec3(uv,-1));
            Surface co=rayMarch(eye,rayDir,MIN_DIST,MAX_DIST);

            if(co.sd>MAX_DIST){
                col+=backgroundColor;
            }
            else{
                vec3 p=eye+rayDir*co.sd;
                vec3 normal=calcNormal(p);

                col +=lighting(p, rayDir, normal, co);
            }
        }
    }

    col /= float(AA*AA);
    gl_FragColor=vec4(col,1.);
}

```

Como se comentó en la [Sección 3.3](#), unos de los objetivos de la aplicación es que sea lo más accesible posible al usuario, siendo la detección y comunicación de errores esencial, y que la elección de la librería gl-react fue en gran parte debida a que incorpora la posibilidad

de instalar manejadores para multitud de eventos. Para ello debemos crear un centinela, o Visitor, y asignas manejadores a los eventos que consideramos oportunos, en nuestro caso `onSurfaceDrawError`. Este nos permitirá capturar y reaccionar ante los fallos de compilación del *shader*, así como acceder al mensaje de error. El mensaje obtenido suele ser muy extenso, pero siempre tiene una estructura similar, de forma que podemos extraer la información útil para el usuario y mostrársela en tiempo real. Como ejemplo, observemos el siguiente error.

```
exampleSDF(p, 1.0000): gl-shader: Error compiling vertex shader helloGL:
ERROR: 0:270:'lengthh' :no matching overloaded function found

Error compiling vertex shader helloGL:
268:         float z =p.b;
269:
270:         return lengthh(p) -r;
```

La información que interesa al usuario es la que está en la segunda línea. Con la expresión regular /ERROR: \d+:\d+:(.*)/ podemos buscar el patrón `ERROR:` seguido de dos grupos de uno o más dígitos (`\d+`) separados por dos puntos . Luego, nos quedamos con el texto que aparezca después del patrón hasta el final de la línea con `(.+)`, obteniendo el mensaje que queríamos. Adicionalmente, el componente recibe un prop opcional `onError` con una función que actúe cuando haya un fallo, para que si el lienzo es hijo de un componente mayor este también pueda reaccionar, como veremos en la [Subsección 3.4.5](#).

Teniendo en cuenta todo lo explicado, ya podemos entender la estructura básica del componente `Lienzo` de React.

Listing 3.4 Componente del lienzo

```
// Importar fragment shader
import {fs } from "./fragmentShader";

// Obtener las funciones de las primitivas del contenedor de Zustand
const selector =() =>(store: any) =>{
    savedPrimitives: store.primitives.map((p: any) =>
        float ${p.fHeader}{

            float x = p.r;
            float y = p.g;
            float z = p.b;

            return ${p.parsedInput};
        }\n
    ).join("\n"),
);

function Lienzo(props: {
    sdf: string;
    primitives: string;
    width: number | null;
```

```

height: number |null;
onError?: (e: string) =>void;
uniforms?: any[];
material: Material;
}) {
    const {savedPrimitives }=usePrimitiveStore(selector(), shallow);
    const [compileError, setCompileError] =useState(false);
    const [errorMsg, setErrorMsg] =useState("");
    const [shader, setShader] =useState<ShaderIdentifier>(defaultShader);
    // ... Otras variables ...
    const visitor =new Visitor(); // Centinela de gl-react

    // Manejador de error de compilacion del shader
    visitor.onSurfaceDrawError =(e: Error) =>{
        if (props.onError) props.onError(e.message);
        console.warn(`ERROR COMPILING SHADER ${props.sdf}: `, e.message);
        setCompileError(true);
        setErrorMsg(e.message);
        return true;
    };

    // Recompilar cuando recibimos nuevo SDF o primitivas
    useEffect(() =>{
        if(props.sdf ===""){

```

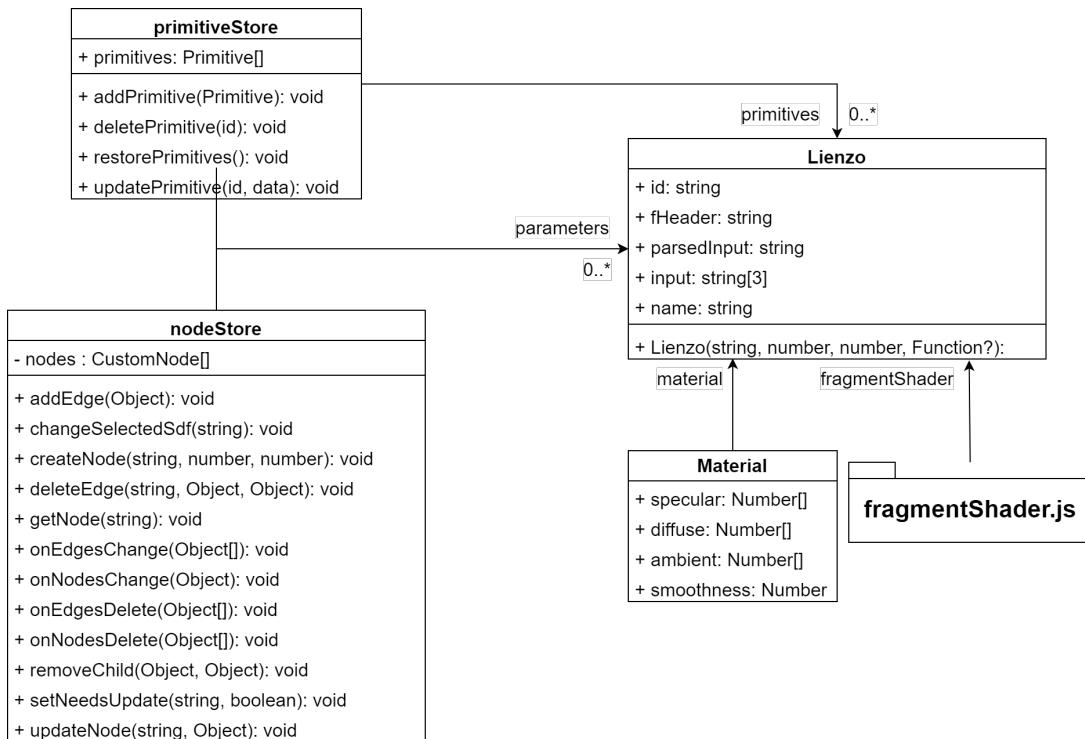


Figura 3.13: Diagrama de clases del componente Lienzo

```

        setCompileError(true);
        return;
    }

    setCompileError(false);
    setShader(fs(props.sdf, props.primitives));
}, [props.sdf, props.primitives]);

// ... manejadores de ratón (zoom, arrastrar, etc) ...

return (
<div>
    <Surface
        visitor={visitor}
        width={props.width || 100}
        height={props.height || 100}
    >
        <Node
            shader={shader}
            uniforms={{
                u_specular, u_diffuse, u_smoothness, u_emission,
                u_ka, u_ks, u_kd,
                u_ambientEnv, u_ambient,
                u_cameraAng, u_zoom,
                u_resolution,
                u_lightsPos, u_lightsColor, u_lightsSize
            }}
        />
    </Surface>
</div>
)
}

export const Shader = memo(MyShader);

```

3.4.3. Editor de nodos

Cada nodo en React Flow tiene puertos de entrada (solo permiten la conexión con un nodo) y salida (permiten conectarse a varios nodos), así como un cuerpo que representa su contenido mediante un componente de React o una expresión JSX. Ya sabemos que React Flow permite definir tipos de nodos según nuestras necesidades, y en nuestro caso usaremos dos categorías principales: nodos de primitivas y nodos de operaciones.

La estructura de todos estos nodos será similar. Todos tendrán 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 estas puedan tener, 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, los parámetros

sí dependen 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`.

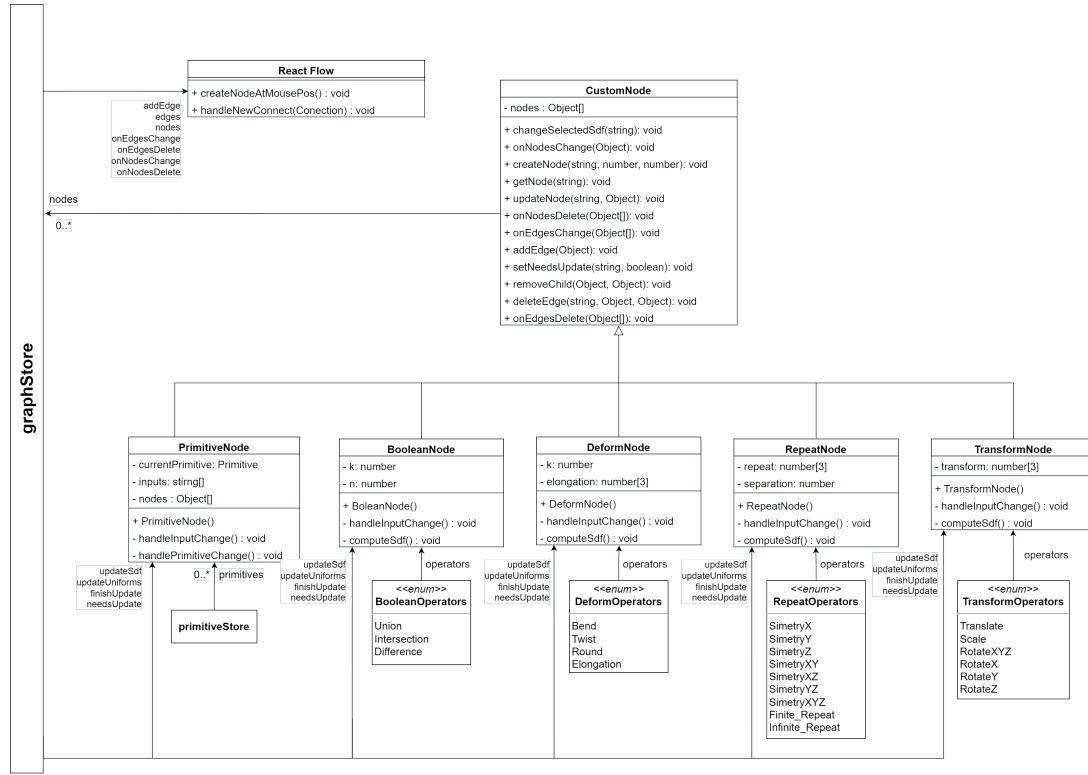


Figura 3.14: Diagrama de clases de editor de nodos

El **nodo de primitiva** es el más sencillo, pues su única función es proporcionar una primitiva base sobre la que realizar operaciones y no tiene ningún puerto de entrada. Para poder mostrar los controles adecuados según el tipo de primitiva, basta leer los parámetros del contenedor de primitivas, y según los valores que introduzca el usuario, obtener la expresión de la SDF.

Entre los **nodos de operaciones**, los 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, incrementando el número de puertos de entrada según se conecten nuevas primitivas. La razón de permitir más de dos primitivas de entrada cuando los operadores booleanos son binarios es que si se quiere realizar una misma operación de forma reiterada sobre varias primitivas se necesitarían crear muchos nodos. Trabajar con tantos nodos sería tedioso y haría mucho menos legible el editor. Así, si un nodo booleano recibe las primitivas A_i con $i \in \{1, \dots, n\}$, irá aplicando la operación de forma sucesiva sobre el resultado anterior según el orden de conexión. Por ejemplo, para la unión tendríamos

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

De manera interna React Flow usa Zustand para almacenar la información relativa a los identificadores de los nodos, sus posiciones, conexiones, etc. Nosotros extenderemos esta funcionalidad para poder usar el modelo de generación de superficies en forma de árbol presentado en la [Sección 1.4](#) a través de los métodos `onNodesChange`, `onEdgesChange`, etc. que vemos en el diagrama de clases de la [Figura 3.14](#). Para ello necesitaremos almacenar de manera adicional para cada nodo la información relativa a su SDF actual, sus entradas y sus nodos hijos. Cuando un nodo detecte una nueva conexión en algún puerto de entrada (`onEdgesChange`) se leerán las SDFs de todos ellos, y junto con los propios parámetros del nodo se actualizará la SDF del nodo. Cuando se elimina alguna conexión (`onEdgesDelete`, en el caso de los nodos diferentes al booleano simplemente la SDF pasa a ser indefinida. En el caso del booleano, habrá que tener en cuenta si todavía queda alguna entrada conectada, y en este caso 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.

También debemos actualizar los nodos hijos de un nodo cuando el usuario modifique el valor de alguno de sus parámetros. La notificación del nodo padre al hijo se hace a través del método expuesto del contenedor de nodos `updateSdf`, ya que cada instancia de un nodo no es consciente de sus hijos, sino que esa información se encuentra en dicho contenedor. Desde el contenedor se pondrá a `true` el atributo `needsUpdate` del nodo hijo, que como estará suscrito al contenedor, recibirá los nuevos datos de su padre y se volverá a renderizar. Una vez se haya actualizado, con el método `finishUpdate` el valor de `needsUpdate` volverá a ser `false` y notificará a todos sus hijos, repitiendo el proceso. De esta forma conseguimos que el cambio original se expanda por todo el subárbol cuya raíz es el nodo modificado por el usuario.

Para mejorar la experiencia del usuario almacenaremos además la información relativa a las entradas que ha proporcionado al nodo (elección de operador o primitiva y valores de los parámetros). La razón de que hagamos esto es que si cambiamos de pestaña dentro de la aplicación, al no estar renderizando el editor de nodos esta información se perdería, y el usuario no sería capaz de retomar la construcción anterior. Manteniendo una copia de esta información podremos restaurar el estado cuando el usuario vuelva a la pestaña del editor.

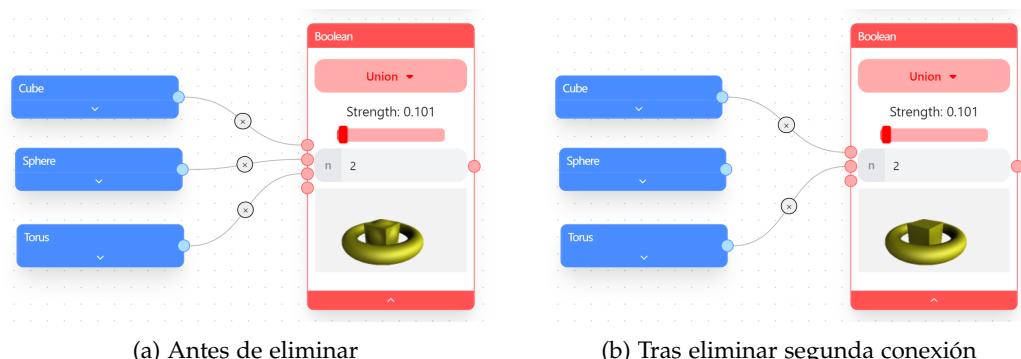


Figura 3.15: Ejemplo de eliminación de conexión en nodo booleano

3.4.4. Librería de polinomios en varias variables

El código completo se encuentra disponible en <https://github.com/Daniel2000815/multivariate-polynomial>, junto a su documentación, ejemplos de uso y tests usados. La librería consta de tres clases principales, cuya estructura se muestra en la [Figura 3.16](#) y estudiaremos en las siguientes secciones. Todas los métodos cuentan con amplias comprobaciones de seguridad, que en caso de no cumplirse arrojan fallos para que puedan ser manejados. Omitiremos la mayoría de estas comprobaciones en lo siguiente para mayor claridad.

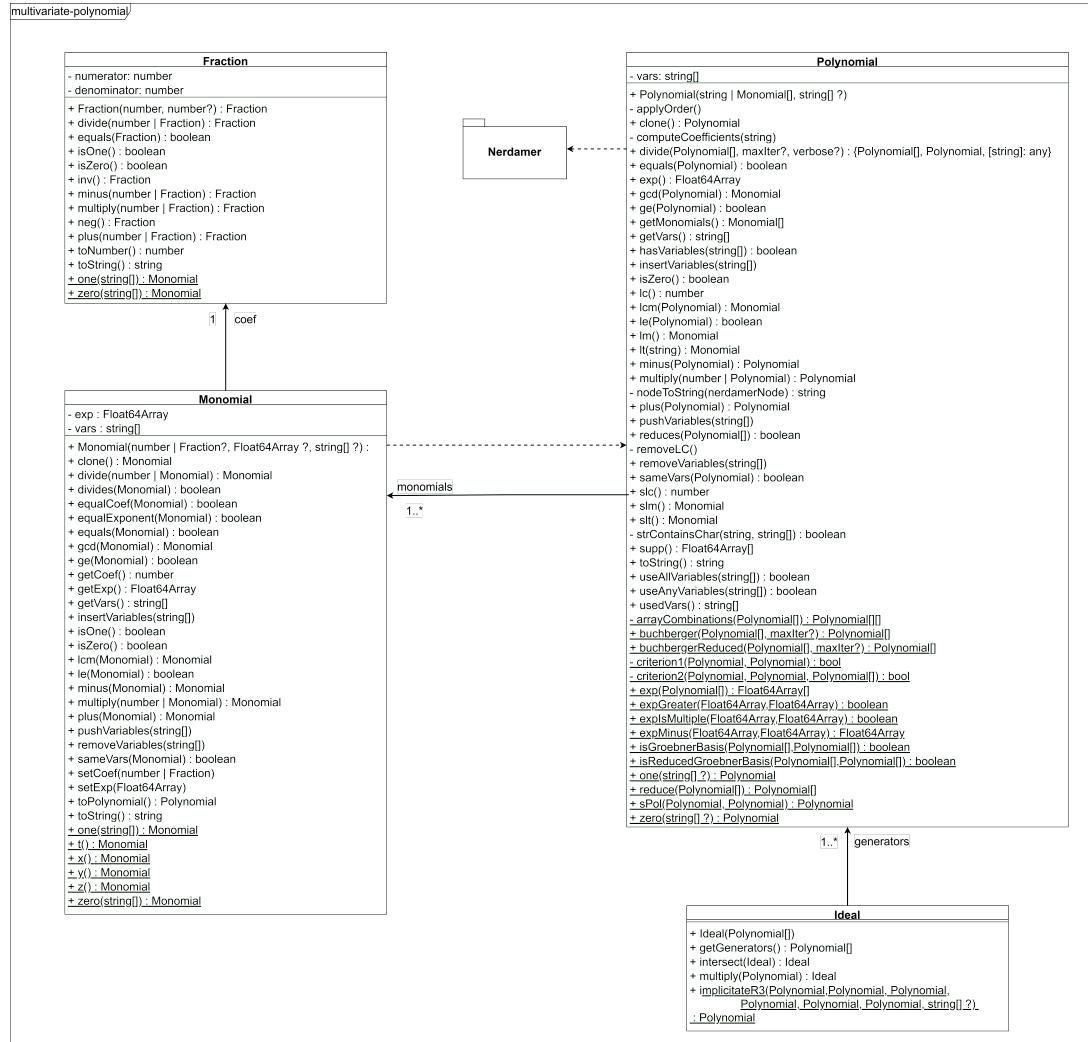


Figura 3.16: Diagrama de clases de la librería `multivariate-polynomial`

3.4.4.1. Clase Monomial

Representa un monomio en varias variables basándose en la [Def. 1.22](#) a través de los siguientes tres atributos.

- **coef:** el coeficiente del monomio. En la primera versión implementada, este atributo estaba representado mediante el tipo primitivo de JavaScript `number`, el cual permite representar flotantes con una precisión similar al tipo `double` de Java o C#. Esta elección hacía que en ocasiones hubiera errores en los cálculos debido a la falta de precisión que este tipo de representaciones acarrean, haciendo que por ejemplo al calcular una base de Gröbner no se pudieran reducir los términos líder correctamente. Por este motivo se decidió trabajar con monomios sobre $A = \mathbb{Q}$, que es lo más común en el contexto de bases de Gröbner. Así, el tipo del coeficiente será `Fraction`. Esta clase representa un número racional mediante un numerador y un denominador, e implementa todos los métodos que cabría esperar en una clase de este tipo.
- **exp:** el exponente del monomio representado como un `Float64Array`, es decir, un array de flotantes de 64 bits. Se utilizó esta representación en lugar de un array convencional del tipo `number[]` debido a que al conocer de antemano el tamaño de los elementos del array (flotantes de 64 bits), tanto la escritura como la lectura son mucho más rápidas. Esto no ocurre con `number[]`, ya que `number` representa cualquier tipo de número.
- **vars:** variables del anillo al que pertenece el monomio representadas como un array de caracteres. El orden utilizado es el lexicográfico, que denotaremos como *lex*, en función del orden que ocupan las variables en el array.

El **constructor** de la clase tiene valores por defecto para cada atributo pudiendo el usuario indicar los que considere necesarios. El valor por defecto de los atributos son los del monomio nulo (`coef=0` y `exp=[0, 0, 0, 0]`) con variables $t \leq x \leq y \leq z$. Para comodidad del usuario, tanto en el constructor como en otros métodos se permite usar el tipo `number` y `number[]`, los cuales serán convertidos internamente al tipo `Fraction` y `Float64Array` respectivamente. Además, se incluye un constructor de copia profunda y métodos estáticos para crear de forma rápida los monomios nulo, unidad, t , x , y y z . Todos los atributos cuentan además con sus respectivos *getters* y *setters*.

