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, seleccion 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 realtime 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 3 de Septiembre de 2025.

D. **Juan Carlos Torres Cantero y Luis López Escudero**, Profesores del Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Granada.

Informan:

Que el presente trabajo, titulado ***Desarrollo de una aplicación de modelado 3D basada en Signed Distance Functions (SDF)*** , ha sido realizado bajo su supervisión por **Juan Carlos Torres Cantero y Luis López Escudero**, 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 5 de Septiembre de 2025.

Los directores:

Juan Carlos Torres Cantero **Luis López Escudero**

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 de gran valor.

Índice general

1. Introducción **19**
1.1. Motivación 20
1.2. Descripción del problema 20
1.3. Objetivos 21
1.4. Estructura de la memoria 22

2. Fundamentos teóricos **23**
2.1. Modelado tridimensional: Paradigmas y fundamentos 23
2.2. Funciones de distancia con signo (SDF) 23
2.3. Ray Marching 24
2.3.1. Ray Tracing 25
2.3.2. Sphere Tracing 25
2.3.3. Ventajas y limitaciones de las SDF 25
2.4. WebGPU: Acceso moderno a la GPU 26
2.4.1. Motivación y principios de diseño 27
2.4.2. Conceptos clave de WebGPU 27
2.5. Shaders y lenguaje WGSL 28
2.5.1. Dawn: Implementación nativa de WebGPU 28
2.6. Herramientas auxiliares 29

3. Arquitectura del sistema y compilación **31**
3.1. Visión General de la Arquitectura 31
3.2. Análisis de Componentes por Capa 32
3.2.1. Capa de Orquestación y Ciclo de Vida 33
3.2.2. Capa de Datos y Lógica de la Escena (Modelo) 33
3.2.3. Capa de Presentación (Vista) 33
3.2.4. Capa de Control de Usuario (Controlador) 34
3.3. Conclusiones sobre la Arquitectura 34
3.4. Sistema de construcción y compilación del proyecto 35
3.4.1. Estructura del archivo CMake 35
3.4.2. Proceso de compilación 37
3.4.3. Función del sistema de construcción 38

4. Implementación del módulo Ventana **39**
4.1. Introducción 39
4.2. Diseño y estructura de la clase Window 39
4.2.1. Atributos principales 39
4.2.2. Métodos públicos 39
4.2.3. Callbacks de eventos 40
4.3. Proceso de inicialización y configuración 40
4.4. Gestión avanzada de eventos 41
4.5. Interacción con otros componentes 41

5. Implementación de la cámara **43**
5.1. La matriz de vista en Copper 43
5.2. Control de la cámara y actualización de la vista 44
5.3. Paso de la matriz de cámara al shader: Uniforms 45
5.3.1. Motivos para utilizar una matriz 4×4 en la cámara de Copper 46

6. Implementación y diseño de los controles **47**
6.1. Introducción 47
6.2. Controles de interfaz gráfica: ImGui 47
6.2.1. Estructura y funciones principales 47
6.2.2. Ejemplo de interacción 48
6.3. Controles de gizmo: GizmoControls 49
6.3.1. Arquitectura y flujo de interacción 49
6.3.2. Gizmo de movimiento lineal en ejes X, Y y Z 50
6.3.3. Gizmo de movimiento en planos XY, XZ y YZ 51

7. Implementación de los Shaders **53**
7.1. Introducción 53
7.2. Arquitectura General 53
7.3. Funcionamiento General 53
7.4. Generación Dinámica del Código Shader 54
7.5. Personalización y Parametrización 54
7.6. Funciones SDF y Operaciones Booleanas 55
7.7. Renderizado y Ray Marching 55
7.7.1. Cálculo de Normales 56
7.7.2. Modelo de Iluminación Blinn-Phong 56
7.7.3. Cálculo de Sombra 57
7.8. Picking en el shader 57

8. Funcionalidades adicionales implementadas **61**
8.1. Introducción 61
8.2. Gestión de la escena: guardado y carga 61
8.2.1. Guardar escena 61

8.2.2. Cargar escena 61
8.2.3. Integración en la interfaz 62
8.3. Visualización de FPS y métricas 62
8.3.1. Pruebas de rendimiento con `addTest` 62

9. Conclusiones y mejoras futuras **65**

Bibliografía **66**

A. Escenas **67**

Glosario **69**

Índice de figuras

2.1. Representación visual del algoritmo de sphere tracing, en el que cada círculo representa el rango libre de obstáculos determinado por la SDF. 25

