



ugr

Universidad  
de Granada

TRABAJO FIN DE GRADO  
GRADO EN INGENIERÍA INFORMÁTICA

# Desarrollo de una aplicación de modelado 3D

---

Motor de renderizado en tiempo real basada en SDFs

**Autor**

Pablo Cantudo Gómez

**Directores**

Juan Carlos Torres Cantero y Luis López Escudero



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE  
TELECOMUNICACIÓN

Granada, Septiembre de 2025







# Desarrollo software de una aplicación de modelado 3D basada en Signed Distance Functions (SDF)

Pablo Cantudo Gómez

**Palabras clave:** WebGPU, C++, ray marching, motor de renderizado, funciones de distancia con signo (SDF)

## Resumen

Este trabajo presenta el desarrollo de Copper, un motor de renderizado 3D en tiempo real implementado en C++ utilizando la API WebGPU a través de Dawn. El motor emplea técnicas de ray marching sobre funciones de distancia con signo (SDF), lo que permite una representación eficiente y flexible de geometría tridimensional. Una de las principales ventajas de utilizar SDF es la capacidad de expresar de forma directa y compacta operaciones booleanas entre primitivas geométricas —como uniones, intersecciones o diferencias— mediante simples expresiones algebraicas. Además, a diferencia de los métodos tradicionales basados en mallas o polígonos, las SDF permiten aplicar operaciones booleanas suaves, como la unión con suavizado (smooth union), de forma prácticamente trivial desde el punto de vista computacional.

Este enfoque simplifica notablemente la construcción de formas complejas, evitando problemas típicos de la computación geométrica como el manejo de vértices, normales o topologías complejas. También se facilita la animación y transformación de objetos mediante funciones continuas. El motor incluye un sistema de sombreado en WGSL, con soporte para sombras suaves, operaciones modulares sobre la escena, selección de objetos, carga y guardado de modelos. Los resultados demuestran que el uso de SDF no solo ofrece un modelo más elegante para definir geometría, sino que permite construir escenas visualmente complejas con menos código y una mayor expresividad gráfica. En conjunto, el sistema demuestra la viabilidad técnica y creativa del uso de SDF en entornos modernos.



# Development of a 3D modeling application based on Signed Distance Functions/Fields (SDF)

Pablo Cantudo Gomez

**Keywords:** WebGPU, C++, ray marching, rendering engine, Signed Distance Functions/Fields (SDF)

## Abstract

This work presents the development of Copper, a real-time 3D rendering engine implemented in C++ using the WebGPU API through Dawn. The engine employs ray marching techniques over Signed Distance Functions/Fields (SDF), enabling an efficient and flexible representation of three-dimensional geometry. One of the main advantages of using SDF is the ability to directly and compactly express Boolean operations between geometric primitives — such as unions, intersections, or differences— through simple algebraic expressions. Furthermore, unlike traditional mesh- or polygon-based methods, SDF allows for smooth Boolean operations, such as smooth union, in a virtually trivial manner from a computational standpoint.

This approach greatly simplifies the construction of complex shapes, avoiding typical problems in geometric computing such as handling vertices, normals, or complex topologies. It also facilitates the animation and transformation of objects through continuous functions. The engine includes a shading system in WGSL, with support for soft shadows, modular scene operations, object selection, and model loading and saving. The results show that the use of SDF not only offers a more elegant model for defining geometry, but also enables the creation of visually complex scenes with less code and greater graphical expressiveness. Overall, the system demonstrates the technical and creative feasibility of using SDF in modern environments.





---

Yo, **Pablo Cantudo Gómez**, alumno de la titulación Grado en ingeniería informática de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 78243264, 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.

Fdo: Pablo Cantudo Gómez

Granada a x de Septiembre de 2025.



---

D. **Tutores**, Profesores del Área de x del Departamento x de la Universidad de Granada.

**Informan:**

Que el presente trabajo, titulado ***Nombre***, ha sido realizado bajo su supervisión por **Tutores**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 11 de Noviembre de 2023.

**Los directores:**

**Tutores**



# Agradecimientos

Quiero expresar mi agradecimiento a todas las personas que han contribuido directa o indirectamente a la realización de este trabajo. En primer lugar, a mi tutor Juan Carlos Torres Cantero y cotutor Luis López Escudero por la ayuda durante el desarrollo y la corrección del proyecto, a mis amigos por las discusiones y las ideas, y a mi familia por su apoyo incondicional.

Finalmente, me gustaría mencionar a la comunidad de desarrolladores y recursos abiertos, especialmente los foros, artículos y proyectos relacionados con WebGPU, SDF y ray marching, cuya documentación y ejemplos han sido una fuente de aprendizaje invaluable.



# Índice general

<b>1. Introducción</b>	<b>19</b>
1.1. Motivación . . . . .	20
1.2. Descripción del problema . . . . .	20
1.3. Objetivos . . . . .	21
1.4. Estructura de la memoria . . . . .	22
<b>2. Fundamentos teóricos</b>	<b>23</b>
2.1. Modelado tridimensional: Paradigmas y fundamentos . . . . .	23
2.1.1. Ventajas de las SDF frente al modelado poligonal . . . . .	23
2.2. Funciones de distancia con signo (SDF) . . . . .	24
2.2.1. Propiedades y aplicaciones . . . . .	24
2.3. Ray Marching . . . . .	25
2.3.1. Comparación con Ray Tracing . . . . .	25
2.3.2. Sphere Tracing . . . . .	25
2.4. WebGPU: Acceso moderno a la GPU . . . . .	26
2.5. Shaders y lenguaje WGSL . . . . .	26
2.6. Herramientas auxiliares . . . . .	27
<b>3. Arquitectura del sistema</b>	<b>29</b>
3.1. Módulo Core . . . . .	29
3.2. Módulo Window . . . . .	29
3.3. Módulo Renderer . . . . .	30
3.4. Subsistema de Cámara . . . . .	30
3.5. Módulo Coder . . . . .	31
3.6. Módulo GizmoControls . . . . .	31
3.7. Interfaz gráfica (GUI) . . . . .	32
3.8. Flujo de datos e interacción . . . . .	32
3.9. Diagrama de arquitectura . . . . .	33
3.10. Justificación de la arquitectura . . . . .	33
<b>4. Implementación del módulo Ventana</b>	<b>35</b>
4.1. Introducción . . . . .	35
4.2. Diseño y estructura de la clase Window . . . . .	35