```
let m =new Monomial(2, [2,0,1,1],["t", "x", "y", "z"]) // $2t^2yz en Q[t,x,y,z]$
let one =Monomial.one()
let x =Monomial.x()

let coef =x.getCoef() // 1
let exp =m.getExp() // [2,0,1,1]
```

Las operaciones entre monomios son sencillas de implementar, pues solo requieren de manipulaciones básicas de los atributos `coef` y `exp`, junto a una comprobación de la igualdad del atributo `vars` para comprobar que trabajamos con monomios del mismo anillo. Dado que JavaScript no permite la sobrecarga de operadores, la única forma de usar las operaciones es a través de métodos de instancia que devuelvan un nuevo monomio. Se implementan las siguientes operaciones:

- Suma: basándonos en la [Proposición 1.5](#), sumamos los coeficientes tras comprobar que el exponente es el mismo.
- Multiplicación: se permite multiplicar tanto por un monomio como por un número. Si se pasa un número, bastará con multiplicarlo con el coeficiente del monomio. Si

en su lugar se pasa un monomio, de nuevo por la [Proposición 1.5](#), se multiplican los coeficientes y se suman los exponentes componente a componente.

- Resta: se implementa como una suma junto a una multiplicación por -1 al segundo monomio.
- División: similar a la multiplicación, pero en su lugar los coeficientes se dividen y las componentes de los exponentes se restan. Antes de llevarla a cabo se debe comprobar que el monomio por el que se va a dividir es no nulo. En particular, es posible obtener un monomio con alguna componente del exponente negativa, de forma diferente a lo que indica la [Def. 1.22](#). Se tomó esta decisión para que la librería fuera útil también en otro tipo de escenarios, pues un gran número de usuarios podrían esperar la presencia de esta funcionalidad.
- Igualdad: comprueba si tanto el coeficiente como el exponente de dos monomios coinciden.
- Orden: el método `le` implementa el orden lexicográfico ([Def. 1.26](#)) para comparar dos monomios. Se recorren simultáneamente las componentes de los exponentes de cada monomio hasta encontrar una posición en la que no coincidan. Se devolverá `true` si la entrada del monomio actual es menor a la del otro, y `false` en otro caso. Si ambos monomios tengan el mismo exponente se devuelve `true`. También existe el método `ge`, similar al anterior pero tomando como criterio si la entrada del exponente del monomio actual es mayor a la del otro.
- Mínimo común múltiplo: dado que estamos en \mathbb{Q} , el coeficiente del mínimo común múltiplo ([Def. 1.41](#)) siempre será uno. En cuanto al exponente, basta con tomar el máximo de los exponentes de cada monomio en cada componente.
- Máximo común divisor: igual que el mínimo común múltiplo pero tomando el mínimo de las componentes de los exponentes ([Def. 1.41](#)).

```
var resta = m.minus(one)           // 2t^2yz - 1
var mult = m.multiply(x)          // 2t^2xyz
var div  = m.divide(Monomial.t()) // 2tyz
var lcm  = m.lcm(x)              // t^2xyz
```

Que el usuario obtenga una representación clara y concisa del monomio será esencial para su experiencia con la librería. Esto también resulta útil a lo largo del desarrollo de las diferentes clases, pues proporciona una forma de consultar la información mucho más cómoda. Por ello incluimos el método `toString()`, el cual devuelve una representación como **cadena de texto** de la instancia de la clase usando *lex*. En su implementación se tienen en cuenta varios aspectos que mejoran la legibilidad, como que si el coeficiente es positivo su signo se sobreentiende, y si es uno o menos este puede ser omitido. En cuanto a las variables, comprobaremos si el valor de cada una en el exponente es cero, en cuyo caso no debe ser mostrada, o si es uno y no es necesario indicar la potencia.

Otros métodos de interés son los que permiten modificar las variables del anillo del monomio. Se permitirá insertar o eliminar una lista de variables en cualquier posición. Este

cambio tendrá una gran repercusión, pues recordemos que internamente se usa *lex* con el orden de las variables según su posición en el array. También existen funciones para realizar diferentes comprobaciones sobre uno o varios monomios. Estas son usadas internamente para realizar comprobaciones previas antes de realizar determinadas operaciones, pero también están a disposición del usuario. Ejemplos de estas comprobaciones son si un monomio es nulo, si uno divide a otro, dos tienen el mismo exponente, mismas variables, etc. Por último, existe la opción de obtener una instancia de un polinomio formado por el monomio en cuestión. Este será de tipo `Polynomial`, clase que estudiamos en la siguiente sección.

```

let s =new Monomial(2, [1,0,0,0,0],["s","t", "x", "y", "z"])
let m =new Monomial(2, [2,0,1,1],["t", "x", "y", "z"])

m.toString()           // 2t^2yz
m.isZero()            // false
m.divides(x)          // false
m = m.multiply(s)     // ERROR: MONOMIALS IN DIFFERENT RINGS

// Insertar las variables "w" al final y "s,u" en la posicion 2
m.pushVariables(["w"]) // x.getVars() -> ["t","x","y","z","w"]
m.insertVariables(["s","u"], 2) // x.getVars() -> ["t","x","s","u","y","z","w"]