3.1. Diagrama de la arquitectura por componentes de *Copper*. . . 32

5.1. Vectores base del espacio de la cámara, $\vec{x}$, $\vec{y}$, $\vec{z}$ y posición c. . 44

6.1. Interfaz gráfica ImGui general. 48
6.2. Interfaz gráfica ImGui del objeto seleccionado. 49
6.3. Visualización de gizmo de manipulación. 49

A.1. Combinación de cubo y esfera mediante SDF. 67
A.2. Escena con objetos sin plano de referencia. 67
A.3. Escena con objetos y plano de referencia activado. 68
A.4. Ejemplo de operación smooth subtract entre objetos. 68
A.5. Ejemplo de operación smooth union entre objetos. 68
A.6. Ejemplo de operación de sustracción entre objetos. 69
A.7. Ejemplo de operación de unión entre objetos. 69

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 y diseño de los controles:** Se aborda el desarrollo de la interfaz de usuario, incluyendo la disposición de los elementos, la gestión de eventos y la interacción con la escena 3D.
- **Implementación de los shaders:** Se describe el desarrollo del generador de código de WGSL, y su actualización dinámica para controlar la escena.
- **Implementación de la cámara:** Se detalla la construcción de la matriz de vista y la gestión de la posición y orientación de la cámara en el espacio 3D.
- **Funcionalidades adicionales implementadas:** Se documentan las características extra añadidas a la aplicación, como el sistema de guardado/carga de escenas y la gestión de la luz.
- **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 basa en el modelado mediante funciones de distancia con signo (SDF), el renderizado por *ray marching* y el uso del estándar gráfico WebGPU, usando Dawn para aplicarlo a C++. Este capítulo describe en profundidad cada uno de estos fundamentos.

2.1. Modelado tridimensional: Paradigmas y fundamentos

El modelado tridimensional tradicional se basa en mallas poligonales, en las que los objetos se representan mediante listas de vértices, aristas y caras conectadas para formar la superficie. Herramientas como *Blender*, *Maya*, y *3ds Max* utilizan este enfoque, que ofrece gran flexibilidad para la edición y animación, pero que implica la gestión explícita de la topología, el almacenamiento de grandes cantidades de datos y una complejidad elevada para ciertas operaciones.

Como alternativa, existen los métodos de representación implícita, como los campos de distancia con signo (*Signed Distance Fields, SDF*). 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[Har96]. 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.

2.2. 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.
- **Cilindro, cono, plano...**: Cada primitiva se expresa como una función matemática que determina la distancia a su superficie.

Y también 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.

Las SDF son especialmente potentes para casos de generación procedural, efectos visuales, física, simulaciones y renderizado implícito, permitiendo la composición jerárquica de formas y la aplicación de transformaciones a nivel funcional.

2.3. Ray Marching

El *ray marching* es una técnica de renderizado que avanza iterativamente un rayo en el espacio hasta aproximar la intersección con una superficie implícita[Har96; PJH16]. El procedimiento puede describirse en los siguientes pasos:

1. Lanzar un rayo desde la cámara en una dirección determinada.
2. Evaluar la distancia para el siguiente punto a lo largo del rayo.
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.

Este método sumado a las SDF se llama sphere tracing, en este caso la distancia devuelta por la SDF se utiliza para avanzar el rayo de manera óptima.

2.3.1. Ray Tracing

A diferencia del ray marching, el ray tracing cálcula la primera intersección exacta entre el rayo y las primitivas geométricas de la escena[Whi80]. Este método es especialmente eficiente para escenas donde la geometría está definida de forma explícita, como mallas poligonales. El procedimiento es similar al ray marching, se lanza un rayo desde la cámara, pero esta vez en vez de avanzar iterativamente, se calcula cual es el primer objeto con el que colisionaría el rayo.

El ray tracing no se puede usar con SDF, ya que esta solo devuelve la distancia al objeto más cercano.

2.3.2. Sphere Tracing

La técnica de **sphere tracing**, introducida por Hart[Har96], es una variante eficiente de ray marching. Utiliza la propia SDF para calcular el avance óptimo en cada paso del rayo. En cada iteración, la distancia devuelta por la SDF se interpreta como el radio de una esfera libre de obstáculos centrada en el punto actual. Así, el rayo puede avanzar exactamente esa distancia sin riesgo de atravesar ninguna superficie.