4.2.1. Atributos principales . . . . .	35
4.2.2. Métodos públicos . . . . .	35
4.2.3. Callbacks de eventos . . . . .	36
4.3. Proceso de inicialización y configuración . . . . .	36
4.4. Gestión avanzada de eventos . . . . .	37
4.5. Interacción con el ciclo de vida de la aplicación . . . . .	38
4.6. Decisiones técnicas y justificación . . . . .	38
4.7. Interacción con otros componentes . . . . .	38
4.8. Flujo de datos y ciclo de vida . . . . .	39
<b>5. Implementación y diseño de los controles</b>	<b>41</b>
5.1. Introducción . . . . .	41
5.2. Fundamentos matemáticos y uso de <code>glm</code> . . . . .	41
5.2.1. Transformaciones en el espacio 3D . . . . .	41
5.2.2. Rotaciones y cuaterniones . . . . .	42
5.2.3. Intersecciones y picking . . . . .	42
5.3. Controles de interfaz gráfica: <code>ImGui</code> . . . . .	42
5.3.1. Estructura y funciones principales . . . . .	42
5.3.2. Ejemplo de interacción . . . . .	43
5.4. Controles de gizmo: <code>GizmoControls</code> . . . . .	43
5.4.1. Arquitectura y flujo de interacción . . . . .	43
5.4.2. Matemáticas implicadas . . . . .	43
5.4.3. Sincronización con cámara y escena . . . . .	44
5.5. Controles de cámara: <code>CameraController</code> . . . . .	44
5.5.1. Estructura y flujo de eventos . . . . .	44
5.5.2. Matemáticas y código relevante . . . . .	44
5.5.3. Integración con el resto del sistema . . . . .	44
<b>6. Implementación: Shaders</b>	<b>45</b>
<b>7. Funcionalidades adicionales implementadas</b>	<b>47</b>
7.1. Introducción . . . . .	47
7.2. Gestión de la escena: guardado y carga . . . . .	47
7.2.1. Guardar escena . . . . .	47
7.2.2. Cargar escena . . . . .	47
7.2.3. Integración en la interfaz . . . . .	48
7.3. Gestión de la luz . . . . .	48
7.3.1. Control de la luz desde la GUI . . . . .	48
7.3.2. Actualización del pipeline . . . . .	48
7.4. Visualización y edición de objetos SDF . . . . .	48
7.4.1. Edición en tiempo real . . . . .	48
7.4.2. Eliminación y gestión de objetos . . . . .	49
7.5. Control de operaciones entre objetos . . . . .	49
7.5.1. Integración y propagación de cambios . . . . .	49



ÍNDICE GENERAL	17
7.6. Mostrar/ocultar el plano de referencia (floor) . . . . .	49
7.7. Visualización de FPS y métricas . . . . .	49
8. Pruebas	51
9. Conclusiones	53
Bibliografía	53
A. Código	55
B. Escenas	57
Glosario	57



# Índice de figuras

2.1. Representación visual del algoritmo de sphere tracing. . . . .	26
3.1. Arquitectura de Copper: relación entre módulos principales. .	33



# Capítulo 1

## Introducción

El desarrollo de los gráficos tridimensionales por computadora ha sido un campo de investigación activo y en constante evolución desde sus inicios. La capacidad de crear representaciones visuales de objetos y escenas en tres dimensiones ha revolucionado diversas industrias, desde el entretenimiento hasta la medicina y la ingeniería.

A medida que la tecnología avanza, también lo hacen las técnicas y herramientas utilizadas para generar gráficos 3D, lo que plantea nuevos desafíos y oportunidades para los investigadores y desarrolladores.

En sus inicios, la generación de gráficos 3D estaba limitada por la capacidad de cómputo y se basaba en *pipelines* gráficos fijos compuestos por etapas de transformación, iluminación y rasterización.

Posteriormente, con la llegada de los *shaders* programables en GPU (Nvidia GeForce 3 en 2001), fue posible sustituir los *pipelines* fijos por *pipelines* programables, lo que abrió un abanico de posibilidades para la creación de efectos visuales complejos y personalizados, como los algoritmos no basados en polígonos.

Esto impulsó la investigación en técnicas de representación más avanzadas, como el *ray tracing* y, en particular, el *ray marching*.

En el contexto del renderizado basado en funciones implícitas, las *Signed Distance Functions* (SDF) no son una invención reciente, sino que tienen sus raíces en trabajos mucho más antiguos.

El concepto de combinar funciones implícitas mediante operaciones booleanas se remonta al trabajo de Ricci en 1972 [Ric73], y fue ampliado en 1989 por B. Wyvill y G. Wyvill con el modelado de *soft objects* [WMW86].

Ese mismo año, Sandin, Hart y Kauffman aplicaron *ray marching* a SDF para renderizar fractales tridimensionales [HSK89].

Posteriormente, en 1995, Hart documentó de nuevo la técnica, a la que denominó *Sphere Tracing* [Har96].

La popularización moderna de las SDF en el ámbito del *renderizado en tiempo real* se debe en gran parte a la comunidad *demoscene*, especialmente

a partir de mediados de la década de 2000.

Trabajos como el de Crane (2005) y Evans (2006) introdujeron la idea de restringir el campo a una distancia euclidiana real, mejorando el rendimiento y la calidad visual.

Sin embargo, el uso de esta técnica no está tan extendido en aplicaciones de renderizado en tiempo real, a pesar de su potencial para crear gráficos visuales y eficientes.

## 1.1. Motivación

Como cualquier persona nacida en los dos mil, he crecido rodeado de videojuegos y el avance en la tecnología de gráficos 3D ha sido un aspecto fascinante de esta industria. Durante la carrera de informática, estudié diversas asignaturas relacionadas con gráficos por computadora, cuyos proyectos despertaron y consolidaron mi interés en este campo.

El desarrollo de motores gráficos y técnicas de renderizado siempre me ha resultado un área especialmente atractiva, no solo por su complejidad técnica, sino también por el impacto directo que tienen en sectores como el entretenimiento, la simulación o la realidad virtual. A lo largo de mis estudios me encontré con herramientas muy potentes, pero también con la dificultad que implica dominarlas o adaptarlas a entornos experimentales. Esto me llevó a plantearme la posibilidad de crear una aplicación propia que sirviera como espacio de exploración.

La motivación principal de este trabajo es profundizar en tecnologías emergentes, en concreto **WebGPU**, un estándar reciente que promete unificar el desarrollo gráfico multiplataforma con un acceso eficiente a las GPU modernas. Asimismo, me interesaba experimentar con el modelado mediante **funciones de distancia (SDF)**, que representan una alternativa flexible al modelado poligonal clásico. Considero que la combinación de ambas tecnologías constituye un terreno de investigación con un gran potencial, tanto en aplicaciones prácticas como en entornos educativos.

Finalmente, este proyecto me ofrece la oportunidad de afianzar mis conocimientos en programación gráfica, shaders y arquitecturas modernas de GPU, a la vez que desarrollo un software propio que pueda servir de base para futuros trabajos de investigación o aplicaciones más complejas en el ámbito del diseño 3D.

## 1.2. Descripción del problema

El campo del modelado y renderizado 3D ha estado tradicionalmente dominado por herramientas complejas y de gran envergadura, como Blender, Maya o 3ds Max. Si bien estas aplicaciones ofrecen una gran potencia

y versatilidad, presentan también limitaciones importantes: requieren elevados recursos de hardware, poseen curvas de aprendizaje pronunciadas y no siempre resultan adecuadas para entornos de experimentación ligera o proyectos educativos.

Por otro lado, las API gráficas más extendidas, como OpenGL o DirectX, han demostrado su eficacia a lo largo de los años, pero presentan restricciones en cuanto a eficiencia y portabilidad en plataformas modernas. El reciente estándar **WebGPU** surge como respuesta a estas carencias, ofreciendo un modelo de programación más cercano al hardware y multiplataforma, con el objetivo de unificar el desarrollo gráfico en navegadores y aplicaciones nativas.

En el ámbito del modelado, el paradigma poligonal sigue siendo el más utilizado, pero alternativas como las **funciones de distancia (SDF)** permiten representar geometrías complejas de manera más compacta y flexible, facilitando la combinación de primitivas y operaciones booleanas. Sin embargo, la integración de estas técnicas en aplicaciones prácticas todavía es limitada, especialmente en combinación con tecnologías emergentes como WebGPU.

El problema que aborda este trabajo consiste en la falta de herramientas ligeras que sirvan como demostración y entorno de experimentación para el modelado y renderizado basados en SDF sobre WebGPU.

## 1.3. Objetivos

### Objetivo general

El objetivo principal de este Trabajo de Fin de Grado es el desarrollo de una aplicación de diseño 3D basada en **WebGPU** y en técnicas de **modelado mediante funciones de distancia (SDF)**, que permita explorar y demostrar el potencial de estas tecnologías como alternativa al modelado poligonal clásico y como herramienta de experimentación en el ámbito de los gráficos por computadora.