m = m.multiply(s)
m.toString()           // 2t^2syz

```

3.4.4.2. Clase `Polynomial`

Siguiendo la Def. 1.23, podemos representar un polinomio como una colección de monomios, que representaremos mediante el atributo `monomials` del tipo `Monomial[]`. La posición de los monomios en el array será en todo momento siguiendo el orden *lex* para así simplificar la implementación de los métodos de instancia. Al igual que con los monomios, tendremos un atributo `vars` indicando las variables del anillo al que pertenece el polinomio, que será el mismo que el de los monomios que lo conforman. Así, el constructor de la clase `Polynomial` recibirá una lista con las variables del anillo y otra con los monomios. Se comprobará que todos los monomios pertenezcan al mismo anillo, en cuyo caso serán ordenados usando *lex* y el atributo `vars` tomará el valor del atributo homónimo de cualquier monomio pasado como argumento. De no pasarse ningún monomio, el polinomio será inicializado con el monomio nulo y las variables $t \leq x \leq y \leq z$.

Además de inicializar un polinomio mediante una colección de monomios creados individualmente, se permite al usuario pasar una cadena de texto del polinomio que quiere crear. Dado que internamente necesitamos trabajar con una lista de monomios, tendremos que extraer los monomios de la expresión. Para ello empezaremos parseando el string a un formato estructurado usando Nerdamer. Esta librería nos permite trabajar con los polinomios de manera básica, pero operaciones más avanzadas como el algoritmo de división no pueden ser implementadas, entre otros motivos porque Nerdamer no es compatible con el uso de órdenes monomiales. Nerdamer simplifica por defecto la expresión del polinomio que le pasemos, lo cual nos interesa, pero lo devuelve como producto de sus raíces. Podemos cambiar

este comportamiento y hacer que nos lo muestre como una suma de monomios través del método `expand()`. Esto nos será necesario para continuar con el procedimiento de conversión del polinomio a nuestro formato. Tras esto ya podemos obtener la representación en árbol con notación polaca inversa (RPN) de la expresión parseada usando el método `tree()`.

Una vez obtenida la estructura de árbol es sencillo localizar los monomios, pues estos solo pueden estar separados por un símbolo de adición o sustracción. Así, recorriendo el árbol en preorden, cuando encontremos un nodo que no tenga ninguno de estos símbolos como descendiente, el subárbol del que es nodo raíz representará un monomio. Una vez localizado el nodo, podemos obtener la expresión del monomio recorriendo el subárbol de nuevo en preorden.

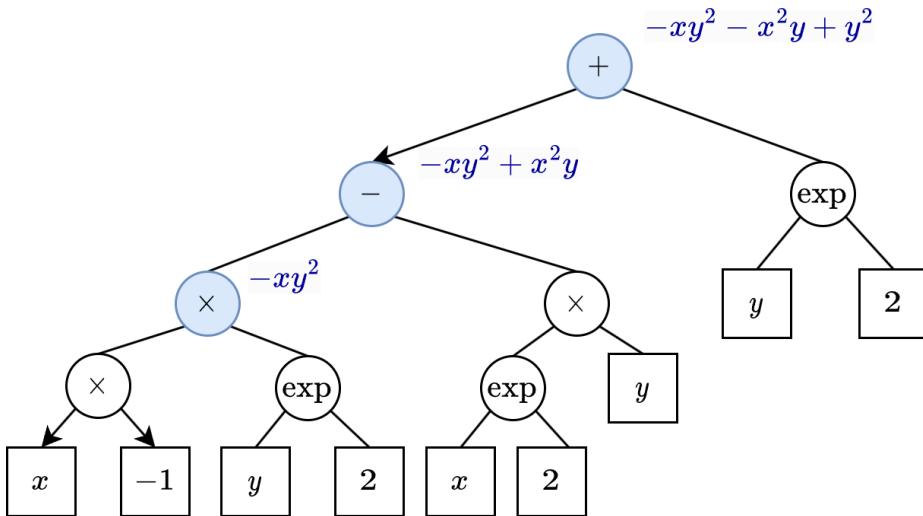


Figura 3.17: Ejemplo de búsqueda de monomios en el árbol

Dado que las variables del anillo siempre son pasadas al constructor del polinomio, solo necesitamos extraer el coeficiente y exponente de la expresión de cada monomio encontrado en el árbol. Como la expresión resultado de expandir cada nodo de un monomio siempre tendrá la forma `<coeficiente><variables>`, podemos simplemente recorrer la expresión hasta detectar un carácter no numérico, el cual marcará la división entre el coeficiente y las variables. Con esta información ya podemos crear la instancia de cada monomio que conforma al polinomio, pudiendo calcular su exponente de forma sencilla usando el método `deg` de Nerdamer, que nos proporciona el grado de la variable recibida como argumento de la expresión. A lo largo de este proceso tendremos que haber tenido cuidado de realizar varias comprobaciones de errores, como que se detecte alguna variable que no pertenece al anillo del polinomio, o casos especiales, como que el coeficiente sea uno o menos uno y no aparezca en la expresión. Como último paso deberemos aplicar el orden *lex* a los monomios encontrados. La manera más sencilla es a través del método `sort` de JavaScript, el cual recibe un comparador binario como argumento. En nuestro caso este comparador será el de menor o igual ya implementado en la clase `Monomial`. Observamos que no puede ocurrir que dos monomios tengan el mismo exponente, ya que si no Nerdamer los habría simplificado en un

único monomio.

La implementación de la mayoría de las **operaciones básicas** es sencilla una vez se ha conseguido la representación como lista de monomios, pues muchas se basan en las operaciones ya implementadas para monomios.

- Suma: aplicando la [Proposición 1.5](#), para cada monomio se comprueban los siguientes de la lista cuyo exponente es el mismo mientras estos se van sumando y añadiendo a una lista auxiliar para asegurarnos de que no vuelvan a ser tenidos en cuenta. Si el monomio suma no es cero, se añade al polinomio resultado. Al igual que para los polinomios, se implementa la operación de resta a través de esta junto con un producto.
- Multiplicación: como en el caso de los monomios, este operador admite como parámetro tanto un `number` como una instancia de `Polynomial`. En el caso de que el argumento sea un escalar, solo hay que multiplicar cada monomio por él, teniendo en cuenta que si es cero podemos devolver directamente el polinomio nulo. En cuanto a la multiplicación por otro polinomio, aplicando otra vez la [Proposición 1.5](#), hay que realizar un doble bucle multiplicando cada monomio de un polinomio por todos los del otro. En este proceso pueden generarse varios monomios con el mismo exponente aunque esto no ocurriese con los polinomios originales, como sería el caso de la multiplicación

$$(xy - ty) \cdot (t + x) = txy + x^2y - t^2y - txy = x^2y - t^2y.$$

Los monomios con mismo exponente serán por tanto sumados y añadidos al resultado si son distintos de cero.

- División: implementa el [Algoritmo 1](#) haciendo uso de las operaciones de monomios y polinomios ya presentadas. Dado que esta operación puede llevar una cantidad considerable de tiempo, se puede indicar el máximo número de iteraciones a realizar, entendiendo como iteración el reducir el término líder del polinomio actual. Además, existe la posibilidad de obtener como resultado una lista con los pasos detallados seguidos para obtener el resultado mediante el parámetro `verbose`, aunque esta funcionalidad no está presente en la versión usada en la aplicación por motivos de eficiencia.
- Igualdad: comprueba si cada monomio de un polinomio está también en el otro. Dado que las entradas de `monomials` están ordenadas con `lex` basta comparar cada componente del array.
- Orden: permite comparar dos polinomios usando la [Def. 1.26](#) del orden `lex` aplicando los métodos `le` y `ge` de la clase `Monomial` sobre los monomios líder de ambos polinomios.
- S-polinomio: implementado siguiendo la [Def. 1.42](#) usando las operaciones resta y producto de polinomios y resta de exponentes (esta última no disponible para el usuario) tras comprobar que ambos polinomios pertenecen al mismo ideal. Es un método estático.
- Mínimo común múltiplo y máximo común divisor: usa los métodos de la clase `Monomial` sobre los exponentes de ambos polinomios.

Como no podía ser de otra forma, se incluyen getters para los atributos de la clase, así como métodos que nos permitan obtener información básica del polinomio de forma rápida,

como `lm()` para el monomio líder, `sm()` para el segundo monomio líder, `supp()` para obtener el soporte del polinomio, etc. Gracias a la representación que hemos elegido, estos se basarán en aplicar consultas muy sencillas sobre los atributos de la clase. Otros métodos muy útiles son los que comprueban propiedades del polinomio, como si este reduce a cero sobre un conjunto de polinomios F con `reduce(F)`.

Esta clase también implementa su **representación como cadena de texto** basándose en la de la clase `Monomial`. En este caso se tienen en cuenta algunas consideraciones adicionales. Dado que cuando el coeficiente de un monomio era positivo no incluíamos el símbolo, no podemos simplemente concatenar las cadenas de texto de los monomios del polinomio, sino que habrá que comprobar para cada uno si debemos añadir nosotros el símbolo `+` como separador, a excepción de para el primero. También deberemos asegurarnos que el espaciado entre los monomios es consistente y se deja un espacio entre su símbolo y el resto de la expresión.

Los métodos más importantes de esta clase son los relacionados con las **bases de Gröbner**, los cuales son estáticos. Se implementa el [Algoritmo 2](#) en el método `buchberger`, el cual recibe como argumento los generadores del ideal y opcionalmente un máximo de iteraciones. Este método hace un uso intensivo de los operadores de división y `S-polimio`, así como la función `arrayCombinations` no accesible al usuario, que obtiene las parejas de polinomios que comparar siguiendo la estrategia normal. También hace uso de los criterios de Buchberger del [Teorema 1.6](#). Sus implementaciones son un buen ejemplo de como usar esta biblioteca de forma más avanzada, y a continuación mostramos la del segundo criterio.

```

private static criterion2(f: Polynomial, g: Polynomial, G: Polynomial[]): boolean {
    let res = false;
    const startIndex = Math.max(G.indexOf(f), G.indexOf(g)) + 1;

    for(let i=startIndex; i<G.length !res; i++){
        const h = G[i];
        const sPolfh = Polynomial.sPol(f, h);
        const sPolgh = Polynomial.sPol(g, h);

        if(!h.lm().divides(f.lcm(g))) continue;
        if(sPolfh.reduces(G) || sPolgh.reduces(G)) return true
        else{
            const lmH = h.lm();
            const gcdFG = f.gcd(g);

            const cond1 = lmH.divides(f.lm().divide(gcdFG)) !
                g.sm().multiply(h.lm()).equals(h.sm().multiply(g.lm()));
            const cond2 = lmH.divides(g.lm().divide(gcdFG)) !
                f.sm().multiply(h.lm()).equals(h.sm().multiply(f.lm()));

            res = cond1 || cond2;
        }
    }
    return res;
}

```

Para reducir la base obtenida se puede usar la función `reduce`, que asume que recibe una base de Gröbner y la reduce a través del [Algoritmo 3](#).

```

static reduce(G: Polynomial[]) :Polynomial[]{
    let res :Polynomial[] =[];
    G = G.map(g =>g.multiply(1/g.lc())); // hacemos cada polinomio mnico (lc=1)

    for(let i=0; i<G.length; i++){
        let g =G[i];
        let div =g.divide(G.filter(e =>!e.equals(g))); // devuelve el resto y los
                                                       coeficientes

        // si el resto no es cero, sustituimos el polinomio por el y lo añadimos a la
        // base resultado
        if(!div.remainder.isZero()){
            G[i] =div.remainder;
            res.push(div.remainder);
        }
    }
    return res;
}

```

Para obtener una base reducida directamente se puede usar `buchbergerReduced`, que ejecuta consecutivamente los métodos `buchberger` y `reduce`.

```

static buchbergerReduced(F: Polynomial[], maxIter: number =1000) {
    return this.reduce(this.buchberger(F, maxIter));
}

```

Por último, al igual que para la clase `Monomial`, se incluyen los métodos `insertVariables`, `pushVariables` y `removeVariables` para añadir o eliminar variables del cuerpo del polinomio. Estos tendrán todavía más trascendencia que en la clase `Monomial`, pues modificar el orden de las variables cambia por completo las propiedades del polinomio, tales como su monomio líder, y por tanto el resultado obtenido al usarlo en el algoritmo de división. Además, dado que para implementar el algoritmo de implicitación deberemos ser capaces de comprobar si un polinomio contiene ciertas variables, se ha implementado el método `useAnyVariables`, que recibe una lista de variables y comprueba si la entrada correspondiente en el exponente del polinomio de alguna de ellas es distinta de cero. También se puede usar `useAllVariables` para comprobar si se usan todas.

3.4.4.3. Clase Ideal

Esta clase consta de un único atributo, `gens`, representando una lista de los polinomios generadores del ideal. Los generadores son pasados como argumento al constructor, pero antes de asignar su valor a `gens` se calcula una base de Gröbner reducida suya para trabajar con el menor número posible de polinomios. Se incluyen los operadores de producto e intersección de ideales, que son muy sencillos de implementar usando los generadores. Sin embargo, el objetivo principal de esta librería era el de resolver el problema de implicitación de una parametrización racional. Usando bases de Gröbner habíamos llegado a la solución

descrita [Teorema 1.9](#), y que implementamos en el método estático `implicitateR3` de esta clase. Como su nombre indica, se ha implementado para el caso $A^n = \mathbb{R}^3$ por motivos de eficiencia de la aplicación. Supondremos por tanto que la parametrización es de la forma

$$x = \frac{f_1(t_1, \dots, t_r)}{q_1(t_1, \dots, t_r)}, \quad y = \frac{f_2(t_1, \dots, t_r)}{q_2(t_1, \dots, t_r)}, \quad z = \frac{f_3(t_1, \dots, t_r)}{q_3(t_1, \dots, t_r)},$$

donde $f_1, f_2, f_3, q_1, q_2, q_3 \in \mathbb{R}[t_1, \dots, t_r]$ y $q_1, q_2, q_3 \neq 0$. A continuación estudiamos en detalle la implementación del algoritmo.

La función recibe como argumentos las instancias de los polinomios $f_1, f_2, f_3, q_1, q_2, q_3$, comprobando que todos pertenecen al mismo cuerpo. En el caso de nuestra aplicación, esta creará estos polinomios según los datos que haya introducido el usuario, forzando a que este use las variables s y t para la parametrización. Sin embargo, la librería admite que se use cualquier combinación de variables a excepción de x, y, z , que están reservadas para las variables del ideal de eliminación del teorema. Si se detecta su uso, se generará una excepción. Por último, dado que en la aplicación el usuario es capaz de usar parámetros para modificar la superficie en el editor de nodos, estos deberán ser tratados como variables más del ideal de eliminación final, y se pueden pasar como parámetro opcional. Una vez identificadas las variables que ha usado el usuario, debemos buscar una más que no esté siendo usada para que asuma la función de la variable auxiliar y del enunciado del teorema. Para ello basta con recorrer el alfabeto hasta encontrar un nombre libre para esta variable.

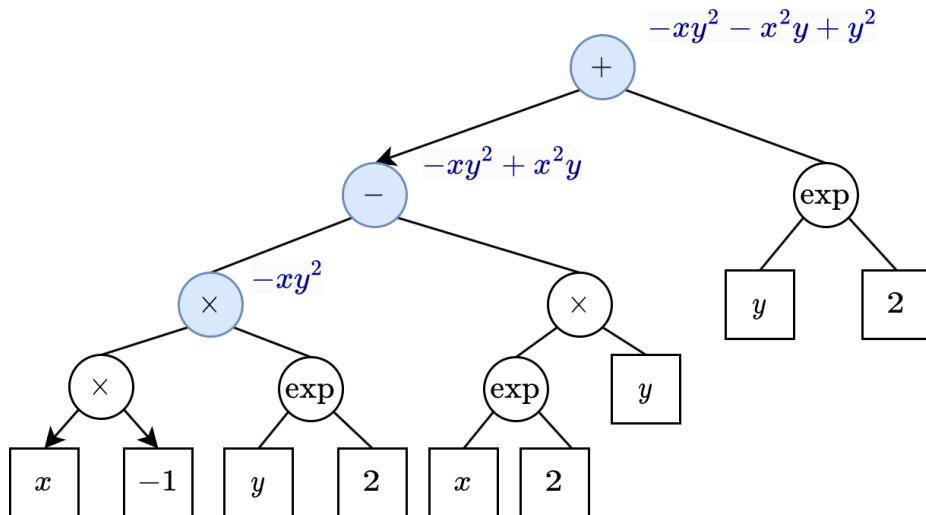


Figura 3.18: Ejemplo de búsqueda de monomios en el árbol

Las comprobaciones anteriores nos permiten obtener los siguientes arrays de variables:

- `elimVars`: son las variables t_1, \dots, t_r, y del teorema, y no formarán parte del ideal resultado. En nuestro caso, serán las variables de los polinomios $f_1, f_2, f_3, q_1, q_2, q_3$ junto a la variable auxiliar obtenida previamente.
- `resVars`: variables x_1, \dots, x_n del teorema. En nuestro caso serán x, y, z y los parámetros

usados por el usuario en la parametrización.

- `impVars`: variables del ideal I del teorema, y por tanto las del cuerpo en el que trabajaremos durante el resto del algoritmo. Será la unión de los arrays anteriores. Dado que los polinomios $f_1, f_2, f_3, q_1, q_2, q_3$ pasados como argumento solo usan las variables en `elimVars`, el resto deberán ser añadidas con `pushVariables`.

Por ejemplo, en el caso de la parametrización del Ej. 1.3, tendríamos

$$\begin{aligned} f_1(s, t) &= s + t, & f_2(s, t) &= s, & f_3(s, t) &= t, \\ q_1(s, t) &= 1, & q_2(s, t) &= 1, & q_3(s, t) &= 1, \\ \text{elimVars} &= [s, t], & \text{resVars} &= [x, y, z], & \text{elimVars} &= [s, t, x, y, z]. \end{aligned}$$

Para obtener los generadores del ideal I , primero creamos polinomios para cada una de las variables x, y, z , además de la auxiliar. Estos pueden ser creados tanto pasando la cadena de texto con la variable como creando el exponente a mano. Por motivos de legibilidad, en este ejemplo elegimos la primera opción, aunque la segunda es más eficiente. Mediante operaciones de resta y producto obtenemos los generadores finales del ideal

$$I = \langle q_1x - f_1, q_2y - f_2, q_3z - f_3, 1 - q_1q_2q_3v_{aux} \rangle,$$

que son usados para crear una instancia de la clase `Ideal`. Una vez construido este ideal I , para obtener el ideal de eliminación J basta con tomar los generadores de I que no usen ninguna de las variables en `elimVars` mediante el método `useAnyVariables`. Si no se encuentra ningún generador, se devuelve el polinomio nulo, y si se obtiene más de uno se crea una excepción indicando que la parametrización no puede ser representada por una única ecuación implícita. En el mejor de los casos se obtendrá un único generador del ideal de eliminación, aunque este seguirá usando las variables de `elimVars`, lo cual no es deseable ni lo que el usuario espera, así que antes de devolverlo se eliminan esas variables del anillo del polinomio con `removeVariables`.

El código final del algoritmo es el siguiente.

Listing 3.5 Algoritmo de implicitación