Sphere tracing es especialmente útil en escenas donde las funciones de distancia son suaves y bien definidas, permitiendo renderizar geometría compleja con costes computacionales bajos. Sin embargo, en superficies muy delgadas o SDFs poco continuas, el algoritmo puede avanzar muy poco en cada paso, reduciendo la eficiencia y generando artefactos visuales.

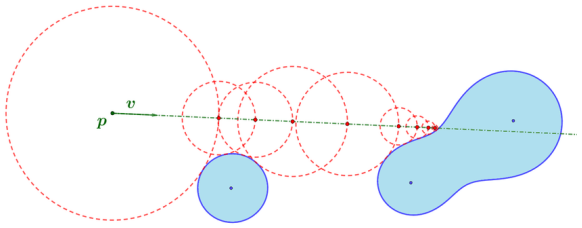

Figura 2.1: Representación visual del algoritmo de sphere tracing, en el que cada círculo representa el rango libre de obstáculos determinado por la SDF.

2.3.3. Ventajas y limitaciones de las SDF

Las SDF ofrecen varias ventajas sobre los métodos de modelado tradicionales:

- **Compacidad**: Las SDF se definen por fórmulas matemáticas en lugar de listas de vértices, lo que reduce el espacio necesario para describir objetos complejos.

- **Facilidad de combinación**: La geometría puede combinarse empleando operadores matemáticos (unión, intersección, resta) de forma eficiente y expresiva.
- **Transformaciones geométricas**: Las transformaciones como traslación, rotación y escalado se aplican directamente sobre la función, facilitando la manipulación.
- **Cálculo de normales**: La normal en la superficie se obtiene como el gradiente de la función de distancia, lo que simplifica el cálculo de iluminación.
- **Flexibilidad**: Permiten crear transiciones suaves entre objetos mediante operadores suavizados, lo que es difícil de lograr con mallas.

Pero también presentan diferentes limitaciones a la hora de representar geometría:

- **Precision y aliasing**: El umbral de colisión y el número de pasos afectan la calidad de la imagen y pueden causar aliasing o superficies rugosas.
- **Superficies delgadas**: Si la SDF varía abruptamente o la superficie es muy fina, el avance óptimo se reduce y el número de pasos aumenta significativamente.
- **Efectos avanzados**: Efectos como la refracción y la reflexión requieren múltiples rayos y aumentan el coste computacional.
- **Desempeño**: El rendimiento depende de la complejidad de las funciones SDF y del número de objetos en escena.
- **Depuración**: La depuración de SDF complejas puede resultar poco intuitiva debido a su naturaleza matemática y a la falta de herramientas específicas, requiriendo de herramienta de visualización adicionales o renderizados auxiliares.

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. Surge como respuesta a las limitaciones de APIs tradicionales como OpenGL y DirectX, para su uso en la web, pretende ser el sucesor de WebGL, ofreciendo un acceso más directo y eficiente a las capacidades de la GPU.

2.4.1. Motivación y principios de diseño

WebGPU fue diseñado para proporcionar los siguientes atributos a los desarrolladores:

- **Multiplataforma**: Disponible en Windows, Linux, macOS y en navegadores modernos (Chrome, Firefox, Safari).
- **Eficiencia**: Permite describir el flujo de datos entre la CPU y la GPU mediante buffers y pipelines, minimizando el coste de las llamadas de función y la sobrecarga del sistema.
- **Seguridad y portabilidad**: WebGPU abstrae detalles específicos del hardware, garantizando que el mismo código funcione en diferentes dispositivos y plataformas.
- **Modelo explícito**: El usuario configura directamente los recursos (buffers, texturas, pipelines) y controla el ciclo de vida de los datos, permitiendo optimizaciones avanzadas.

WebGPU optimiza la comunicación entre la CPU y la GPU, permitiendo un alto grado de paralelismo. A diferencia de APIs más antiguas, donde la CPU a menudo debe esperar a que la GPU complete una tarea, WebGPU permite enviar comandos de manera asíncrona.

El flujo de trabajo se basa en la creación de **colas de comandos** (*command queues*). La CPU registra una serie de operaciones (dibujar objetos, actualizar buffers, etc.) en un *command buffer*, que luego se envía a la cola de la GPU para su ejecución. Una vez encolados, los comandos se ejecutan de manera asíncrona, aunque el programador dispone de mecanismos de sincronización como *fences* y *promises* (p. ej., `queue.onSubmittedWorkDone()`) si necesita coordinar CPU y GPU. Este modelo minimiza los tiempos de espera y maximiza el uso de ambos procesadores, lo que es fundamental para aplicaciones de alto rendimiento como el renderizado en tiempo real.