### Objetivos específicos

Para alcanzar este objetivo general, se plantean los siguientes objetivos específicos:

- Investigar y comprender en profundidad el funcionamiento de la API WebGPU y su integración en aplicaciones nativas mediante la librería Dawn.
- Diseñar e implementar un motor de renderizado basado en *ray marching* sobre funciones de distancia, capaz de representar primitivas y combinaciones mediante operaciones booleanas y suaves.

- Incorporar técnicas de sombreado y efectos visuales (iluminación, sombras suaves) que mejoren la calidad del renderizado.
- Desarrollar una interfaz gráfica sencilla que permita al usuario interactuar con la escena y manipular las primitivas.

## 1.4. Estructura de la memoria

La presente memoria se organiza en los siguientes capítulos:

- **Introducción:** Se expone el contexto del trabajo, la motivación, los objetivos planteados y la justificación de la elección de las tecnologías empleadas.
- **Fundamentos teóricos:** Se revisan los conceptos clave de modelado y renderizado 3D, las funciones de distancia (*Signed Distance Functions*, *SDF*) y el estándar WebGPU, contextualizando el trabajo en el estado actual de la tecnología.
- **Arquitectura de la aplicación:** Se describe la estructura general de la aplicación Copper, detallando los principales módulos, componentes y la interacción entre ellos.
- **Implementación:** Este capítulo se divide en varias secciones:
  - **Ventana y motor de renderizado:** Se explica el proceso de inicialización de la ventana, integración con Dawn/WebGPU y el desarrollo del motor de renderizado basado en SDF y ray marching.
  - **Controles e interacción:** Se detalla la implementación de los controles de cámara y de manipulación de objetos mediante la interfaz gráfica.
  - **Shaders y efectos visuales:** Se describe el desarrollo de los shaders WGSL utilizados, incluyendo técnicas de sombreado, iluminación y efectos visuales implementados.
- **Pruebas y validación:** Se presentan las pruebas realizadas para evaluar el rendimiento, la estabilidad y la calidad visual de la aplicación, así como los resultados obtenidos.
- **Conclusiones y trabajos futuros:** Se realiza un balance del trabajo realizado, se revisa el grado de cumplimiento de los objetivos y se plantean posibles líneas de investigación y desarrollo futuras.
- **Bibliografía y anexos:** Se recopilan las referencias bibliográficas consultadas y se incluyen materiales complementarios relevantes para la comprensión y reproducibilidad del trabajo.



## Capítulo 2

# Fundamentos teóricos

El modelado y renderizado en 3D es posible gracias a una serie de técnicas matemáticas y computacionales que permiten representar, manipular y visualizar geometría en entornos virtuales. Copper se apoya principalmente en el modelado mediante funciones de distancia con signo (SDF), el renderizado por ray marching y el uso del estándar gráfico WebGPU. Este capítulo describe en profundidad cada uno de estos fundamentos.

### 2.1. Modelado tridimensional: Paradigmas y fundamentos

El modelado tridimensional tradicional utiliza mallas poligonales, donde los objetos se representan mediante vértices, aristas y caras conectadas entre sí. Este método, empleado en la mayoría de herramientas profesionales (*Blender*, *Maya*, *3ds Max*), permite una gran flexibilidad, pero implica gestionar topología y almacenar grandes cantidades de datos, lo que complica la edición y la generación procedural.

Como alternativa, existen métodos de representación implícita, siendo los campos de distancia con signo (*Signed Distance Fields*, *SDF*) los más destacados. Una SDF es una función  $f(\vec{x})$  que, para cada punto  $\vec{x}$  del espacio, devuelve la distancia mínima a la superficie del objeto. El signo indica si el punto está en el interior (negativo), sobre la superficie (cero) o en el exterior (positivo). Este enfoque permite describir objetos mediante expresiones matemáticas, simplificando la combinación y manipulación de geometría compleja.

#### 2.1.1. Ventajas de las SDF frente al modelado poligonal

- **Compacidad:** Las SDF están definidas por fórmulas matemáticas, no por listas de vértices, lo que reduce el espacio necesario para describir objetos.

- **Facilidad de combinación:** Pueden combinarse mediante operadores matemáticos (unión, intersección, resta) de forma eficiente.
- **Transformaciones geométricas:** Admiten fácilmente traslaciones, rotaciones y escalados aplicando transformaciones sobre la función.
- **Cálculo de normales:** La normal en un punto de la superficie se obtiene como el gradiente de la función de distancia.
- **Flexibilidad:** Permiten crear transiciones suaves entre objetos mediante operadores suavizados.

## 2.2. Funciones de distancia con signo (SDF)

Las SDF asignan a cada punto del espacio la distancia mínima a una superficie implícita. Formalmente, para una función  $f(\vec{x})$ , la superficie se define como el conjunto de puntos donde  $f(\vec{x}) = 0$ . Las SDF permiten describir primitivas básicas como:

- **Esfera:**  $f_{esfera}(\vec{x}) = ||\vec{x} - \vec{c}|| - r$ , donde  $\vec{c}$  es el centro y  $r$  el radio.
- **Caja:**  $f_{caja}(\vec{x}) = ||\max(|\vec{x} - \vec{c}| - \vec{s}, 0)|| + \min(\max(d_x, \max(d_y, d_z)), 0)$ , donde  $\vec{s}$  es el tamaño.

Las SDF pueden combinarse empleando operadores booleanos y suaves:

- **Unión:**  $f_{union}(a, b) = \min(a, b)$
- **Intersección:**  $f_{inter}(a, b) = \max(a, b)$
- **Resta:**  $f_{resta}(a, b) = \max(a, -b)$
- **Unión suave:** Interpolación entre distancias y colores para crear transiciones continuas.

### 2.2.1. Propiedades y aplicaciones

Las SDF permiten:

- Evaluar la función en cualquier punto del espacio para detectar colisiones, calcular iluminación o generar geometría procedural.
- Calcular la normal en un punto como el gradiente  $\nabla f(\vec{x})$ .
- Construir geometría fractal y orgánica mediante funciones recursivas o combinaciones arbitrarias.

Se emplean en renderizado, simulación física, generación de terrenos y efectos visuales avanzados.

## 2.3. Ray Marching

El *ray marching* es una técnica de renderizado que, utilizando la SDF, avanza iterativamente un rayo en el espacio hasta aproximar la intersección con una superficie implícita.

El algoritmo sigue estos pasos:

1. Lanzar un rayo desde la cámara en una dirección determinada.
2. Evaluar la SDF en la posición actual para obtener la distancia mínima a la superficie más cercana.
3. Avanzar el punto a lo largo del rayo una distancia igual al valor obtenido.
4. Repetir hasta que la distancia sea menor que un umbral (colisión con la superficie) o se alcance un límite de pasos o distancia máxima.

### 2.3.1. Comparación con Ray Tracing

- **Ray tracing:** Calcula la intersección exacta con primitivas geométricas.
- **Ray marching:** Utiliza la distancia local para aproximar superficies implícitas y fractales.

Ray marching facilita la incorporación de sombras suaves, reflexión y refracción, y la renderización eficiente de geometría compleja.

### 2.3.2. Sphere Tracing

La técnica de **sphere tracing** es una variante eficiente del ray marching, utilizada específicamente para el renderizado de superficies implícitas definidas por funciones de distancia con signo (SDF). Su principal ventaja respecto a los métodos tradicionales de ray casting es que utiliza la propia SDF para calcular el avance óptimo en cada paso del rayo, evitando colisiones innecesarias y acelerando la detección de intersecciones.