```
static implicitateR3(fx: Polynomial, fy: Polynomial, fz: Polynomial, qx: Polynomial,
                      qy: Polynomial, qz: Polynomial, parameters: string[] = []){
    if(!fx.sameVars(fy) || !fx.sameVars(fz) || !fx.sameVars(qx) ||
       !fx.sameVars(qy) || !fx.sameVars(qz))
        throw new Error("PARAMETRIZATIONS IN DIFFERENT RINGS");
    if (qx.isZero() || qy.isZero() || qz.isZero())
        throw new Error("DENOMINATORS CAN'T BE 0");
    // Variables que ha usado el usuario en la parametrización distintas a x,y,z
    var elimVars = fx.getVars().filter(v =>!parameters.includes(v));
    if(elimVars.some(v =>["x", "y", "z"].includes(v)))
        throw new Error("PARAMETRIZATIONS CAN'T USE X,Y,Z VARIABLES");
    // Buscamos variable auxiliar libre que despues sera eliminada
```

```

const allVars = elimVars.join('');
var variableAuxiliar ='a';
for (var letter of 'abcdefghijklmnopqrstuvwxyz') {
    if (!allVars.includes(letter)) {
        variableAuxiliar =letter;
        break;
    }
    throw new Error("TOO MANY VARIABLES")
}
var posVarAux =elimVars.push(variableAuxiliar);

// Variables del ideal J del teorema, parametros incluidos
var resVars =[ "x", "y", "z"].concat(parameters);

// Variables del ideal I del teorema
const impVars =elimVars.concat(resVars);

// Construimos generadores de I
const x =new Polynomial("x", impVars);
const y =new Polynomial("y", impVars);
const z =new Polynomial("z", impVars);
const varAuxPol =new Polynomial(variableAuxiliar, impVars);

const variablesToAdd =[variableAuxiliar].concat(resVars);
fx.insertVariables(variablesToAdd, 2); qx.insertVariables(variablesToAdd, 2);
fy.insertVariables(variablesToAdd, 2); qy.insertVariables(variablesToAdd, 2);
fz.insertVariables(variablesToAdd, 2); qz.insertVariables(variablesToAdd, 2);

const gen1 =qx.multiply(x).minus(fx);
const gen2 =qy.multiply(y).minus(fy);
const gen3 =qz.multiply(z).minus(fz);
const prod =Polynomial.one(impVars).minus(
    qx.multiply(qy).multiply(qz).multiply(varAuxPol)
);
const I =new Ideal([gen1, gen2, gen3, prod].concat());
}

// Los generadores de J son los de I que contengan solo las variables de resVars
var J :Polynomial[] =[];
I.getGenerators().forEach(gen =>{
    if(!gen.useAnyVariables(elimVars)){
        J.push(gen);
    }
})

// Comprobar resultado
if(J.length ==0)
    return Polynomial.zero(resVars);
else if(J.length ==1){
    var intersection =J[0];
    intersection.removeVariables(elimVars);
    return intersection;
}

```

```

    else
        throw new Error("PARAMETRIZATION DOES NOT SATISFY AN UNIQUE IMPLICIT EQUATION");
}

```

3.4.5. Panel de primitivas

Este componente empezará leyendo la información presente en el contenedor de primitivas de Zustand para presentar una tabla que muestra la información más relevante de las primitivas presentes en el sistema, incluyendo en cada fila botones para editar o eliminar cada primitiva. En caso de que se desee eliminar alguna, bastará con llamar a la función

		NAME	TYPE	PARAMETERS	INPUT
		Sphere	Implicit	r	$x^2 + y^2 + z^2 - r$
		Sphere Parametric	Parametric		$x = \frac{2 \cdot s}{s^2 + t^2 + 1}, y = \frac{2 \cdot t}{s^2 + t^2 + 1}, z = \frac{s^2 + t^2 - 1}{s^2 + t^2 + 1}$
		Plane	Parametric		$s + t, s, t$
		Hyperbolic Paraboloid	Parametric		$t, s, t^2 - s^2$
		Torus	SDF	R, r	$\text{length}(\text{vec2}(\text{length}(p.xz)-R,p.y)) - r$
		Cube	SDF	l	$\text{length}(\max(\text{abs}(p) - \text{vec3}(l),0.0)) + \min(\max(\text{abs}(p.x) - l,\max(\text{abs}(p.y) - l,\text{abs}(p.z) - l)),0.0)$
		Ellipsoid	Parametric		$s, t, s^2 + t^2$

Figura 3.19: Tabla de primitivas

correspondiente del contenedor. Si en cambio se desea realizar una modificación, se abrirá un cuadro de diálogo con toda la información guardada de la primitiva, incluyendo los parámetros y ecuaciones que introdujo el usuario en el momento de la creación y el método de definición de la superficie (a través de una ecuación implícita, parametrización o SDF). Toda esta información se encuentra convenientemente almacenada en el contenedor.

Como sabemos, el componente Lienzo para que el usuario pueda previsualizar la superficie que está definiendo puede recibir como argumento un manejador de excepción a través del prop `onError` para actuar en caso de fallo de compilación. Nosotros lo usaremos para que, en caso de error, el manejador obtenga el mensaje de error y se muestre al usuario en el campo correspondiente.

La forma de obtener la función distancia dependerá del método de entrada elegido por el usuario. La manera más directa es a través del modo **SDF**, ya que el usuario introduce la función distancia con signo con sintaxis GLSL y puede ser pasada directamente al lienzo. El manejo de errores es también muy sencillo, pues `gl-react` nos pasa el mensaje a través del centinela y nosotros solo tenemos que mostrárselo al usuario.

3.4 Implementación de la aplicación web

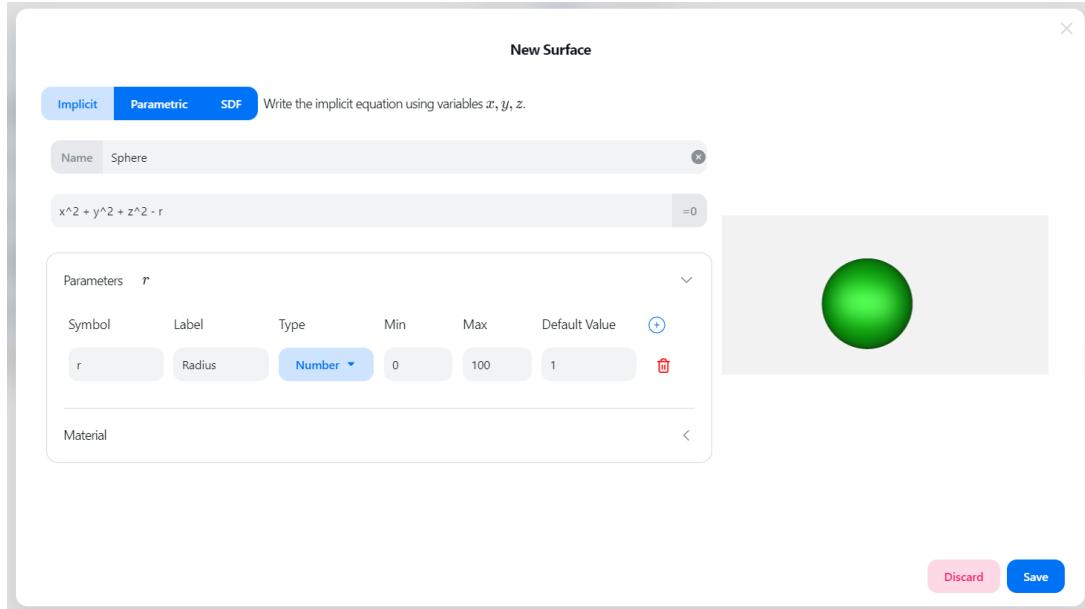


Figura 3.20: Diálogo de edición de primitivas

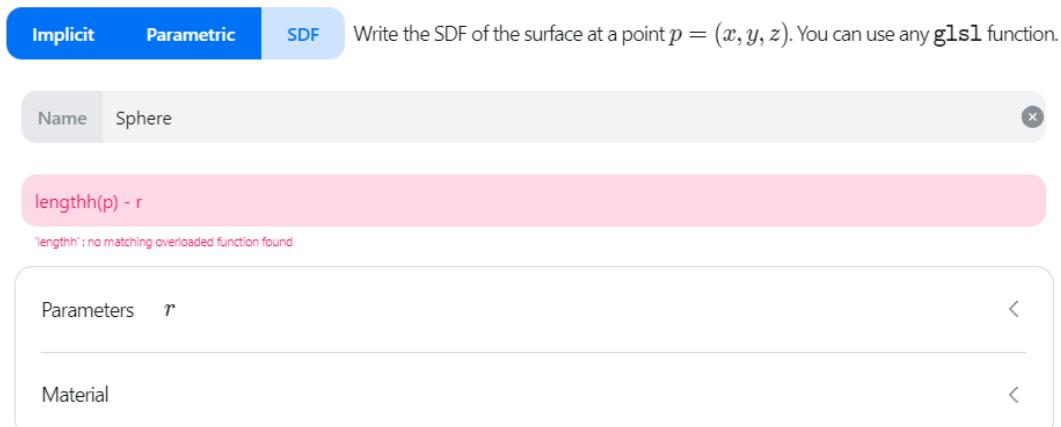


Figura 3.21: Ejemplo de error al usar un parámetro no definido

La siguiente forma de definir una superficie es a través de su **ecuación implícita**, en cuyo caso lo que recibirá el componente del lienzo será la cota función distancia con signo obtenida en la [Proposición 1.4](#). Para obtenerla volveremos a hacer uso de Nerdamer, pues nos permitirá calcular el gradiente y su norma de forma más directa. La función que calcula esta cota es la siguiente.

Listing 3.6 Obtención de cota de ecuación implícita

```
function ImplicitToSDF(implicit: string, parameters: Parameter[], evaluate: boolean = false) :string{
```

```

let f: string |null =null; // Parsed string by nerdamer
let res ="";

try {
    f = nerdamer(implicit).toString();
} catch (error: any) {
    error =error.message.split("at ")[0];
    throw new Error(`ERROR PARSING EQUATION ${implicit}`);
}

// Calculamos norma
const dfdx =nerdamer.diff(f, "x", 1);
const dfdy =nerdamer.diff(f, "y", 1);
const dfdz =nerdamer.diff(f, "z", 1);
const norm =nerdamer(`sqrt((${dfdx})^2 + (${dfdy})^2 + (${dfdz})^2)`);

// Comprobamos que podemos dividir por la norma
if (norm.toString() ==="0") {
    throw new Error("NORM CAN'T BE 0");
}
else{
    let sdf =nerdamer(`(${f})/(${norm})`);
    var x =nerdamerTS.tree(sdf.toString());
    res =StringToSDF(x, parameters, evaluate);
}