2.4.2. Conceptos clave de WebGPU

- **Buffers**: Áreas de memoria para almacenar datos como vértices, índices y uniforms.
- **Textures**: Imágenes y mapas de datos que pueden ser leídos y escritos por la GPU.
- **Bind Groups**: Conjuntos de recursos que se vinculan a los shaders, permitiendo el acceso eficiente a datos en la GPU.
- **Pipelines**: Describen la secuencia de operaciones gráficas, incluyendo shaders, estados de rasterización y configuración de recursos.

- **Shaders**: Programas ejecutados en la GPU que transforman datos y calculan colores de píxeles.
- **Command Buffers**: Secuencias de instrucciones que la GPU ejecuta para renderizar o procesar datos.

WebGPU utiliza el lenguaje WGSL (*WebGPU Shading Language*) para la programación de shaders, ofreciendo una sintaxis moderna y expresiva adaptada a las necesidades de la computación gráfica actual.

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. En WebGPU, los shaders se escriben en WGSL (*WebGPU Shading Language*), un lenguaje moderno 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, combinando eficiencia y expresividad.

2.5.1. Dawn: Implementación nativa de WebGPU

Dawn es una implementación nativa de la API WebGPU desarrollada por Google y la comunidad Chromium[GC24]. Dawn permite ejecutar aplicaciones que usan WebGPU fuera del navegador, proporcionando acceso directo y multiplataforma a la GPU. Es el backend utilizado por Copper para interactuar con WebGPU desde C++.

En Copper, Dawn se integra como una dependencia externa, permitiendo:

- **Creación de instancias de WebGPU**: A través de las clases y funciones de Dawn, Copper inicializa `wgpu::Instance`, `wgpu::Device`, y otros objetos fundamentales.
- **Gestión de recursos**: Dawn facilita la creación y gestión de buffers, texturas y pipelines, siguiendo la especificación de WebGPU.

- **Integración con GLFW**: Mediante el módulo `webgpu-glfw`, Copper conecta la gestión de ventanas de GLFW con las superficies de renderizado de Dawn/WebGPU.

El ciclo de inicialización y renderizado en Copper se basa en la creación de la instancia de Dawn (`CreateInstance`), la selección de adaptador y dispositivo (`GetAdapter`, `GetDevice`), y la configuración de la superficie de dibujo (`ConfigureSurface`). Los comandos de renderizado se envían a la GPU mediante `CommandEncoder` y `RenderPassEncoder`, siguiendo el modelo de WebGPU.

2.6. Herramientas auxiliares

El desarrollo de aplicaciones gráficas requiere gestionar ventanas, entrada de usuario y la interfaz gráfica. Copper utiliza las siguientes herramientas:

- **GLFW**: Biblioteca multiplataforma para la gestión de ventanas y eventos, usada a partir de su módulo de Dawn[Dev24].
- **ImGui**: Sistema de interfaz gráfica inmediata para la manipulación interactiva de primitivas y parámetros de escena[Cor24].
- **GLM**: Biblioteca matemática para operaciones con vectores y matrices[Cre24].
- **CMake**: Herramienta de compilación y gestión de dependencias[Kit24].

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

Capítulo 3

Arquitectura del sistema y compilación

El desarrollo de una aplicación gráfica interactiva como *Copper* requiere una base estructural robusta que garantice la separación de responsabilidades, la mantenibilidad y la escalabilidad del código. Una arquitectura de software bien definida es fundamental para gestionar la complejidad inherente a la renderización en tiempo real, la gestión del estado de la escena y la interacción con el usuario.

En este capítulo se presenta un análisis detallado de la arquitectura de *Copper*. El sistema no sigue de forma estricta ningún patrón de diseño, sino que implementa una arquitectura por componentes con una fuerte inspiración en el patrón Modelo-Vista-Controlador (MVC). Este enfoque pragmático permite que cada componente se especialice en una tarea concreta (renderizado, lógica de la escena, control de entrada), interactuando con otros para construir la funcionalidad completa de la aplicación.

A continuación, se describirán los componentes principales, sus responsabilidades y las interacciones entre ellos.