El algoritmo se basa en la siguiente idea: desde el origen del rayo, en cada iteración se evalúa la SDF en la posición actual para obtener la distancia mínima a cualquier superficie. Este valor se interpreta como el radio de una esfera libre de obstáculos centrada en el punto actual. Así, se puede avanzar el rayo exactamente esa distancia sin riesgo de atravesar ninguna superficie. El proceso se repite hasta que la distancia es menor que un umbral prefijado (lo que indica que se ha alcanzado la superficie) o hasta que se supera una distancia máxima o número de pasos.

Sphere tracing es especialmente útil en escenas donde las funciones de distancia son suaves y bien definidas, y permite renderizar geometría compleja con costes computacionales bajos. Sin embargo, puede resultar menos eficiente en casos de superficies muy delgadas o SDFs poco continuas, donde el avance óptimo se reduce drásticamente.

La Figura 2.1 ilustra el funcionamiento del algoritmo: cada círculo representa el rango libre de obstáculos desde la posición actual del rayo, determinado por la evaluación de la SDF. El rayo avanza saltando de esfera hasta que la distancia mínima indica una colisión con la superficie.

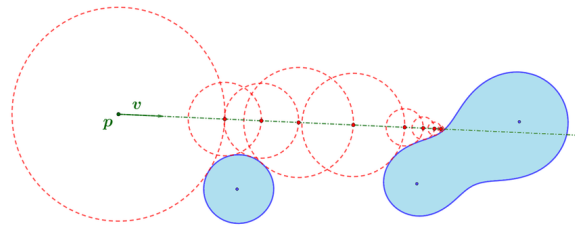


Figura 2.1: Representación visual del algoritmo de sphere tracing.

## 2.4. WebGPU: Acceso moderno a la GPU

**WebGPU** es un estándar gráfico de nueva generación que proporciona acceso eficiente y multiplataforma a la GPU, tanto en navegadores como en aplicaciones nativas. Sus principales características incluyen:

- Multiplataforma: Compatible con Windows, Linux, Mac y navegadores recientes.
- Eficiencia: Modelo de programación cercano al hardware, acceso directo a buffers y pipelines.
- Shaders en WGSL: Permite la programación de efectos personalizados y operaciones matemáticas avanzadas.

WebGPU ofrece ventajas frente a OpenGL y DirectX en control de recursos, computación general y portabilidad.

## 2.5. Shaders y lenguaje WGSL

Los **shaders** son programas que ejecutan operaciones matemáticas en la GPU para transformar vértices, calcular colores y simular efectos visuales.

WGSL (*WebGPU Shading Language*) es el lenguaje nativo de WebGPU, diseñado para expresar funciones de distancia, operadores booleanos, cálculos de iluminación y efectos visuales de forma eficiente.

- **Vertex shaders:** Transforman posiciones y atributos de vértices.
- **Fragment shaders:** Calculan el color final de cada píxel, aplicando modelos de iluminación como Blinn-Phong, efectos de sombras y combinaciones de SDF.

WGSL permite aprovechar la arquitectura de la GPU para realizar renderizado en tiempo real.

## 2.6. Herramientas auxiliares

El desarrollo de aplicaciones gráficas requiere gestionar ventanas, entrada de usuario y la interfaz gráfica:

- **GLFW:** Biblioteca multiplataforma para la gestión de ventanas y eventos.
- **ImGui:** Sistema de interfaz gráfica inmediata para la manipulación interactiva de primitivas y parámetros de escena.
- **GLM:** Biblioteca matemática para operaciones con vectores y matrices.
- **CMake:** Herramienta de compilación y gestión de dependencias.

Estas herramientas proporcionan la infraestructura básica para la interacción y visualización dentro del entorno de Copper.



## Capítulo 3

# Arquitectura del sistema

La arquitectura de **Copper** ha sido concebida siguiendo principios de modularidad, mantenibilidad y claridad estructural. A continuación se detallan los diferentes módulos que componen el sistema, describiendo su funcionalidad y las interacciones que mantienen entre sí, con referencias directas al código fuente.

### 3.1. Módulo Core

El módulo **Core** (`src/core/Core.cpp`, `src/core/Core.h`) es el responsable de orquestrar el ciclo de vida de la aplicación. Sus funciones principales incluyen:

- **Inicialización:** Llama a los métodos de inicialización de los módulos **Window** y **Renderer** para preparar el entorno gráfico y la ventana principal.
- **Bucle principal:** Ejecuta el método `MainLoop()`, que gestiona los eventos y delega el renderizado de la escena.
- **Gestión de estado:** Permite saber si la aplicación sigue ejecutándose (`IsRunning()`) y controla el cierre ordenado de recursos (`Terminate()`).

### 3.2. Módulo Window

El módulo **Window** (`src/ui/Window.cpp`, `src/ui/Window.h`) se encarga de la creación y gestión de la ventana principal de la aplicación, utilizando GLFW. Sus funciones incluyen:

- **Configuración de la ventana:** Establece parámetros como el tamaño, aspecto y modo de interacción.

- **Gestión de eventos:** Implementa callbacks para eventos de redimensionado y de ratón, que se propagan al módulo **Renderer**.
- **Interfaz con el sistema:** Abstrae la interacción con el sistema operativo y facilita la obtención de dimensiones y el manejo del ciclo de vida de la ventana.

Esto permite que la ventana sea independiente del motor de renderizado y fácil de modificar o ampliar.

### 3.3. Módulo Renderer

El **Renderer** (`src/core/Renderer.cpp`, `src/core/Renderer.h`) es el motor gráfico del sistema. Sus principales responsabilidades son:

- **Inicialización de gráficos:** Configura el contexto de WebGPU, las superficies y los buffers necesarios para el renderizado.
- **Ciclo de renderizado:** Gestiona el bucle de renderizado, actualizando los datos de la escena y llamando a la función `Render()` en cada frame.
- **Gestión de uniforms:** Actualiza los uniforms (matriz MVP, posición de luz, tamaño de ventana, etc.) y los envía al shader.
- **Coordinación de módulos:** Instancia y gestiona los módulos **Camera**, **CameraController**, **GizmoControls**, **Coder** e **Interfaz** (GUI), asegurando la comunicación entre ellos.
- **Gestión de interacción:** Recoge eventos de usuario (ratón, teclado) y los distribuye entre los controles de cámara, gizmo y selección de objetos.

Además, el renderer es responsable de decidir cuándo se debe actualizar el pipeline gráfico, por ejemplo, al modificar objetos SDF o parámetros visuales.

### 3.4. Subsistema de Cámara

**Camera** y **CameraController** (`src/core/Camera.cpp`, `src/core/Camera.h`, `src/core/CameraController.cpp`, `src/core/CameraController.h`) forman el subsistema encargado de gestionar la vista de la escena 3D.

- **Camera:** Mantiene el estado (posición, centro, orientación) y permite obtener la matriz de vista y los vectores de orientación.



- **CameraController:** Implementa la lógica de interacción con el usuario (rotaciones, traslaciones, zoom) mediante eventos de ratón y teclado, actualizando la posición y orientación de la cámara en tiempo real.
- **Integración:** La cámara es utilizada tanto por el **Renderer** como por las herramientas de manipulación de objetos (**GizmoControls**).