return res;
}

```

El resultado que nos proporciona Nerdamer no puede ser pasado directamente al lienzo, ya que no está en sintaxis GLSL. Tampoco podemos pasar la cadena de texto que nos proporciona su método `toString()`, ya que funciones como las potencias vendrían escritas como a^b , lo que en GLSL es el OR exclusivo bit a bit. El objetivo de la función `StringToSDF` es el de tener en cuenta estos casos especiales para así devolver una sentencia GLSL válida. Para ello recibe el árbol de la expresión de Nerdamer, y de forma similar a como buscábamos monomios en el constructor de la clase `Polynomial`, ahora buscaremos los operadores que nos interese sustituir. Por ejemplo, en el caso de la potencia deberemos buscar el nodo con el operador `^` y escribir `pow(a, b)`, donde `a` y `b` son el resultado de expandir en preorden los hijos a izquierda y derecha respectivamente de dicho nodo. Un factor a tener en cuenta en este proceso es que, dado que el usuario puede nombrar los parámetros como quiera (incluso con nombres de más de un carácter), tenemos que pasar a Nerdamer el polinomio que queremos que parsee en un formato con el que entienda cuales son las diferentes variables. Un ejemplo de por qué esto es necesario es el siguiente. Imaginemos que tenemos un parámetro `r` y el polinomio $rx - y$. Si usamos el método `getVariables()` de Nerdamer veremos que nos devuelve la lista `[rx, y]`, cuando para nosotros es obvio que debería ser `[r, x, y]`. Para arreglar esto basta con no omitir el símbolo del producto en la representación como cadena de texto del polinomio, lo cual hemos implementado de antemano en los métodos `toString(showProductChar = false)` de las clases `Monomial` y `Polynomial` a través de un parámetro opcional. Así, el polinomio que recibiría Nerdamer sería $r \cdot x + y$, y no habría lugar a confusión a la hora de detectar

las variables. Por último, dado que en este componente el usuario no introduce valor para los parámetros, la función `StringToSDF` incluye la posibilidad de evaluar los parámetros en su valor por defecto para así poder pasar esta expresión al lienzo de previsualización. En el caso del editor de nodos esto no será necesario, pues el usuario indicará el valor de los parámetros.

La última forma de definir una superficie era a través de su **parametrización racional**. Dado que ya tenemos el método `implicitateR3` de la clase `Ideal`, que nos proporciona una ecuación implícita de la superficie, y acabamos de ver como obtener una SDF a partir de ella, basta con aplicar ambos métodos consecutivamente. En caso de que la parametrización no venga representada por una sola ecuación implícita, el método `implicitateR3` devolverá un error y su mensaje será mostrado al usuario.

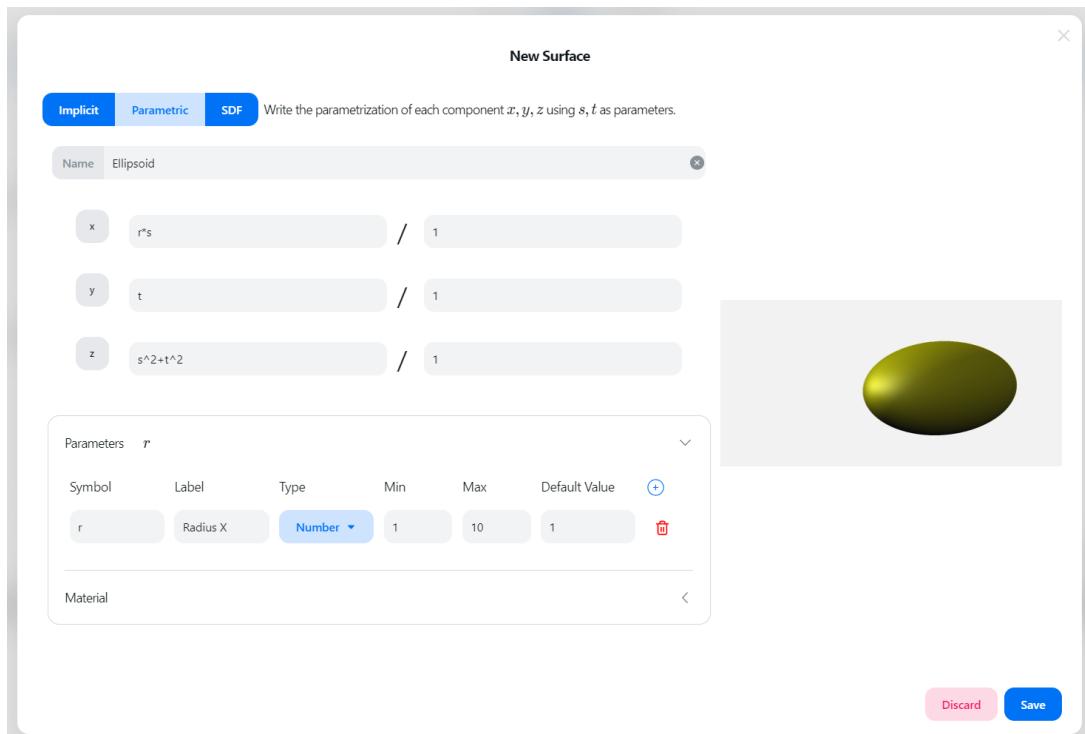


Figura 3.22: Ejemplo de creación de un elipsoide definido paramétricamente

En todo momento se comprueba que el usuario esté introduciendo información válida, capturando posibles errores de `Nerdamer`, `multivariate-polynomial` y `gl-react`. Se ha puesto especial atención en este aspecto durante el desarrollo de `multivariate-polynomial`, que realiza innumerables comprobaciones de seguridad y devuelvo un error lo suficientemente descriptivo en cada caso como para que el usuario pueda actuar correctamente. Ya vimos un ejemplo de estas comprobaciones al final de la Subsubsección 3.4.4.1, en la que al intentar multiplicar dos monomios de distintos anillos se indican las variables que difieren. En caso de que se detecte algún error, se le indica al usuario en el campo correspondiente el mensaje. Para ello, toda entrada de texto se realiza a través de un componente personalizado que puede recibir como prop un mensaje de error de manera opcional, y en caso de que esto ocu-

rra cambia su color y muestra el mensaje en su parte inferior. Un ejemplo de comprobación adicional a las que se realizan al calcular la función distancia es la siguiente. Dado que el identificador se calcula a partir del nombre que introduzca el usuario, no se podrá elegir ningún nombre cuyo identificador asociado ya esté asignado a otra primitiva diferente.

4 Pruebas y rendimiento

En los siguientes apartados vamos a mostrar el procedimiento seguido para validar y medir las capacidades de dos de las piezas más importantes de la aplicación: el componente del lienzo y la librería `multivariate-polynomial`. Estas comprobaciones han sido una fase esencial en la evolución de la aplicación. El lienzo debe de presentar un rendimiento lo suficientemente bueno en todos los dispositivos para una buena experiencia de usuario. Esto es cierto también en el caso de la librería de polinomios, pero en su caso habrá además que comprobar que todos sus métodos y algoritmos producen resultados correctos.

4.1. Lienzo

La sección de la aplicación `Playground` mencionada en la [Subsección 3.4.2](#) fue ideada originalmente como un escenario de pruebas de rendimiento que no estaría en el entorno de producción, pero eventualmente su estado avanzó hasta tomar la forma actual. En su primera etapa de desarrollo, este componente solo contaba con un lienzo dibujando una escena de prueba similar a la de la [Figura 2.33](#) con una cámara orbitando de forma continua, un medidor de su resolución, y unos interruptores con los que elegir qué algoritmos de visualización usar. Actualmente esta sección cuenta además con controles para los materiales de las primitivas y las propiedades de las luces, incluyendo color, tamaño e intensidad, y está totalmente disponible su uso al usuario. Junto a la información del rendimiento de las herramientas de desarrollador de Google Chrome, que permiten medir los FPS (fotogramas por segundo) de la aplicación, tenemos información suficiente para evaluar el rendimiento del lienzo.

Se han hecho pruebas de rendimiento en dos equipos diferentes, uno de sobremesa y un portátil, cuyas características se muestran en la [Figura 4.1](#). En ambos equipos se han probado todas las combinaciones de activación de los algoritmos de visualización de sombras y oclusión ambiental junto a valores del factor de supermuestreo *AA* del *antialiasing* entre uno (no aplicar *antialiasing*) y cuatro (resolución interna dieciséis veces mayor a la mostrada), todo esto en dos configuraciones de resolución para el lienzo. En las gráficas que iremos mostrando a continuación se representan en el eje vertical los fotogramas por segundo medidos, y en el horizontal el factor de escalado del *antialiasing*, pues veremos que será este factor el que más afecte al rendimiento.

Empecemos estudiando el ordenador de sobremesa en su configuración de resolución más alta para el lienzo ([Figura 4.3](#)). En su versión más básica (sin sombras, oclusión ambiental o *antialiasing*) se obtiene un rendimiento muy alto, tanto que está limitado por la frecuencia de refresco de la pantalla, en este caso 144 Hz. Como habíamos adelantado, el algoritmo más demandante es el de *antialiasing*, y cada vez que se aumenta su factor de escalado el rendimiento se reduce aproximadamente a la mitad para cualquier otra combinación de valores. Todo lo contrario ocurre con el algoritmo de oclusión ambiental, cuyo efecto en el rendimiento es prácticamente anecdótico, incluso en esta resolución tan alta. El de cálculo de sombras sin embargo sí que reduce el rendimiento en alrededor de un 25 % en todos los

4 Pruebas y rendimiento

casos. De hecho, al aplicar ambos algoritmos simultáneamente se obtiene prácticamente el mismo rendimiento que aplicando solo el de sombras.

En la configuración de resolución baja obtenemos medidas en la misma línea, pero con una diferencia fundamental. Esta es que el impacto de aumentar el valor de AA es menor, en concreto de un 30 % en lugar de un 50 % (también ocurre con el cálculo de sombras, que pasa a una reducción del rendimiento de un 10 – 15 %). Esto es debido a que, si recordamos de la [Sección 2.3](#), el número de rayos a trazar por cada píxel crece en función de AA^2 , y por consiguiente reducir el número de pixels inicial disminuye también enormemente el número final de rayos. En efecto, dado que ahora trabajamos con un cuarto de la resolución anterior, el número de rayos a evaluar por píxel en esta configuración con $AA = n$ es el mismo que en la configuración de resolución alta con $AA = n - 2$. A primera vista podría parecer que el rendimiento debería ser el mismo en ambos escenarios, pero en la [Figura 4.4](#) podemos observar que esto no ocurre. El motivo de esto es que no todo el tiempo de renderizado se dedica a cálculos relacionados con los rayos, sino que el *shader* también emplea otra parte de este tiempo en cálculos que se hacen una única vez por píxel (independientemente de AA), y otra en tiempos fijos (independientes de AA y del número de pixels).

	$AA = 1$	$AA = 2$	$AA = 3$	$AA = 4$