3.1. Visión General de la Arquitectura

La arquitectura de *Copper* se puede descomponer en cuatro capas lógicas principales que encapsulan las diferentes funcionalidades, como se ve en la Figura3.1.

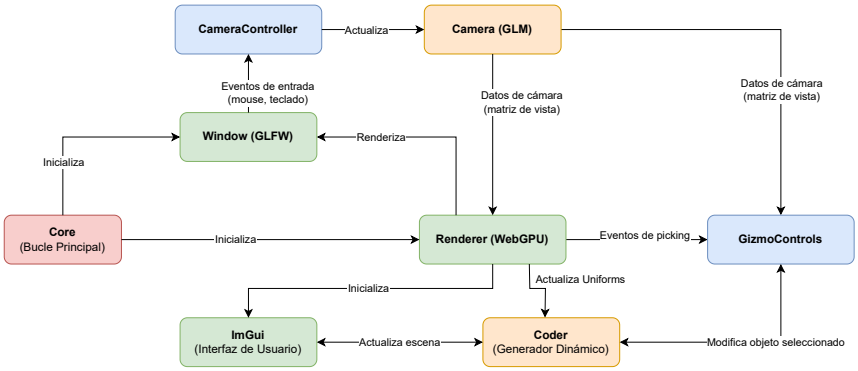

Figura 3.1: Diagrama de la arquitectura por componentes de *Copper*.

Las capas identificadas son:

- **Capa de Orquestación y Ciclo de Vida (rojo):** Responsable de inicializar, ejecutar y terminar la aplicación. Es el punto de entrada y el gestor del bucle principal.
- **Capa de Datos y Lógica de la Escena (Modelo) (naranja):** Contiene el estado de la escena 3D, incluyendo los objetos, sus propiedades y la cámara. Encapsula la lógica de negocio, como la generación dinámica de shaders.
- **Capa de Presentación (Vista) (verde):** Se encarga de toda la representación visual, tanto el renderizado de la escena 3D mediante WebGPU como la interfaz gráfica de usuario (GUI) construida con ImGui.
- **Capa de Control de Usuario (Controlador) (azul):** Captura y procesa las entradas del usuario (ratón y teclado) y las traduce en acciones que modifican el estado del Modelo o la Vista.

Esta separación permite que, por ejemplo, se pueda modificar el motor de renderizado (`Renderer`) sin afectar a la lógica de la escena (`Coder`), o cambiar la forma en que se controla la cámara (`CameraController`) sin alterar su representación de datos (`Camera`).

3.2. Análisis de Componentes por Capa

A continuación, se detallan los componentes clave dentro de cada capa lógica, referenciando sus ficheros de implementación.

3.2.1. Capa de Orquestación y Ciclo de Vida

El núcleo de la aplicación reside en la clase `Core`, definida en `src/core/Core.h` y `src/core/Core.cpp`.

`Core`: Actúa como el orquestador principal. Su responsabilidad es inicializar todos los subsistemas en el orden correcto (`Core::Initialize`), gestionar el bucle principal de la aplicación (`Core::MainLoop`) y liberar los recursos de forma ordenada al finalizar (`Core::Terminate`). El bucle principal consulta el estado de la ventana y delega la tarea de dibujado al `Renderer` en cada fotograma.

3.2.2. Capa de Datos y Lógica de la Escena (Modelo)

Esta capa representa el estado y la lógica de negocio de la aplicación.

`Coder` (`src/core/Coder.h`, `.cpp`) : Es el componente central del modelo. Gestiona la lista de objetos 3D de la escena (`std::vector<Object>`), sus propiedades (posición, color, forma) y las operaciones booleanas entre ellos. Su responsabilidad más crítica es el método `generateShaderCode()`, que traduce dinámicamente el estado de los objetos en código de shader WGSL. Este mecanismo es la "lógica de negocio"principal de *Copper*, ya que define cómo se representa visualmente la escena. También se encarga de la persistencia de la escena mediante las funciones `saveScene()` y `loadScene()`.

`Camera` (`src/core/Camera.h`, `.cpp`) : Representa los datos y el estado de la cámara virtual. Almacena la matriz de vista (`view_matrix`) y proporciona métodos para acceder a sus vectores fundamentales (posición, dirección, etc.). Aunque es una entidad pasiva, es una parte fundamental del modelo de datos, ya que define la perspectiva desde la que se observa la escena.

3.2.3. Capa de Presentación (Vista)

Esta capa es responsable de todo lo que el usuario ve en la pantalla.