### 3.5. Módulo Coder

El módulo **Coder** (`src/core/Coder.cpp`, `src/core/Coder.h`) se especializa en la generación dinámica de código shader para el renderizado de objetos SDF. Sus funciones incluyen:

- **Gestión de objetos:** Permite añadir, modificar y eliminar objetos SDF (esferas, cajas, conos, cilindros) y operaciones entre ellos (unión, intersección, sustracción).
- **Generación de shaders:** Construye el código shader que será utilizado en WebGPU para renderizar la escena y para operaciones de picking.
- **Selección y edición:** Gestiona el objeto actualmente seleccionado y expone métodos para editar sus propiedades y operación.
- **Serialización:** Permite guardar y cargar escenas desde fichero, facilitando la persistencia y el intercambio de datos.

### 3.6. Módulo GizmoControls

El módulo **GizmoControls** (`src/core/GizmoControls.cpp`, `src/core/GizmoControls.h`) proporciona herramientas para la manipulación visual e interactiva de los objetos SDF en la escena. Sus responsabilidades son:

- **Picking y manipulación:** Permite seleccionar e interactuar con los objetos mediante gizmos (ejes, planos de traslación).
- **Sincronización:** Mantiene la coherencia entre la posición del objeto, la cámara y el estado de la interacción.
- **Integración con el renderizado:** Utiliza la cámara y los datos de la escena para calcular las transformaciones y la interacción del usuario.

### 3.7. Interfaz gráfica (GUI)

El módulo **Interfaz** (`src/ui/Interfaz.cpp`, `src/ui/Interfaz.h`) implementa la interfaz gráfica de usuario utilizando ImGui. Sus funciones principales son:

- **Edición de objetos:** Permite modificar las propiedades de los objetos seleccionados (posición, tamaño, color, tipo, operación).
- **Gestión de la escena:** Ofrece controles para añadir, eliminar y editar objetos, así como para guardar y cargar escenas.
- **Control de parámetros globales:** Permite ajustar parámetros de iluminación, visualización y opciones de renderizado.
- **Comunicación:** Interactúa con los módulos **Coder** y **Renderer** para aplicar los cambios en tiempo real.

### 3.8. Flujo de datos e interacción

El flujo de datos e interacción entre módulos se organiza de la siguiente manera:

- **Core** inicializa **Window** y **Renderer**.
- **Renderer** instancia y coordina **Camera**, **CameraController**, **GizmoControls** y **Coder**.
- **CameraController** y **GizmoControls** gestionan la interacción directa del usuario.
- **Interfaz** permite la edición visual y comunica los cambios a **Coder** y **Renderer**.
- **Coder** actualiza el shader y los objetos SDF, transmitiendo la información al **renderer** para el ciclo de renderizado.

### 3.9. Diagrama de arquitectura

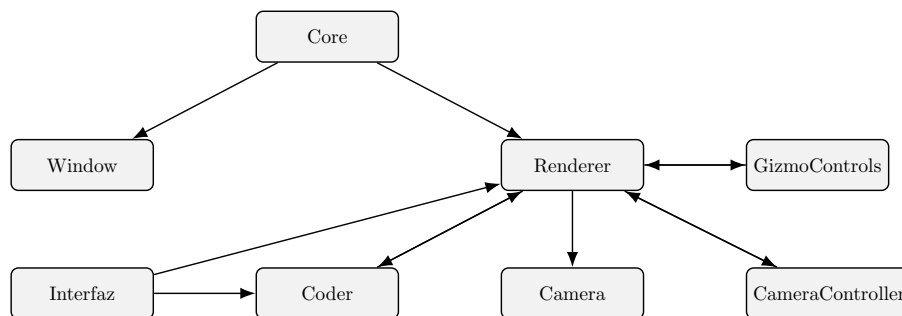


Figura 3.1: Arquitectura de Copper: relación entre módulos principales.

### 3.10. Justificación de la arquitectura

La estructura modular y jerárquica implementada en Copper permite separar responsabilidades, facilitar el mantenimiento y la escalabilidad del sistema. Las decisiones de diseño han sido tomadas para maximizar la claridad, la reutilización y la posibilidad de extensión futura del sistema de acuerdo a los siguientes criterios:

- **Separación de responsabilidades:** El diseño modular permite que cada componente (núcleo, renderizado, cámara, gestión de objetos, manipulación, interfaz) se encargue de una función específica, lo que facilita la comprensión, el mantenimiento y la evolución del sistema. Por ejemplo, **Core** sólo gestiona el ciclo de vida y delega el renderizado a **Renderer**, mientras que la creación y edición de objetos SDF queda aislada en **Coder**.
- **Facilidad de extensión:** La existencia de clases independientes para cada tipo de objeto (**sphere**, **box**, **cone**, **cylinder**) y para operaciones (**union**, **subtract**, **intersection**, etc.) en **Coder** permite añadir nuevos tipos u operaciones sin modificar el núcleo del sistema, como se observa en los métodos **addSphere**, **addBox**, etc. [`src/core/Coder.cpp`, `src/core/Coder.h`].
- **Interactividad y desacoplamiento:** Los subsistemas de cámara y control de objetos (**Camera/CameraController**, **GizmoControls**) están desacoplados de la lógica de renderizado, permitiendo que la interacción del usuario se procese de forma independiente y se integre fácilmente con la interfaz gráfica (**Interfaz.cpp**). Esto se observa en la gestión de eventos de ratón y teclado, donde cada módulo recibe únicamente los datos que necesita y actualiza su estado local.

- **Actualización eficiente del renderizado:** El sistema de notificación de cambios (`pipelineDirty` en `Renderer`, activado por modificaciones en `Coder` o en la escena) permite que el pipeline gráfico se actualice sólo cuando es necesario, optimizando el rendimiento y evitando cálculos redundantes [`src/core/Renderer.cpp`].

## Capítulo 4

# Implementación del módulo Ventana

### 4.1. Introducción

El módulo **Ventana** constituye el punto de entrada visual y de interacción para el sistema Copper. Este componente es responsable de la creación, gestión y destrucción de la ventana principal de la aplicación, así como del manejo de eventos de usuario y del ciclo de vida asociado a la interfaz gráfica.

### 4.2. Diseño y estructura de la clase Window

La clase `Window`, ubicada en `src/ui/Window.h` y `src/ui/Window.cpp`, encapsula toda la funcionalidad relacionada con la ventana.

#### 4.2.1. Atributos principales

- `GLFWwindow* window`: Puntero al objeto de ventana gestionado por GLFW.
- `uint32_t windowHeight, windowWidth`: Dimensiones internas de la ventana, actualizadas dinámicamente.

#### 4.2.2. Métodos públicos

- `bool Initialize(Renderer *renderer)`: Inicializa la ventana, configurando parámetros gráficos y de interacción, y registra los callbacks de eventos. La ventana se crea con una relación de aspecto fija (16:9), facilitando la integración con el pipeline gráfico y evitando distorsiones.
- `GLFWwindow* getWindow()`: Permite acceder al objeto de ventana desde otros módulos, especialmente el `Renderer` y la interfaz gráfica.

- `uint32_t getWindowWidth(), getWindowHeight()`: Proporcionan las dimensiones actuales, utilizadas en la configuración de la proyección y el cálculo del aspecto.
- `void setWindowSize(uint32_t width, uint32_t height)`: Permite modificar las dimensiones internas de la ventana, útil en escenarios de redimensionado.
- `void Destroy()`: Libera los recursos asociados a la ventana, llamando a `glfwDestroyWindow`.
- `void SetWindowUserPointer(void* pointer)`: Permite asociar un puntero de usuario (normalmente, el `Renderer`) para su acceso desde los callbacks de eventos.