2560 px × 1280 px	$3,2768 \times 10^6$	$6,5536 \times 10^6$	$13,1072 \times 10^6$	$26,2144 \times 10^6$
1280 px × 640 px	$0,8192 \times 10^6$	$1,6384 \times 10^6$	$3,2768 \times 10^6$	$6,5536 \times 10^6$

Tabla 4.1: Rayos a trazar en función de la resolución inicial y AA

Este decremento tan sustancial en el número de rayos hace que obtengamos una gran mejora en el rendimiento, como era de esperar, y desactivando las sombras obtenemos 144 Hz incluso con $AA = 2$. Es fácil suponer por tanto que para $AA = 1$ las medidas están claramente limitadas por la tasa de refresco del monitor y serían mucho mayores al valor medido, de en torno a los 200 Hz si asumimos que el rendimiento se reduce en un 30 % al aumentar AA y que para $AA = 2$ las medidas obtenidas no están siendo limitadas.

A vista de las gráficas anteriores podríamos pensar que usar $AA = 3$ es una buena opción, pues obtenemos unas medidas bastante buenas de FPS. En concreto, en resolución baja obtenemos alrededor de 100 FPS, y con resolución alta llegamos a conseguir 40 FPS. En ambos casos obtenemos valores por encima de los 30 FPS, que suele ser considerado el mínimo para una buena experiencia interactiva. Sin embargo, hay que tener en cuenta que hay otros factores además de los fotogramas por segundo que influyen en la fluidez de la imagen, entre los que destaca el *frametime*.

	Sobremesa	Portátil
GPU	NVIDIA GeForce GTX 1060 6GB	NVIDIA GeForce GTX 1650
CPU	Intel(R) Core(TM) i5-8400 @ 2.80GHz	AMD Ryzen 7 4800H @ 2.90 GHz
RAM	8,00 GB	16,0 GB
Pantalla	2560 px × 1440 px × 144 hz	1920 px × 1080 px × 60 hz

Figura 4.1: Especificaciones de los equipos usados en las pruebas

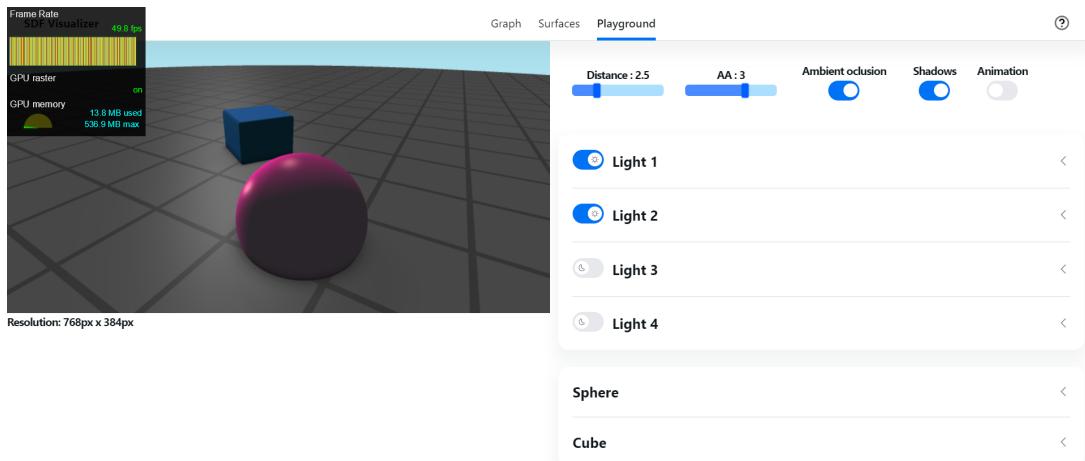


Figura 4.2: Sección Playground de la aplicación usada para medir el rendimiento de esta

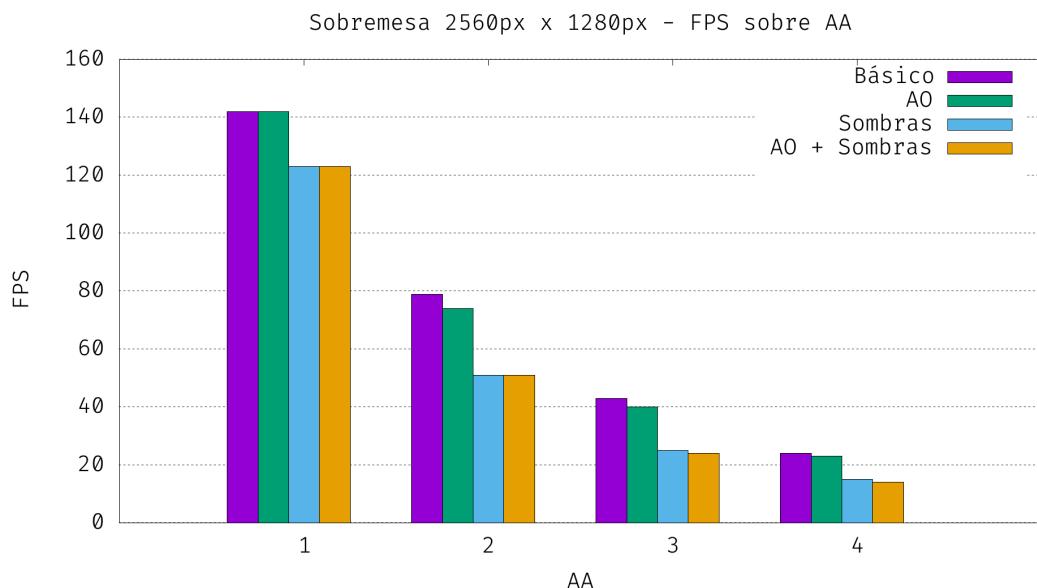


Figura 4.3: Rendimiento en ordenador de sobremesa con resolución alta

El *frametime* es el tiempo en milisegundos que se tarda en renderizar cada fotograma, incluyendo tanto los cálculos visuales como otros adicionales (motor de físicas, postprocesado, etc.). Queremos que este tiempo sea lo más estable posible a lo largo de la ejecución del programa, pues representa cada cuánto tiempo se nos presenta un fotograma nuevo en pantalla. Así, si el *frametime* fluctúa mucho tendremos la sensación de que la imagen no es fluida independientemente del número de fotogramas por segundo que se muestren, ya que para el cálculo de estos solo se tiene en cuenta el numero final de fotogramas presentados.

4 Pruebas y rendimiento

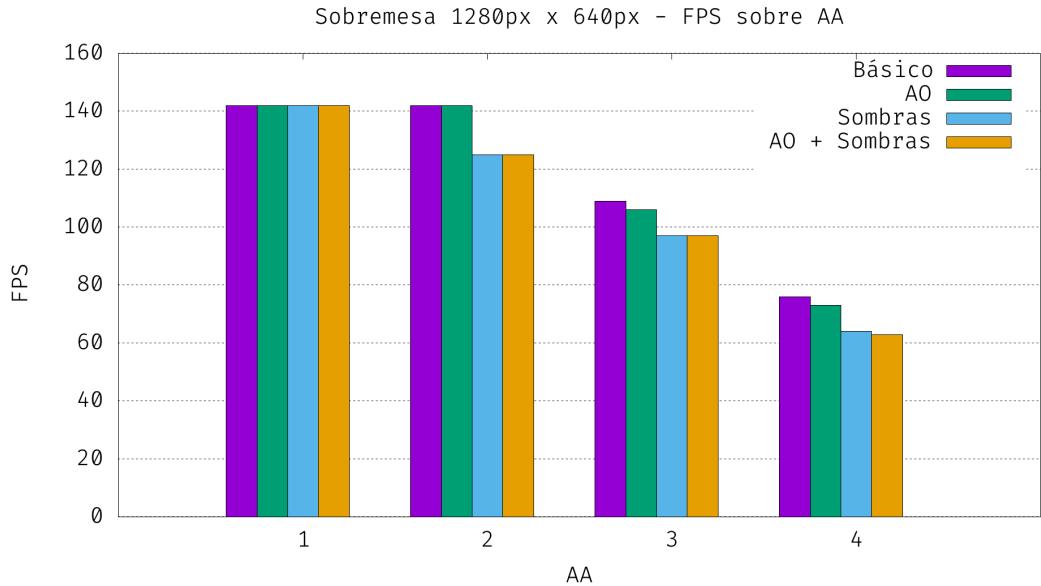


Figura 4.4: Rendimiento en ordenador de sobremesa con resolución baja

Por ejemplo, en una aplicación a 60 FPS, querríamos que el *frametime* fuera de

$$\textit{frametime} = \frac{1\text{s}}{60 \text{ fotogramas}} \cdot \frac{1000ms}{1\text{s}} = \frac{16,667ms}{\text{fotograma}}.$$



Figura 4.5: Videojuego God of War: Ragnarök obteniendo un *frametime* estable de 16.667 ms en PS5 [70]

Las herramientas de desarrollador de Google Chrome permiten también medir el *frametime* de nuestra aplicación. Para valores de *AA* menores a dos obtenemos una gráfica totalmente

plana, indicando que el *frametime* es muy estable, y en consecuencia el movimiento de la cámara del lienzo se percibe fluido. En cambio, a partir del valor $AA = 3$ empezamos a obtener una gráfica con muchos picos, a pesar de que estemos obteniendo un gran número de fotogramas por segundo, como muestra la [Figura 4.6](#). Podemos afirmar por tanto que el valor de $AA = 2$ es la mejor elección en general, pues nos proporciona una imagen mucho más nítida a la que obtendríamos sin *antialiasing* sin sacrificar la fluidez que espera el usuario.

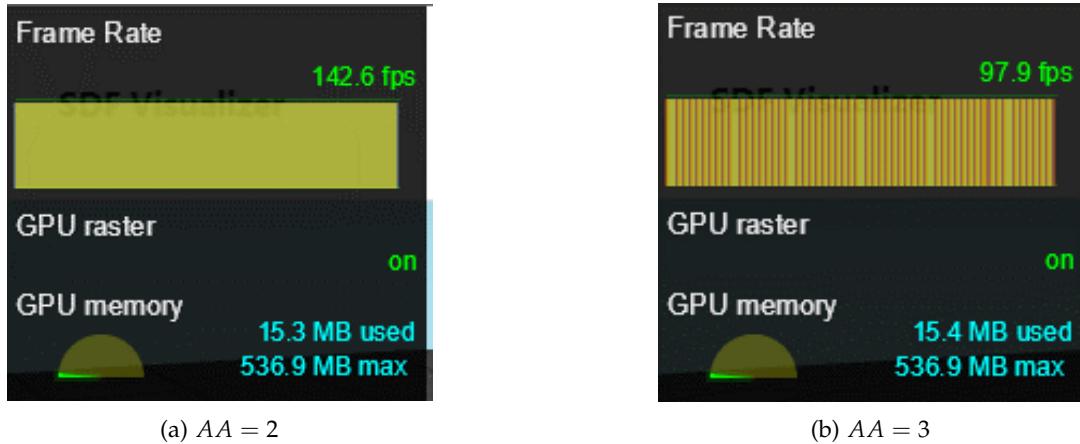


Figura 4.6: Gráficas de *frametime* según el valor de AA

En el ordenador portátil ([Figura 4.7](#)) medimos un comportamiento similar al del equipo de sobremesa, con la diferencia de que las medidas obtenidas decrecen proporcionalmente al peor *hardware*. En particular, ahora la pantalla ahora tiene una tasa de refresco y resolución menores. Con resolución alta (que en realidad es tan solo un poco mayor que la resolución baja del ordenador de sobremesa) obtenemos buenos resultados con hasta $AA = 2$, pero si seguimos aumentando el factor de escalado obtenemos FPS muy bajos, que aún sin tener en cuenta la inestabilidad del *frametime*, harían la aplicación inutilizable. Con resolución baja obtenemos resultados mucho mejores, y se ve claramente la limitación en el refresco de la pantalla a 60 Hz, pues obtenemos 60 FPS tanto con $AA = 1$ como $AA = 2$ con sombras y oclusión ambiental. Con $AA = 3$ obtenemos también buen rendimiento, pero el problema del *frametime* persiste, luego sigue siendo recomendable quedarnos con $AA = 2$.

A la vista de los resultados presentados, podemos concluir que la configuración más óptima para proporcionar una buena experiencia al usuario, tanto en rendimiento como en calidad visual, es tomar $AA = 2$ y activar la oclusión ambiental. En lo que respecta a las sombras, puede ser necesario tener que desactivarlas si se trabaja en un equipo con prestaciones muy bajas, pero en general son una opción viable y añaden mucha riqueza a la escena. Esta es precisamente la configuración de los lienzos que aparecen en cada nodo del editor de nodos. Uno podría pensar que en un caso de uso complejo del editor como fue el mostrado en la [Figura 3.3](#), el rendimiento se vería afectado al contar con tantas instancias del lienzo. Sin embargo, nada más lejos de la realidad, ya que se obtiene un rendimiento estable al mayor número de FPS que la pantalla permite en ambos equipos. ¿Cómo es esto posible? Principalmente por dos motivos.

1. Los lienzos presentes en los nodos tienen una resolución típica mucho menor a la

4 Pruebas y rendimiento

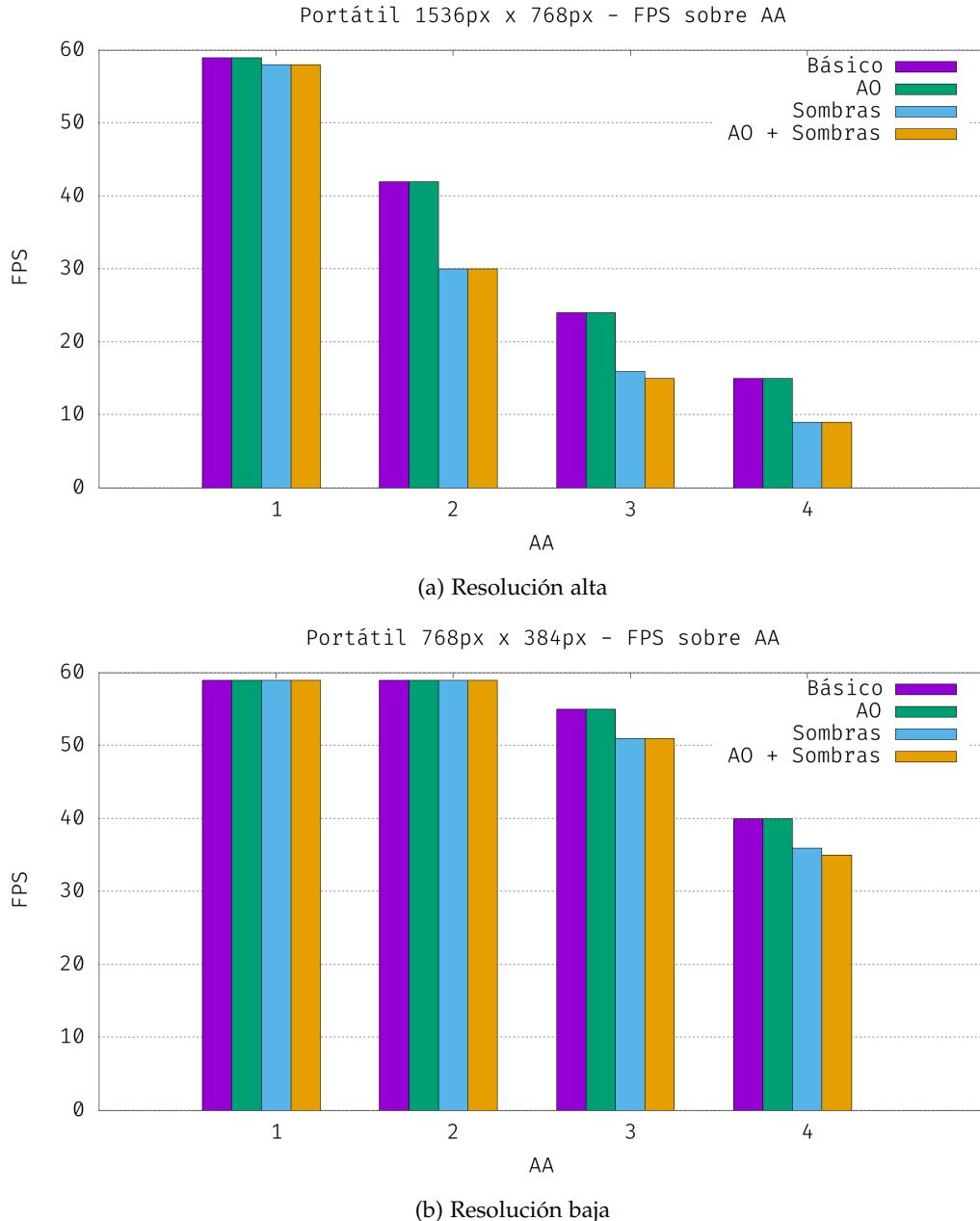


Figura 4.7: Rendimiento en ordenador portátil

usada en las pruebas.

2. El lienzo no calcula nuevos fotogramas a no ser que sea necesario. Es por este motivo que en el lienzo de prueba la cámara se movía continuamente, pero en el caso del editor la cámara está fija a no ser que el usuario interactúe con ella mediante el ratón. Esto hace que en cada momento como mucho habrá un solo lienzo que se esté actualizando

continuamente, mientras que el resto estarán en un estado de reposo.

4.2. Librería de polinomios en varias variables

Dado que las clases que conforman la librería tienen un alto grado de dependencia entre sí, primero se desarrolló Monomial, después Polynomial, y finalmente Ideal. Durante el desarrollo de todas ellas se fueron generando tests para comprobar su correcto funcionamiento, desde los métodos más básicos hasta comprobaciones de seguridad en casos extremos. Las principales ventajas que nos ha aportado esta forma de trabajar son las siguientes.

- Al trabajar, por ejemplo, sobre la clase Polynomial, al ya habernos cerciorado de que Monomial funciona correctamente, podemos estar seguros de que los fallos que nos encontramos estarán dentro de esta nueva clase y enfocar nuestros esfuerzos en consecuencia.
- En varias ocasiones se introdujeron cambios de importancia en alguna de las clases, por ejemplo cambiar el tipo del atributo coef de Monomial de number a Fraction. Al contar con los tests podemos comprobar de forma rápida que todo sigue funcionando correctamente y de forma transparente al resto de componentes que la usen.

Para la generación de los tests se ha usado MochaJS [71], que también nos permite medir el rendimiento de los tests que deseemos. Los tests se encuentran disponibles en el mismo repositorio que el de la librería. Los que comprueban los métodos básicos de las clases Monomial y Polynomial, tales como suma, producto, comparaciones o inserción de variables, se encuentran en los archivos tests/monomial.ts y tests/polynomial.ts, y no requieren mucha explicación. Sí son más interesantes otros dos archivos de tests adicionales, uno de Polynomial para todos los métodos relacionados con bases de Gröbner (cálculo, reducción, comprobación de si es base, etc.), y otro para Ideal con pruebas de aplicación del algoritmo de implicitación.

En el caso de los tests de Polynomial, es de vital importancia asegurarnos de que los métodos que comprueban si un conjunto de polinomios es base de Gröbner de un ideal dado y si una base es reducida funcionan correctamente, pues usaremos estos para el resto de tests más adelante. Para ello nos hemos apoyado en el uso de SageMath, que nos proporciona métodos que nos permiten realizar estas comprobaciones fácilmente. Dado un ideal, estos son `is_groebner`, que verifica si una lista de polinomios es base de Gröbner del ideal, y `groebner_basis()`, que calcula una base de Gröbner reducida suya. Un ejemplo de uso de estos métodos es el siguiente.