`Renderer` (`src/core/Renderer.h`, `.cpp`) : Es la vista principal. Su función es dibujar la escena 3D utilizando la API WebGPU. Orquesta el proceso de renderizado en su método `Render()`, donde toma los datos del `Coder` y la `Camera`, actualiza los *uniforms* del shader y ejecuta los comandos de dibujado. También gestiona recursos de la GPU como la *pipeline* de renderizado y los búferes.

`Interfaz` (`src/ui/Interfaz.h`, `.cpp`) : Componente especializado de la vista que gestiona la Interfaz Gráfica de Usuario (GUI) mediante la

biblioteca ImGui. Es responsable de crear todos los paneles, botones y controles deslizantes que permiten al usuario interactuar con la aplicación. Lee el estado directamente del `Coder` para mostrar las propiedades de los objetos y, a su vez, invoca métodos en el `Coder` para modificar dicho estado, lo que representa una desviación del patrón MVC estricto.

`Window` (`src/ui/Window.h, .cpp`) : Gestiona la ventana nativa de la aplicación utilizando GLFW. Proporciona el "lienzo"sobre el que dibujan el `Renderer` y la `Interfaz`. Además, es responsable de registrar y despachar los eventos de entrada del sistema operativo (como clics de ratón o cambios de tamaño) a los componentes correspondientes.

3.2.4. Capa de Control de Usuario (Controlador)

Esta capa gestiona la entrada del usuario y la traduce en comandos para el Modelo y la Vista.

`CameraController` (`src/core/CameraController.h, .cpp`) : Es un controlador especializado en la manipulación de la cámara. Captura los eventos de arrastre del ratón y la rueda de desplazamiento para calcular la nueva orientación y posición de la cámara. En su método `update_camera()`, modifica directamente el estado del objeto `Camera` (el Modelo) para reflejar la interacción del usuario.

`GizmoControls` (`src/core/GizmoControls.h, .cpp`) : Otro controlador altamente especializado, dedicado a la manipulación de los gizmos de transformación de los objetos. Cuando el usuario interactúa con un gizmo, esta clase calcula el desplazamiento correspondiente y actualiza la posición del objeto seleccionado. El nuevo estado se comunica al `Coder` para que la escena se actualice.

`Renderer` **como despachador de eventos** : Excepcionalmente, la clase `Renderer` también asume un rol de controlador, debido al picking realizado por shader. Su método `OnMouseButton()` actúa como un punto de entrada que decide si la acción del usuario corresponde a una selección de objeto (lo que desencadena la pipeline de *picking*) o a la manipulación de un gizmo (delegando el control a `GizmoControls`).

3.3. Conclusiones sobre la Arquitectura

La arquitectura de *Copper* puede definirse como un sistema basado en componentes con una clara separación de responsabilidades, que se alinea con los principios del patrón Modelo-Vista-Controlador (MVC) sin seguir su implementación más estricta.

La separación es evidente:

- **Modelo**: `Coder` y `Camera` contienen el estado y la lógica central.
- **Vista**: `Renderer` e `Interfaz` se ocupan de la presentación visual.
- **Controlador**: `CameraController` y `GizmoControls` gestionan la entrada del usuario de forma aislada.

Sin embargo, se observan desviaciones pragmáticas del patrón MVC clásico, como la comunicación directa entre la Vista (`Interfaz`) y el Modelo (`Coder`) para modificar propiedades de objetos. Este enfoque, común en aplicaciones gráficas, reduce la complejidad al eliminar la necesidad de un controlador intermediario para cada pequeña modificación de estado.

En conclusión, la estructura elegida es eficaz y adecuada para una aplicación de modelado 3D. Promueve la cohesión dentro de cada componente y mantiene un bajo acoplamiento entre las distintas capas lógicas, lo que facilita el mantenimiento, la depuración y la futura expansión del software.

3.4. Sistema de construcción y compilación del proyecto

El proyecto utiliza **CMake** como sistema de construcción para gestionar la compilación, las dependencias y la organización de los archivos fuente. CMake es una herramienta ampliamente empleada en proyectos de desarrollo de software en C y C++ por su capacidad para generar archivos de construcción para distintos sistemas y entornos [Kit24].