#### 4.2.3. Callbacks de eventos

La clase `Window` define y registra varios callbacks estáticos:

- `FramebufferSizeCallback(GLFWwindow* window, int width, int height)`: Se ejecuta cuando la ventana es redimensionada. Utiliza el puntero de usuario para acceder al `Renderer` y llama a su método `OnResize()`, lo que desencadena la reconfiguración de los buffers gráficos y la superficie de renderizado. Esto garantiza que el aspecto visual de la escena se mantenga correcto ante cualquier cambio de tamaño.
- `MouseButtonCallback(GLFWwindow *window, int button, int action, int mods)`: Captura los eventos de pulsación de botones de ratón. Si la interfaz gráfica (`ImGui`) no está capturando el ratón, el evento se propaga al método `OnMouseButton` del `Renderer`, permitiendo la interacción directa con los objetos 3D a partir de los gizmos.

El registro de los callbacks se realiza en el método `Initialize`, asegurando que todos los eventos relevantes sean gestionados desde el inicio de la aplicación.

### 4.3. Proceso de inicialización y configuración

El proceso de inicialización de la ventana sigue una secuencia de pasos bien definida:

1. Llamada a `glfwInit()` para inicializar la biblioteca GLFW.
2. Configuración de los hints de ventana (`GLFW_CLIENT_API`, `GLFW_RESIZABLE`) para deshabilitar la API gráfica por defecto y permitir el redimensionado.

3. Obtención del monitor y modo de vídeo principal para adaptar la ventana al entorno gráfico del usuario.
4. Creación de la ventana con `glfwCreateWindow`, ajustando el tamaño al modo de vídeo y estableciendo el título (`Çopper`).
5. Fijación de la relación de aspecto mediante `glfwSetWindowAspectRatio`, asegurando que la escena 3D se muestre correctamente.
6. Registro de los callbacks de eventos (`FramebufferSizeCallback`, `MouseButtonCallback`).
7. Configuración de los modos de entrada (`GLFW_CURSOR`, `GLFW_RAW_MOUSE_MOTION`) para gestionar el cursor y la interacción avanzada.

Este diseño asegura que la ventana esté lista para recibir eventos y para integrarse con el sistema de renderizado desde el primer momento.

## 4.4. Gestión avanzada de eventos

El manejo de eventos en la ventana es fundamental para la interactividad del sistema. El diseño implementado permite:

- **Sincronización con el pipeline gráfico:** Los eventos de redimensionado se propagan automáticamente al `Renderer`, que actualiza buffers, superficie y matrices de proyección, manteniendo la coherencia visual.
- **Integración con ImGui:** Antes de procesar eventos de ratón, se comprueba si ImGui está capturando el ratón, evitando conflictos entre la interfaz gráfica y la manipulación 3D.
- **Extensibilidad:** Aunque el callback de movimiento de ratón (`MouseMoveCallback`) está declarado, no se encuentra activado por defecto, permitiendo su futura ampliación sin modificar la estructura principal.

El siguiente fragmento de código muestra la gestión y propagación de eventos (`Window.cpp`):

```
void Window::MouseButtonCallback(GLFWwindow *window, int button, int action, int mods) {
    auto* renderer = static_cast<Renderer*>(glfwGetWindowUserPointer(window));
    if (renderer != nullptr && !ImGui::GetIO().WantCaptureMouse) {
        renderer->OnMouseButton(button, action);
    }
}
```

## 4.5. Interacción con el ciclo de vida de la aplicación

La ventana se integra en el ciclo de vida de la aplicación a través del módulo `Core` (`src/core/Core.cpp`, `src/core/Core.h`), que gestiona su inicialización y destrucción. El flujo es el siguiente:

1. `Core::Initialize()` llama a `window.Initialize(renderer)` y a `renderer.Init(window)` asegurando que la ventana y el sistema gráfico están listos antes de iniciar el bucle principal.
2. Durante la ejecución, `Core::MainLoop()` llama a `glfwPollEvents()` y a `renderer.Render()`, manteniendo la actualización continua de la ventana y la escena.
3. Al finalizar, `Core::Terminate()` invoca `window.Destroy()` y `glfwTerminate()`, liberando todos los recursos y evitando fugas de memoria.

## 4.6. Decisiones técnicas y justificación

La elección de GLFW como biblioteca para la gestión de ventanas se justifica por varios motivos:

- **Portabilidad:** GLFW es multiplataforma, permitiendo ejecutar Copper en diferentes sistemas operativos sin cambios en el código.
- **Compatibilidad:** La integración con WebGPU y con ImGui está ampliamente soportada, simplificando el desarrollo y la depuración.
- **Modularidad:** El diseño de la clase `Window` permite modificar el backend de ventanas sin afectar al resto del sistema, siguiendo el principio de separación de responsabilidades.
- **Eficiencia:** La gestión nativa de eventos y la actualización dinámica de los buffers gráficos optimizan el rendimiento y la fluidez de la aplicación.

Estas decisiones están alineadas con las recomendaciones de la literatura técnica (Sommerville, 2016) y con las guías docentes de la ETSIT, que aconsejan implementar componentes modulares, eficientes y fácilmente mantenibles.

## 4.7. Interacción con otros componentes

El módulo Ventana está estrechamente integrado con:



- **Renderer:** Recibe y procesa los eventos relevantes para actualizar el pipeline gráfico y la escena.
- **Interfaz gráfica (GUI):** Sirve como base para la integración de ImGui, permitiendo la edición y visualización de parámetros y objetos.
- **Subsistema de cámara y controles:** Los eventos de entrada capturados por la ventana se transmiten al controlador de cámara y gizmos, facilitando la manipulación interactiva de la escena.

La arquitectura facilita la extensión futura del sistema, permitiendo añadir nuevas funcionalidades (por ejemplo, soporte para pantallas múltiples o diferentes sistemas de entrada) sin modificar la lógica principal.

## 4.8. Flujo de datos y ciclo de vida

Durante la ejecución de la aplicación, la ventana realiza las siguientes tareas:

- Recoge eventos de entrada (ratón, teclado, redimensionado) y los distribuye entre los módulos correspondientes.
- Mantiene actualizadas las dimensiones de la ventana, permitiendo al renderer ajustar el aspecto y las matrices de proyección.
- Facilita la integración de la interfaz gráfica y la visualización de los controles de usuario.

El ciclo de vida completo está gestionado por el módulo **Core**, garantizando la correcta inicialización y liberación de recursos.



## Capítulo 5

# Implementación y diseño de los controles

### 5.1. Introducción

El sistema de controles en Copper permite la manipulación de la escena 3D y la interacción directa con los objetos SDF (Signed Distance Fields). Este capítulo describe en profundidad la arquitectura, matemáticas y tecnologías utilizadas (especialmente la librería `glm`), y se divide en tres grandes apartados: controles de interfaz gráfica (ImGui), controles de gizmo (GizmoControls) y controles de cámara (CameraController).

### 5.2. Fundamentos matemáticos y uso de `glm`

Copper utiliza `glm` (OpenGL Mathematics) como librería principal para el manejo de vectores, matrices y cuaterniones, esenciales en la transformación y manipulación de objetos 3D.

#### 5.2.1. Transformaciones en el espacio 3D

En los archivos `Camera.cpp`, `CameraController.cpp` y `GizmoControls.cpp`, se emplea `glm` para:

- Representar posiciones y direcciones mediante `glm::vec3`.
- Calcular matrices de vista (`glm::mat4`), rotaciones con cuaterniones (`glm::quat`), y transformaciones de objetos y cámara.
- Realizar operaciones de producto escalar, normalización, traslaciones y rotaciones.