```
x,y,z = QQ['x,y,z'].gens()
I = ideal(x^5 + y^4 + z^3 - 1, x^3 + y^3 + z^2 - 1)
B = I.groebner_basis()
B.is_groebner() # True
```

Una vez convencidos de que estos métodos de comprobación funcionan, podemos usarlos para validar los tests de las funciones que calculan y reducen bases de Gröbner. Como no tendremos que hacer las comprobaciones a mano, nos podemos permitir generar un gran número de tests, en este caso de hasta 50 ideales con conjuntos de generadores cada vez más

4 Pruebas y rendimiento

complejos. Es esta además una buena oportunidad para empezar a realizar mediciones sobre el rendimiento de la librería. Además del tiempo de cómputo necesario para calcular todas las bases, podemos medir también el impacto que tiene el uso de los criterios de Buchberger introducidos en el [Teorema 1.6](#). Otro factor de nuestro interés es medir si la inclusión del uso de nuestra propia clase de fracciones en lugar del tipo genérico `number` tiene algún impacto en el rendimiento. En la [Figura 4.8](#) se muestran todas estas comparaciones sobre un conjunto de veinte ideales en el ordenador portátil (todas las pruebas de esta sección se realizarán en este equipo).

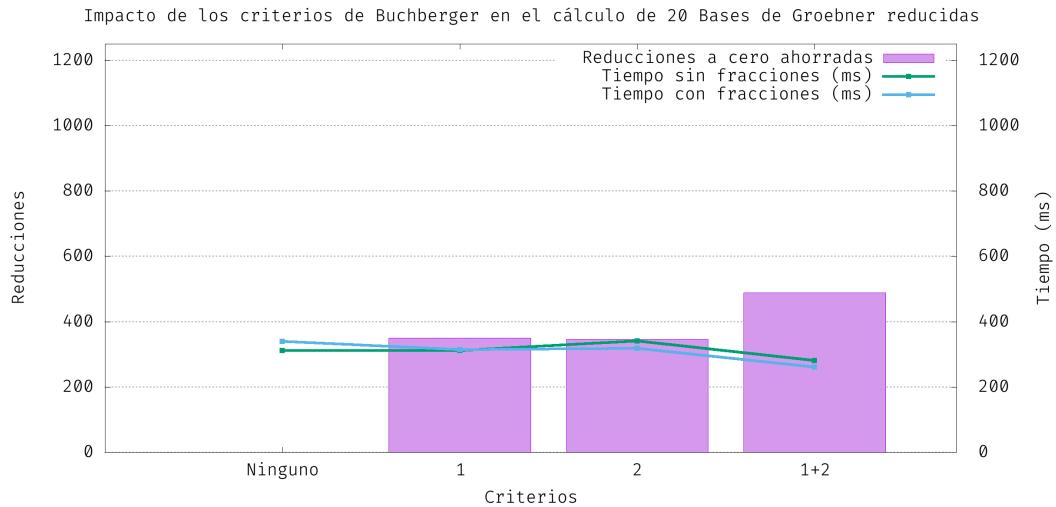


Figura 4.8: Rendimiento del cálculo de bases de Gröbner reducidas en una muestra de 20 ideales

Si empezamos observando el número de reducciones a cero innecesarias, ambos criterios detectan un número similar ellas. No obstante, este número crece si ambos criterios se usan en tandem, lo cual indica que los casos que detectan uno y otro difieren, en este caso sobre un 30 %. Como consecuencia de esto los tiempos son también menores cuando se usan ambos criterios, aunque la diferencia no es muy significativa respecto a no usar ninguno en este caso. El uso de fracciones no parece tener tampoco ninguna repercusión destacable en los tiempos obtenidos.

Si aumentamos el número de bases a calcular a 50 observamos que varias medidas que en la prueba anterior parecían similares ahora empiezan a distanciarse. Como primera diferencia vemos que el primer criterio ha conseguido detectar más reducciones innecesarias que el segundo, aunque este último mantiene la tendencia de detectar un 30 % diferentes a las del primero. Otra ventaja del primer criterio sobre el segundo es que los casos que detecta parecen ser los más útiles, ya que el tiempo total de ejecución entre usar el segundo criterio o no usar ninguno es el mismo, así como el de usar el primero o ambos a la vez. Respecto a las diferencias entre usar la clase `Fraction` o no, encontramos resultados sorprendentes, y es que usar fracciones nos proporciona mejor rendimiento que el tipo nativo `number` en todos los casos. Esta ventaja a priori tan inesperada tiene una explicación muy simple si analizamos la implementación de nuestra clase, en concreto del método de división. En él se calcula la

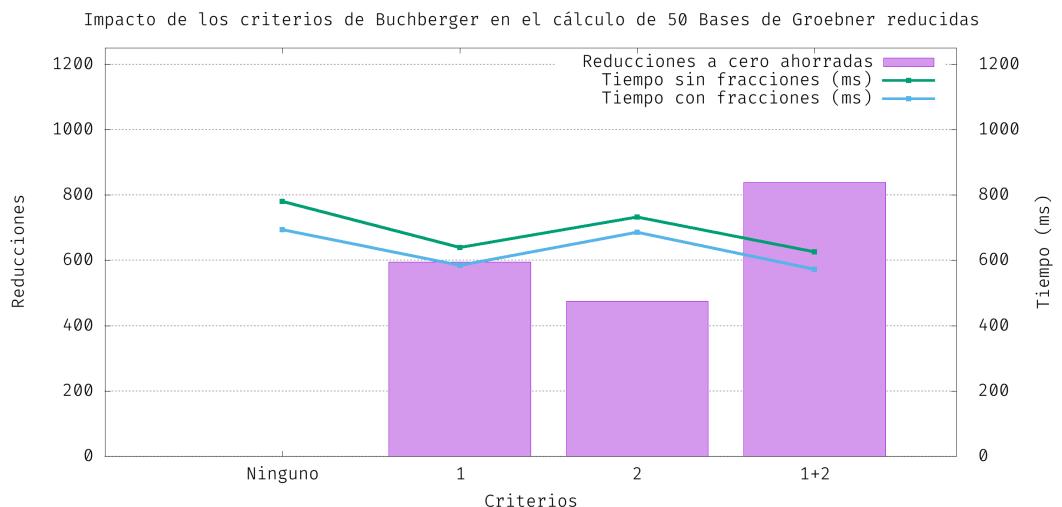


Figura 4.9: Rendimiento del cálculo de bases de Gröbner reducidas en una muestra de 50 ideales

división de dos fracciones como el producto cruzado de sus numeradores y divisores. Es decir, para dividir dos fracciones en ningún momento estamos realizando una división de flotantes, sino dos multiplicaciones. Lo que en última instancia nos importa para mejorar el rendimiento es consumir la menor cantidad posible de ciclos de reloj, y mientras que una multiplicación únicamente usa 1-2 ciclos, una división puede llegar a exceder los 24. Como en el algoritmo de división de polinomios se realizan cientos de reducciones del término líder, y en cada una de ellas dos divisiones, esta diferencia de ciclos se acumula hasta llegar a suponer una ventaja de 100 ms en nuestras pruebas.

Una vez analizado el comportamiento de la librería sobre un conjunto amplio de datos, pasamos a estudiar casos más concretos y prácticos del uso de cálculo de bases de Gröbner reducidas: la implicitación. Los tests de la clase `Ideal` buscan comprobar el correcto funcionamiento del algoritmo de implicitación implementado, para lo cual se han usado parametrizaciones racionales del plano y diversas superficies cuádricas de las que conocemos su ecuación implícita. Para tener más datos sobre los que trabajar, también se han usado parametrizaciones aleatorias cuya ecuación implícita asociada se ha obtenido con la ayuda de SageMath con el siguiente código.