Por ejemplo, la actualización de la matriz de vista en `Camera.cpp` se realiza con:

```
this->view_matrix = glm::lookAt(this->eye, this->center, this->up);
```

Esto aplica la transformación de cámara estándar, permitiendo la navegación y manipulación de la escena.

### 5.2.2. Rotaciones y cuaterniones

El manejo de rotaciones evita el problema de gimbal lock y permite una navegación suave. En `CameraController.cpp`, se construyen cuaterniones para rotar la cámara y los objetos:

```
glm::quat rot_matrix_y = glm::angleAxis(this->vert_angle, glm::vec3(1, 0, 0));  
glm::quat rot_matrix_x = glm::angleAxis(this->horiz_angle, glm::vec3(0, 1, 0));  
glm::quat q = (rot_matrix_y * rot_matrix_x);
```

La matriz de vista se actualiza aplicando estas rotaciones, y las transformaciones se propagan a los controles de gizmo y objetos.

### 5.2.3. Intersecciones y picking

Para la manipulación mediante gizmos y picking de objetos, se emplean cálculos geométricos como la intersección de líneas y planos (`planeLineIntersection`) y el cálculo de rayos desde la cámara usando coordenadas UV normalizadas.

## 5.3. Controles de interfaz gráfica: ImGui

El módulo de interfaz gráfica (`Interfaz.cpp`, `Interfaz.h`) utiliza **ImGui** para implementar menús, sliders y herramientas de edición.

### 5.3.1. Estructura y funciones principales

La clase `Interfaz` interactúa con los módulos `Coder` y `Renderer` para:

- Mostrar propiedades de objetos seleccionados (posición, color, tipo, tamaño).
- Permitir la edición directa mediante sliders y campos de entrada.
- Gestionar la creación, edición y borrado de objetos SDF.
- Proporcionar controles globales de la escena (luz, renderizado, operaciones).
- Implementar la gestión de archivos para guardar y cargar escenas.

### 5.3.2. Ejemplo de interacción

Al seleccionar un objeto, ImGui presenta sus propiedades y permite modificarlas:

```
ImGui::SliderFloat("X", &selectedObject.x, -10.0f, 10.0f);
ImGui::ColorEdit3("Color", &selectedObject.r);
// Para esferas:
ImGui::SliderFloat("Radius", &selectedObject.size[0], 0.1f, 5.0f);
```

Cada cambio marca el pipeline como "dirty" para que el renderizador actualice la escena en tiempo real.

## 5.4. Controles de gizmo: GizmoControls

El módulo `GizmoControls` (`GizmoControls.cpp`, `GizmoControls.h`) permite la manipulación visual e interactiva de los objetos SDF seleccionados mediante gizmos (flechas, planos).

### 5.4.1. Arquitectura y flujo de interacción

- **Inicialización:** Al seleccionar un objeto y pulsar sobre el gizmo, se inicia la manipulación (`initDrag`).
- **Picking:** Se determina qué parte del gizmo ha sido seleccionada (eje, plano) usando ray marching y cálculos de distancia mínimos.
- **Arrastre y movimiento:** El punto de intersección inicial se calcula con `gizmoIntersection` y se actualiza en tiempo real mientras el usuario mueve el ratón.
- **Actualización de propiedades:** El centro del objeto se actualiza en `update`, propagando el nuevo valor al objeto seleccionado en el módulo `Coder`.

### 5.4.2. Matemáticas implicadas

Se emplean funciones de distancia (SDF) para flechas y planos, y transformaciones con matrices y cuaterniones para rotar y trasladar los gizmos respecto al objeto y la cámara. Ejemplo de cálculo de intersección de un rayo con un plano:

```
float d = glm::dot(planeNormal, planePoint - linePoint) / glm::dot(planeNormal, lineDirection);
return linePoint + d * lineDirection;
```

### 5.4.3. Sincronización con cámara y escena

El gizmo se actualiza en función de la cámara activa y el aspecto de la ventana, asegurando coherencia entre la visualización y la interacción.

## 5.5. Controles de cámara: CameraController

El módulo `CameraController` (`CameraController.cpp`, `CameraController.h`) gestiona la navegación de la cámara en la escena, aplicando transformaciones mediante eventos de ratón y teclado.

### 5.5.1. Estructura y flujo de eventos

- **Zoom:** El scroll del ratón modifica el radio de la cámara alrededor del centro, actualizando la vista.
- **Rotación:** El arrastre con el botón izquierdo aplica rotaciones sobre los ángulos vertical y horizontal, usando cuaterniones para evitar gimbal lock.
- **Traslación:** El arrastre con el botón derecho permite mover el centro de la cámara en el plano.
- **Gestión de estados:** El controlador mantiene el estado del ratón y actualiza la cámara solo cuando es necesario, evitando interferencias con la interfaz gráfica.

### 5.5.2. Matemáticas y código relevante

Las transformaciones se basan en las funciones de `glm` para matrices y cuaterniones. Ejemplo de rotación acumulada:

```
glm::quat rot_matrix_y = glm::angleAxis(this->vert_angle, glm::vec3(1, 0, 0));  
glm::quat rot_matrix_x = glm::angleAxis(this->horiz_angle, glm::vec3(0, 1, 0));  
glm::quat q = (rot_matrix_y * rot_matrix_x);  
this->total_rotation = q;  
this->update_view_matrix();
```

La matriz de vista se recalcula en función de la posición y orientación deseadas.

### 5.5.3. Integración con el resto del sistema

El controlador de cámara está sincronizado con la ventana principal y el renderizador, recibiendo eventos de entrada (mouse, scroll) y actualizando la matriz de vista que se utiliza en el ciclo de renderizado y en la manipulación de gizmos.

## Capítulo 6

# Implementación: Shaders





## Capítulo 7

# Funcionalidades adicionales implementadas

### 7.1. Introducción

Además de las capacidades principales de modelado y renderizado SDF, Copper incorpora un conjunto de funcionalidades adicionales que mejoran la experiencia de usuario, la usabilidad y la flexibilidad del sistema. En este capítulo se describen y documentan con detalle las funcionalidades que han sido añadidas sobre la base del código existente, su integración en los distintos módulos y su justificación técnica.

### 7.2. Gestión de la escena: guardado y carga

La posibilidad de guardar y cargar escenas permite al usuario almacenar el estado actual del modelado y recuperarlo posteriormente. Esta funcionalidad está implementada en el módulo `Coder`, mediante los métodos `saveScene` y `loadScene` (`src/core/Coder.cpp`, `src/core/Coder.h`).

#### 7.2.1. Guardar escena

El método `saveScene(const std::string& filename)` serializa todos los objetos presentes en la escena, incluyendo tipo, posición, tamaño, color, operación y el identificador único. El formato del archivo es texto plano y sigue una estructura legible, facilitando la interoperabilidad y la depuración.

#### 7.2.2. Cargar escena

El método `loadScene(const std::string& filename)` permite restaurar la escena a partir de un archivo previamente guardado. El sistema parsea cada línea, reconstruye los objetos y actualiza su identificador, asegurando la coherencia y la compatibilidad con versiones futuras.

### 7.2.3. Integración en la interfaz

La gestión de archivos está directamente integrada en la interfaz gráfica (`src/ui/Interfaz.cpp`), mediante el uso de `ImGuiFileDialog`. El usuario puede seleccionar el archivo deseado para guardar o cargar la escena, y el sistema actualiza la visualización en tiempo real. Este flujo está soportado por el siguiente fragmento:

```
if (ImGuiFileDialog::Instance()->IsOk()) {  
    std::string filePath = ImGuiFileDialog::Instance()->GetFilePathName();  
    coder->saveScene(filePath);  
}
```

## 7.3. Gestión de la luz

Copper permite modificar la posición de la fuente de luz principal en la escena. Esta funcionalidad se encuentra en la interfaz gráfica y en el módulo `Renderer` (`src/core/Renderer.cpp`, `src/core/Renderer.h`).