```
# Definimos anillo
# l es la variable de eliminacion; s,t variables de las parametrizaciones
P.<s,t,l,x,y,z> = PolynomialRing(QQ, 6, order='lex')

def implicitate(f1,f2,f3,q1,q2,q3):

    I = ideal(q1*x-f1, q2*y-f2, q3*z-f3, 1-q1*q2*q3*l)
    B = I.groebner_basis()

    # generadores del ideal de eliminacion
```

```
return [p for p in B if all(var not in p.variables() for var in [s, t, l])]
```

Todas las parametrizaciones usadas en los tests se listan en la [Tabla 4.2](#) junto a sus ecuaciones implícitas, las cuales la librería ha calculado sin problema.

Pasando al rendimiento, podemos ver el tiempo empleado para cada parametrización en la [Figura 4.10](#). Para las superficies cuádricas se obtienen muy buenos resultados, de en torno a los 10 ms. La única excepción es el caso de la esfera, que es muy demandante debido al gran número de reducciones a cero que acarrea. Tiempos similares a los de la esfera se obtienen para el cálculo de las implícitas obtenidas con Sage, que sin llegar a ser de un segundo, sí es cierto que puede llegar a ser un elemento distractivo en una aplicación interactiva, aunque para su uso en diferido se trata de tiempos razonables.

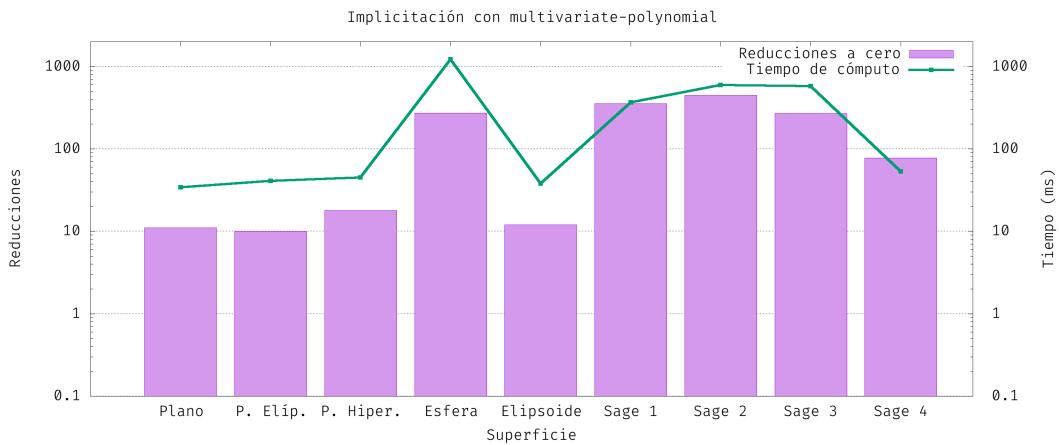


Figura 4.10: Rendimiento de la implicitación sobre varias parametrizaciones

Acabamos el capítulo mostrando una comparativa de tiempos entre nuestra librería y Sage, a sabiendas de que obtendremos resultados bastante peores por dos motivos principales.

1. Sage es de los *software* de código abierto más completos y optimizados. En concreto, para el cálculo de bases de Gröbner usa Singular [72], la librería de código abierto más potente en cuanto al cálculo de bases de Gröbner a fecha de realización de este trabajo. Al estar escrita en C++, un lenguaje compilado y fuertemente tipado, no es de extrañar esta diferencia con nuestra implementación en TypeScript, que no deja estar basado en JavaScript, el cual es interpretado.
2. JavaScript no está pensado para optimizar este tipo de cálculos, sino que está diseñado para ejecutarse en navegadores web y brindar interactividad a las páginas web. Además, si bien nuestra versión del algoritmo implementa importantes optimizaciones, esta podría admitir otras que no se han abordado en este trabajo.

El principal objetivo de esta comparación es por tanto ver cómo de cerca hemos conseguido quedarnos respecto a SageMath. En la [Figura 4.11](#) vemos que usando groebner_basis, Sage consigue realizar cada una de las implicitaciones en torno a los 2 ms, siendo estas parametrizaciones en tan solo dos variables muy sencillas para esta versión del algoritmo. Usando el

Superficie	Ecuación implícita	Parametrización racional
Plano	$x + y + z - 1 = 0$	$\begin{cases} x = s + t \\ y = s \\ z = t \end{cases}$
Paraboloide elíptico	$x - y^2 - z^2 = 0$	$\begin{cases} x = t^2 + s^2 \\ y = t \\ z = s \end{cases}$
Paraboloide hiperbólico	$x^2 - y^2 - z = 0$	$\begin{cases} x = t \\ y = s \\ z = t^2 - s^2 \end{cases}$
Esfera unidad	$x^2 + y^2 + z^2 - 1 = 0$	$\begin{cases} x = \frac{2s}{s^2+t^2+1} \\ y = \frac{2t}{s^2+t^2+1} \\ z = \frac{s^2+t^2-1}{s^2+t^2+1} \end{cases}$
Elipsoide	$x^2 + y^2 - z = 0$	$\begin{cases} x = s \\ y = t \\ z = s^2 + t^2 \end{cases}$
Sage 1	$x + y - 1 = 0$	$\begin{cases} x = \frac{t-s}{t} \\ y = \frac{s}{t} \\ z = \frac{s^3+t^2}{t^2} \end{cases}$
Sage 2	$xz - x - 1 = 0$	$\begin{cases} x = \frac{s}{t} \\ y = \frac{s}{t^2} \\ z = \frac{s+t}{s} \end{cases}$
Sage 3	$x + yz - z = 0$	$\begin{cases} x = \frac{s}{st} \\ y = \frac{t^2-s}{t^2} \\ z = \frac{t}{s} \end{cases}$
Sage 4	$y - \frac{z^2}{3} = 0$	$\begin{cases} x = \frac{t-s^2}{st} \\ y = \frac{s^2}{3} \\ z = s \end{cases}$

Tabla 4.2: Superficies usadas en los tests de implicitación

4 Pruebas y rendimiento

método buchberger, aún siendo este una versión básica del algoritmo de Buchberger, consigue ser más rápido que nuestra implementación por los motivos mencionados anteriormente, a excepción del caso de la esfera, donde obtiene tiempos similares a nuestra versión.

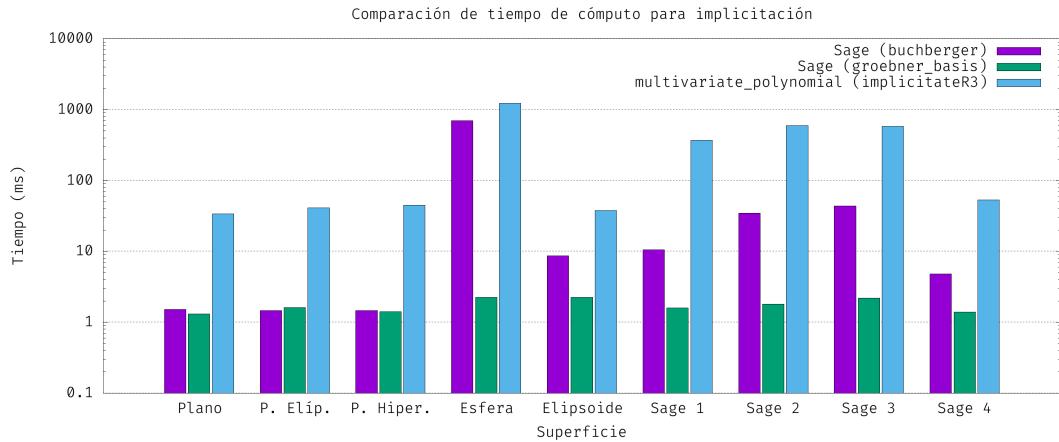


Figura 4.11: Comparativa de rendimiento entre Sage y multivariate-polynomial

5 Conclusión y líneas futuras

Una vez expuestos todos los resultados de los capítulos anteriores, podemos concluir que hemos cumplido todos los objetivos propuestos en la introducción de este trabajo, además de haber realizado varios logros clave:

1. Estudio de los **algoritmos de rasterización y raytracing** de visualización por computador, y cómo combinarlos para el renderizado de superficies definidas por SDFs de forma eficiente en la gran mayoría de dispositivos actuales usando técnicas de iluminación avanzadas y *antialiasing*, elaborando los *vertex* y *fragment shaders* necesarios.
2. Implementación de una **librería de código abierto propia en TypeScript** para el manejo de polinomios en varias variables, incluyendo implementaciones de algoritmos de cálculo de bases de Gröbner e implicitación.
3. Desarrollo de una **aplicación web con React eficiente, accesible a todo el mundo, y de código abierto**, que permite definir y modificar con una gran variedad de operadores superficies definidas mediante SDFs y ecuaciones paramétricas o implícitas.

En el transcurso de la obtención de estos resultados ha sido necesario hacer una intensiva labor de investigación a través de artículos y documentaciones del ámbito tanto matemático como informático. Además, para asegurar la calidad y buen funcionamiento de los productos *software* elaborados, tanto de la aplicación como de la librería, se han realizado pruebas de rendimiento y tests de forma constante durante su desarrollo.

Considero que el *software* desarrollado en este trabajo realiza una valiosa **contribución** al panorama actual. La librería de polinomios es la única que existe a fecha de realización de este trabajo para trabajar con polinomios en varias variables y bases de Gröbner de forma nativa en TypeScript. De forma similar ocurre con la aplicación web. Si bien existen aplicaciones web que permiten manipular SDFs, como se mencionó al principio del documento, ninguna de ellas permite obtener SDFs a partir de ecuaciones implícitas o paramétricas, y aunque implementen el modelo de árbol, todas presentan carencias que nuestra aplicación cubre, ya sea su facilidad de uso, disponibilidad por ser web, o experiencia de usuario.

Entre las **futuras líneas** de desarrollo de la aplicación es de especial interés añadir la posibilidad de trabajar con mallas de polígonos. Existen métodos tanto para obtener una SDF aproximada dada una malla de triángulos [73, 74, 75, 76], como para obtener una malla a partir de una SDF [77, 78, 79, 80], de entre los que destaca el conocido algoritmo *Marching Cubes* [81]. La inclusión de esta funcionalidad ([Figura 5.1](#)) sería un añadido muy interesante a la aplicación, pues permitiría un flujo de trabajo mucho más cómodo entre ella y programas de edición y visualización de geometría de terceros, ya que trabajar con mallas sigue siendo lo más común en la industria actual. Otra opción interesante es la de incluir la posibilidad de obtener la malla a partir de las ecuaciones paramétricas o implícitas. En efecto, para las paramétricas existen técnicas bastante sencillas para este fin, dado que al evaluarse estas siempre producen puntos en la superficie. Así, un método simple consistiría en evaluar las

ecuaciones paramétricas en una rejilla de puntos del plano para obtener los vértices de la malla en el espacio 3D. En el caso de las implícitas, en la [Sección 1.5](#) ya mencionamos el artículo de P.-A. Fayolle [33] en el que se presentaba un método para obtener una SDF que represente de forma exacta la frontera de la superficie, aunque este se basaba en el uso de redes neuronales. Sería interesante por tanto la investigación de técnicas que se adapten mejor a nuestro caso de uso y que, dado un conjunto de superficies por ecuaciones implícitas y operaciones booleanas, deformantes, etc., permitan calcular la SDF. Una vez conocida la SDF, podríamos obtener la malla usando alguna de las técnicas ya mencionadas, como *Marching Cubes*.

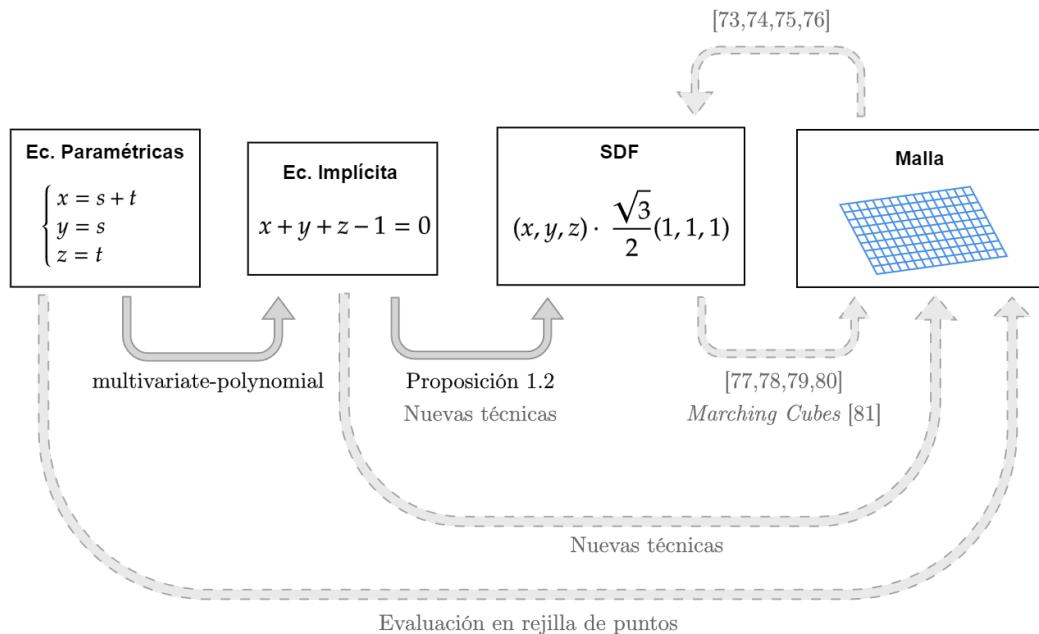


Figura 5.1: Inclusión de compatibilidad con mallas de polígonos

Otras posibilidades de mejora más sencillas para la aplicación son las relacionadas con la extensión de formas de manipulación de las superficies definidas por SDFs, tales como añadir la posibilidad de trabajar con texturas o la definición de materiales procedurales, o pulir ciertos aspectos de compatibilidad en dispositivos móviles, en los que si bien la aplicación es funcional, la interfaz o la visualización de superficies presenta defectos en algunos modelos.

La librería `multivariate-polynomial` tiene también margen de mejora, siendo de especial interés permitir usar otros órdenes admisibles que no sean el lexicográfico, como el lexicográfico graduado inverso o cualquier otro que defina el usuario. Quedan también versiones y mejoras del algoritmo de Buchberger por investigar [82, 83, 84, 85]. Es cierto que muchas de ellas serían poco prácticas en el entorno de una aplicación web [86], pero podrían suponer una mejora muy interesante para el uso de la librería en otros ámbitos. Ambos cambios mencionados se verían también reflejados en el rendimiento de la aplicación, pues el orden

elegido influye en el número de reducciones a cero realizadas en el algoritmo de Buchberger, y este representa la mayor parte del tiempo de ejecución del algoritmo de implicitación racional. También merece la pena explorar algoritmos alternativos para el cálculo de bases de Gröbner, como es el F_4 o el F_5 [87, 88]. Por último, otra vía interesante es expandir la funcionalidad de la clase `Ideal` más allá de su uso para realizar implicitación, añadiendo nuevas operaciones, como la suma, división o saturación de ideales, y métodos que comprueben ciertas propiedades del ideal, como si es primario, su dimensión o la pertenencia de un polinomio a su radical, etc.

Bibliografía

- [1] J. F. Blinn, "A generalization of algebraic surface drawing," *ACM transactions on graphics (TOG)*, vol. 1, no. 3, pp. 235–256, 1982.
- [2] A. Pasko, V. Adzhiev, A. Sourin, and V. Savchenko, "Function representation in geometric modeling: concepts, implementation and applications," *The visual computer*, vol. 11, pp. 429–446, 1995.
- [3] 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.
- [4] B. Foundation, "Blender - 3d creation software - git repository," Último acceso: 6 mayo 2023.
- [5] Clayxels Development Team, "Clayxels." Software de modelado y animación basado en SDFs.
- [6] "Mesh distance fields." <https://docs.unrealengine.com/4.27/en-US/BuildingWorlds/LightingAndShadows/MeshDistanceFields/>. [En línea]. Último acceso: 28 agosto 2023.
- [7] S. Aaltonen, "GPU-based clay simulation and ray-tracing tech in claybook." https://ubm-twvid-eo01.s3.amazonaws.com/o1/vault/gdc2018/presentations/Aaltonen_Sebastian_GPU_Based_Clay.pdf. [En línea]. Último acceso: 28 agosto 2023.
- [8] M. Molecule, "Dreams," 2020.
- [9] I. Quilez, "Inigo quilez." <https://iquilezles.org/>.
- [10] P. Jeremias and I. n. Quilez, "Shadertoy: Live coding for reactive shaders," in *ACM SIGGRAPH 2013 Computer Animation Festival, SIGGRAPH '13*, (New York, NY, USA), p. 1, Association for Computing Machinery, 2013.
- [11] Í. Quílez, "Selfie girl." <https://www.shadertoy.com/view/WsSBzh>, Nov. 2020. [En línea]. Último acceso: 29 agosto 2023.
- [12] EvilRyu, "Mandelbulb." <https://www.shadertoy.com/view/MdXSWn>, May 2014. [En línea]. Último acceso: 29 agosto 2023.
- [13] "hg_sdf." https://github.com/jcowles/hg_sdf. glsl library for building signed distance functions.
- [14] matkcy, "Implicit 3D." <https://github.com/matkcy/MA1104>. Implicit surface visualization.
- [15] Geogebra , "Parametric surface." Parametric surface visualizer.
- [16] J. Bailey, "An interactive shader editor made for programmers and artists," 2021.
- [17] "Material Maker." <https://rodzill4.github.io/material-maker/doc/index.html>. PBR procedural material editor as well as a texture painting tool based on the Godot Engine.
- [18] "SDFPerf." https://github.com/jcowles/hg_sdf. glsl library for building signed distance functions.
- [19] "SDF-UI." <https://github.com/szymonkaliski/SDF-UI>. Experimental node-based UI for generating SDF shaders in a browser.
- [20] T. W. Sederberg and A. K. Zundel, "Scan line display of algebraic surfaces," in *International Conference on Computer Graphics and Interactive Techniques*, 1989.
- [21] 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.

Bibliografía

- [22] 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.
- [23] <https://math.okstate.edu/people/binegar/4013-U98/4013-106.pdf>. [En línea]. Último acceso: 30 mayo 2023.
- [24] C. Bálint, G. Valasek, and L. Gergó, "Operations on signed distance functions," *Acta Cybern.*, vol. 24, no. 1, pp. 17–28, 2019.
- [25] Wikipedia contributors, "Función distancia con signo." https://es.wikipedia.org/w/index.php?title=Funci%C3%B3n_distancia_con_signo&oldid=149321457. [En línea]. Último acceso: 21 mayo 2023.
- [26] 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.
- [27] M. Delfour and J.-P. Zolésio, *Shapes and Geometries: Analysis, Differential Calculus and Optimization*, vol. 4, 01 2001.
- [28] A. R. Sarabia, "Apuntes de la asignatura curvas y superficies de la universidad de granada," curso 21-22.
- [29] O. Knill, "Math s21a: Multivariable calculus." <https://people.math.harvard.edu/~knill/teaching/summer2011/handouts/34-gradient.pdf>. [En línea]. Último acceso: 18 septiembre 2022.
- [30] I. Quílez, "Normals for an sdf." <https://iquilezles.org/articles/normalssdf/>, [En línea]. Último acceso: 6 mayo 2023.
- [31] I. Quílez, "Smooth minimun." <https://iquilezles.org/articles/smin/>, [En línea]. Último acceso: 6 mayo 2023.
- [32] A. H. Barr, "Global and local deformations of solid primitives," *ACM Siggraph Computer Graphics*, vol. 18, no. 3, pp. 21–30, 1984.
- [33] I. Quílez, "Soft shadows in raymarched sdbs." <https://iquilezles.org/articles/rmshadows/>, [En línea]. Último acceso: 31 agosto 2023.
- [34] P.-A. Fayolle, "Signed distance function computation from an implicit surface," *arXiv preprint arXiv:2104.08057*, 2021.
- [35] 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.
- [36] D. Y. Melesse, "Groebner bases and an improvement on buchberger's algorithm," 2007.
- [37] A. S. Semyonov, *Buchberger's Criteria and Trivial Syzygies*.
- [38] E.-W. Chionh and R. N. Goldman, "Using multivariate resultants to find the implicit equation of a rational surface," *The Visual Computer*, vol. 8, pp. 171–180, 1992.
- [39] B. Grenet, P. Koiran, and N. Portier, "On the complexity of the multivariate resultant," *Journal of Complexity*, vol. 29, no. 2, pp. 142–157, 2013.
- [40] Wikipedia contributors, "Homogeneous polynomial." https://en.wikipedia.org/w/index.php?title=Homogeneous_polynomial&oldid=1160440258, June 2023.
- [41] E.-W. Chionh and R. N. Goldman, "Implicitizing rational surfaces with base points by applying perturbations and the factors of zero theorem," in *Mathematical methods in computer aided geometric design II*, pp. 101–110, Elsevier, 1992.
- [42] D. Manocha and J. F. Canny, "Implicit representation of rational parametric surfaces," *Journal of Symbolic Computation*, vol. 13, no. 5, pp. 485–510, 1992.
- [43] OpenGL, "Coordinate systems." <https://learnopengl.com/Getting-started/Coordinate-Systems>. [En línea]. Último acceso: 9 mayo 2023.

- [44] OpenGL, "Matrices." <http://www.opengl-tutorial.org/es/beginners-tutorials/tutorial-3-matrices/#matrices-modelo-vista-y-proyecci%C3%B3n>. [En línea]. Último acceso: 10 mayo 2023.
- [45] C. Ureña, "Apuntes de la asignatura informática gráfica de la universidad de granada," curso 21-22.
- [46] B. T. Phong, "Illumination for computer generated pictures," *Commun. ACM*, vol. 18, p. 311–317, jun 1975.
- [47] 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. [En línea]. Último acceso: 20 mayo 2023.
- [48] J. F. Blinn, "Models of light reflection for computer synthesized pictures," in *Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pp. 192–198, 1977.
- [49] nurof3n, "Ray marched improved shadows." <https://www.shadertoy.com/view/tscSRS>. [Online; Último acceso 7-Mayo-2023].
- [50] I. Quílez, "Distance stimation." <https://iquilezles.org/articles/distance/>, [En línea]. Último acceso: 6 junio 2023.
- [51] "The book of shaders." <https://thebookofshaders.com/glossary/?search=smoothstep>. [En línea]. Último acceso: 1 junio 2023.
- [52] Digital Foundry, "Ratchet and clank: Rift apart PS5 - performance + graphics - all modes tested!," June 2021.
- [53] A. Evans, "Fast approximations for global illumination on dynamic scenes," in *ACM SIGGRAPH 2006 Courses*, pp. 153–171, 2006.
- [54] G. Geer, "Normally sampled ambient occlusion." <https://www.csh.rit.edu/~gman/PersonalWebpage/ao.html>. [En línea]. Último acceso: 4 junio 2023.
- [55] J. Jimenez, D. Gutierrez, J. Yang, A. Reshetov, P. Demoreuille, T. Berghoff, C. Perthuis, H. Yu, M. McGuire, T. Lottes, *et al.*, "Filtering approaches for real-time anti-aliasing," *SIGGRAPH Courses*, vol. 2, no. 3, p. 4, 2011.
- [56] K. B. D. Barron, "Super-sampling anti-aliasing analyzed,"
- [57] A. Fridvalszky and B. Tóth, "Multisample anti-aliasing in deferred rendering," in *Eurographics (Short Papers)*, pp. 21–24, 2020.
- [58] Wikipedia contributors, "Supersampling." <https://en.wikipedia.org/w/index.php?title=Supersampling&oldid=1143405382>, Mar. 2023. [En línea]. Último acceso: 4 junio 2023.
- [59] Facebook, Inc., "React," 2013. JavaScript library for building user interfaces.
- [60] "Zustand," 2020. State management library for React applications.
- [61] "React Flow," 2021. Library for building interactive node-based graphs in React applications.
- [62] "gl-react." <https://github.com/gre/gl-react>, 2015. Library for adding WebGL-powered effects to React applications.
- [63] "shadertoy-react." <https://github.com/mvilledieu/shadertoy-react>, 2021. React component to create responsive shader canvases.
- [64] K. Group, "Webgl - web graphics library," 2009.
- [65] M. Hohenwarter, M. Borcherds, G. Ancsin, B. Bencze, M. Blossier, A. Delobelle, C. Denizet, J. Éliás, A. Fekete, L. Gál, Z. Konečný, Z. Kovács, S. Lizelfelner, B. Parisse, and G. Sturr, "Geogebra." <http://www.geogebra.org>.
- [66] W. Stein *et al.*, *Sage Mathematics Software*. The Sage Development Team. <http://www.sagemath.org>.
- [67] @jiggzson, "Nerdamer: Computer algebra system in javascript." <https://github.com/jiggzson/nerdamer>, "2023". [Online; Último acceso 10-Julio-2023].

Bibliografía

- [68] “Webgl how it works.” <https://webglfundamentals.org/webgl/lessons/webgl-how-it-works.html>. [Online; Último acceso 30-Julio-2023].
- [69] “Webgl fundamentals.” <https://web.dev/webgl-fundamentals/>. [Online; Último acceso 30-Julio-2023].
- [70] Digital Foundry, “God of war ragnarök on PS5 - the digital foundry tech review.” <https://www.youtube.com/watch?v=5-fi23nZGpM>.
- [71] “Mocha.” <https://github.com/mochajs/mocha>. A feature-rich JavaScript test framework.
- [72] “Singular.” <https://github.com/Singular/Singular>. Computer algebra system for polynomial computations.
- [73] J. Bærentzen and H. Aanæs, “Generating signed distance fields from triangle meshes. 2002,” *Informatics and Mathematical Modeling, Technical University of Denmark, DTU*, vol. 20.
- [74] U. Technologies, “Mesh to sdf unity package.” <https://github.com/Unity-Technologies/com.unity.demoteam.mesh-to-sdf>.
- [75] I. Quilez, “Distance to triangle.” <https://iquilezles.org/articles/triangledistance/>, [En línea]. Último acceso: 20 julio 2023.
- [76] E. Games, “Mesh distance fields.” <https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/GettingStarted/>, [En línea]. Último acceso: 23 julio 2023.
- [77] The CGAL Project, “CGAL 5.6 - 3D mesh generation: User manual.” https://doc.cgal.org/latest/Mesh_3/index.html. [En línea]. Último acceso: 22 agosto 2023.
- [78] Z. Zeng, “Approximating signed distance field to a mesh by artificial neural networks,” 2018.
- [79] “Computing signed distances (SDFs) to meshes.” https://www.fwilliams.info/point-cloud-utils/sections/mesh_sdf/. [En línea]. Último acceso: 22 agosto 2023.
- [80] M. Fogleman, “sdf: Simple SDF mesh generation in python.”
- [81] W. E. Lorensen and H. E. Cline, “Marching cubes: A high resolution 3d surface construction algorithm,” in *Seminal graphics: pioneering efforts that shaped the field*, pp. 347–353, 1998.
- [82] T. Sasaki, “An attempt to enhance buchberger’s algorithm by using remainder sequences and gcd operation,” in *2019 21st International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pp. 27–34, IEEE, 2019.
- [83] P. Horan and J. Carminati, “Performance of buchberger’s improved algorithm using prime based ordering,” *arXiv preprint arXiv:0901.4404*, 2009.
- [84] W. Trinks, “On improving approximate results of buchberger’s algorithm by newton’s method,” *ACM SIGSAM Bulletin*, vol. 18, no. 3, pp. 7–11, 1984.
- [85] J. Perry, “An extension of buchberger’s criteria for gröbner basis decision,” *LMS Journal of Computation and Mathematics*, vol. 13, pp. 111–129, 2010.
- [86] J. Mojsilović, D. Peifer, and S. Petrović, “Learning a performance metric of buchberger’s algorithm,” *arXiv preprint arXiv:2106.03676*, 2021.
- [87] J.-C. Faugere, “A new efficient algorithm for computing gröbner bases (f4),” *Journal of pure and applied algebra*, vol. 139, no. 1-3, pp. 61–88, 1999.
- [88] I. Kikuchi and A. Kikuchi, “Quantum computation of groebner basis,” *OSF Preprints. September*, vol. 8, 2021.