### 7.3.1. Control de la luz desde la GUI

La posición de la luz puede ser ajustada mediante un slider en `ImGui`:

```
static float lightPos[3] = {0.0f, 5.0f, 0.0f};  
if (ImGui::SliderFloat3("Light Position", lightPos, -20.0f, 20.0f)) {  
    renderer->setLightPosition(lightPos[0], lightPos[1], lightPos[2]);  
}
```

### 7.3.2. Actualización del pipeline

El método `setLightPosition` actualiza el valor en los uniformes del renderer, permitiendo que el cálculo de iluminación se adapte a la nueva posición en tiempo real. Esto afecta directamente al sombreado en los shaders.

## 7.4. Visualización y edición de objetos SDF

La interfaz gráfica permite visualizar, editar y eliminar objetos SDF desde el panel de objetos. El usuario puede modificar posición, color, tamaño y tipo de operación de cada objeto.

### 7.4.1. Edición en tiempo real

Al seleccionar un objeto, se muestran controles específicos según el tipo (esfera, caja, cono, cilindro). Los cambios realizados se reflejan inmediatamente en la escena, marcando el pipeline como "dirty" para que el renderer actualice el renderizado.

```
ImGui::SliderFloat("Radius", &selectedObject.size[0], 0.1f, 5.0f);  
ImGui::ColorEdit3("Color", &selectedObject.r);
```

#### 7.4.2. Eliminación y gestión de objetos

La interfaz incluye un botón "Delete Object" que elimina el objeto seleccionado. El método `deleteObject` ajusta el identificador y actualiza la escena.

### 7.5. Control de operaciones entre objetos

Copper soporta operaciones booleanas y suaves entre objetos: unión, intersección, sustracción, smooth union, smooth subtract. El usuario selecciona la operación desde la GUI y el sistema actualiza el shader generado.

#### 7.5.1. Integración y propagación de cambios

El cambio de operación en la interfaz se propaga al objeto y marca el pipeline como "dirty". El método `generateShaderCode` en `Coder.cpp` añade la operación correspondiente en el código WGSL, permitiendo combinaciones arbitrarias.

### 7.6. Mostrar/ocultar el plano de referencia (floor)

El plano de referencia (floor) puede activarse o desactivarse desde la interfaz gráfica. El parámetro `renderer->floor` controla si se incluye el plano en la generación del shader y en el renderizado. Esta funcionalidad ayuda a mejorar la visualización y ajuste de los objetos en la escena.

### 7.7. Visualización de FPS y métricas

La interfaz muestra el número de frames por segundo (FPS) utilizando `ImGui::GetIO().Framerate`, permitiendo valorar el rendimiento del sistema durante la edición y el renderizado.



## Capítulo 8

# Pruebas



## Capítulo 9

# Conclusiones





# Bibliografía

- [Ric73] A. Ricci. “A Constructive Geometry for Computer Graphics”. En: *The Computer Journal* 16.2 (mayo de 1973), págs. 157-160. DOI: <http://dx.doi.org/10.1093/comjnl/16.2.157>.
- [WMW86] Geoff Wyvill, Craig McPheeters y Brian Wyvill. “Data Structure for Soft Objects”. En: *The Visual Computer* 2.4 (1986), págs. 227-234. DOI: 10.1007/BF01900346. URL: <https://link.springer.com/article/10.1007/BF01900346>.
- [HSK89] John C. Hart, Daniel J. Sandin y Louis H. Kauffman. “Ray Tracing Deterministic 3-D Fractals”. En: *Computer Graphics* 23.3 (jul. de 1989), págs. 289-296. DOI: 10.1145/74334.74340. URL: <https://www.cs.drexel.edu/~david/Classes/Papers/rtqjs.pdf>.
- [Har96] John C. Hart. “Sphere Tracing: A Geometric Method for the Antialiased Ray Tracing of Implicit Surfaces”. En: *The Visual Computer* 12.10 (1996), págs. 527-545. DOI: 10.1007/s003710050084. URL: <https://link.springer.com/article/10.1007/s003710050084>.
- [Qui08] Iñigo Quílez. *Raymarching Distance Fields*. <https://iquilezles.org/articles/raymarchingdf/>. Accessed: 2025-08-11. 2008.



Apéndice A

Código



## Apéndice B

### Escenas



# Glosario

**Callback** Función que se ejecuta en respuesta a un evento, como la pulsación de un botón o el redimensionado de una ventana..

**Constructive Solid Geometry (CSG)** Método de modelado geométrico que combina primitivas mediante operaciones booleanas como unión, intersección y diferencia..

**Cuaternión** Estructura matemática utilizada para representar rotaciones en el espacio tridimensional, evitando problemas como el gimbal lock..

**Demoscene** Comunidad de programadores, artistas y músicos que crean producciones audiovisuales en tiempo real para mostrar destreza técnica y creatividad..

**Framerate** Número de imágenes (frames) renderizadas por segundo, indicador del rendimiento de una aplicación gráfica..

**Función Implícita** Función matemática que describe una superficie como el conjunto de puntos que satisfacen una ecuación dada, sin necesidad de una parametrización explícita..

**Gizmo** Elemento gráfico interactivo en una interfaz que permite manipular objetos o parámetros visualmente, como mover, rotar o escalar en aplicaciones de diseño o gráficos 3D..

**GLFW** Biblioteca multiplataforma para la gestión de ventanas, entrada de usuario y eventos en aplicaciones gráficas..

**GLM** OpenGL Mathematics, biblioteca para operaciones matemáticas con vectores, matrices y cuaterniones en gráficos 3D..

**Hypertexture** Técnica de Ken Perlin y Louis Hoffert para renderizar volúmenes procedurales, que sirvió de base para el desarrollo del ray marching..

- ImGui** Biblioteca de interfaz gráfica inmediata utilizada para crear menús, controles y herramientas interactivas en aplicaciones gráficas..
- Picking** Técnica para identificar y seleccionar objetos en una escena gráfica mediante la posición del cursor o eventos de usuario..
- Pipeline** Secuencia de etapas de procesamiento en la GPU para renderizar gráficos, desde la entrada de vértices hasta la salida de píxeles..
- Ray Marching** Técnica de renderizado que recorre un rayo en pasos discretos para encontrar intersecciones con superficies definidas implícitamente..
- Renderer** Módulo o componente encargado de gestionar el proceso de renderizado de la escena y la comunicación con la GPU..
- Shader** Programa que se ejecuta en la GPU para transformar vértices, calcular colores y simular efectos visuales en el renderizado..
- Signed Distance Function (SDF)** Función que, dado un punto en el espacio, devuelve la distancia mínima a la superficie más cercana, con signo positivo si el punto está fuera y negativo si está dentro..
- Smooth Blending** Técnica para suavizar las transiciones entre primitivas geométricas en modelado implícito, evitando uniones abruptas..
- Sphere Tracing** Método propuesto por Hart en 1995 para recorrer campos de distancia en el renderizado de superficies implícitas..
- Uniform** Variable global enviada desde la CPU al shader, utilizada para transmitir parámetros como matrices, colores o posiciones..
- WebGPU** Estándar gráfico moderno que proporciona acceso eficiente y multiplataforma a la GPU, tanto en navegadores como en aplicaciones nativas..
- WGSL** WebGPU Shading Language, lenguaje nativo de WebGPU para escribir shaders y operaciones matemáticas avanzadas..