3.4.1. Estructura del archivo CMake

En la raíz del proyecto se encuentra el archivo `CMakeLists.txt`, que define la configuración de la construcción. Sus principales componentes y funcionalidades son:

- **Definición del proyecto:**
  El proyecto se denomina `Copper`, está configurado para compilar en C++20 y se identifica como una aplicación de modelado 3D basada en SDF:

```
1  project(Copper VERSION 0.0.2 LANGUAGES CXX C
   DESCRIPTION "3D SDF-based modeling application")
2  set(CMAKE_CXX_STANDARD 20)
3  set(CMAKE_CXX_STANDARD_REQUIRED ON)
4  set(CMAKE_CXX_EXTENSIONS OFF)
```

Listing 3.1: Definición del proyecto en CMakeLists.txt

- **Gestión de dependencias externas:**
  El archivo especifica la inclusión y compilación de bibliotecas externas necesarias para el funcionamiento del proyecto, como Dawn (WebGPU), GLM (matemáticas), ImGuiFileDialog (diálogos de archivos en la interfaz gráfica), y las fuentes de ImGui. La gestión de dependencias se realiza mediante `add_subdirectory` y `FetchContent`, permitiendo su descarga y compilación automática junto al proyecto principal. Es importante el uso de `DAWN_FETCH_DEPENDENCIES` para no tener que instalar todas las librerías que usa Dawn ya que estos son pesados y pueden no ser necesarios, y así evitamos gestionar manualmente las dependencias.

- **Organización de los archivos fuente:**
  Se recopilan y organizan los archivos fuente del proyecto, excluyendo explícitamente el fichero `main.cpp` para evitar duplicidad, y se añaden los archivos fuente de ImGui.

```
1            file(GLOB_RECURSE COPPER_SOURCES src/*.cpp src
   /*.h)
2            list(REMOVE_ITEM COPPER_SOURCES ${
   CMAKE_SOURCE_DIR}/src/main.cpp)
```

Listing 3.2: Organización de los archivos fuente en CMakeLists.txt

- **Creación del ejecutable:**
  Se define el ejecutable principal `copper`, que se compila a partir de `main.cpp`, los archivos fuente propios y los de ImGui.

```
1    add_executable(copper src/main.cpp ${COPPER_SOURCES} $
   {IMGUI_SOURCES})
```

Listing 3.3: Creación del ejecutable en CMakeLists.txt

- **Definición de directorios de inclusión:**
  Se especifican los directorios que contienen los archivos de cabecera para facilitar la compilación.

```
1    target_include_directories(copper PRIVATE src lib/
   imgui lib/imgui/misc/cpp lib/imgui/backends ${
   CMAKE_CURRENT_BINARY_DIR}/src)
```

Listing 3.4: Definición de directorios de inclusión en CMakeLists.txt

- **Vinculación de bibliotecas:**
  Se vinculan las bibliotecas necesarias para el funcionamiento del programa, como Dawn/WebGPU, GLM, GLFW y el soporte para WebGPU en GLFW.

```
1    target_link_libraries(copper PRIVATE dawn::webgpu_dawn
     glm::glm glfw webgpu_glfw ImGuiFileDialog)
```

Listing 3.5: Vinculación de bibliotecas en CMakeLists.txt

- **Definiciones de compilador:**
  Se incluyen definiciones específicas que permiten, por ejemplo, indicar el directorio de recursos y activar la integración de ImGui con WebGPU usando Dawn.

```
target_compile_definitions(copper PRIVATE
RESOURCE_DIR="${CMAKE_CURRENT_SOURCE_DIR}/src"
IMGUI_IMPL_WEBGPU_BACKEND_DAWN)
```

Listing 3.6: Definiciones de compilador en CMakeLists.txt

3.4.2. Proceso de compilación

El desarrollo principal se ha llevado a cabo sobre un entorno Linux, aunque la elección de herramientas y librerías multiplataforma debería asegurar su compilación en otros sistemas operativos como Windows o macOS, esto podría requerir ajustes en la configuración del proyecto.

Para compilar el proyecto, se debe disponer de CMake y un compilador compatible con C++20. El proceso estándar usando git para compilar el proyecto consiste en:

1. Si fuese necesario añadir los paquetes utilizados por Dawn:

```
sudo apt-get install libxrandr-dev libxinerama-dev
libxcursor-dev mesa-common-dev libx11-xcb-dev pkg-
config nodejs npm
```

2. Clonar el repositorio de Git y actualizar los submódulos:

```
git clone https://github.com/tu-usuario/copper.git
cd copper
git submodule update --init
```

3. Compilar en la carpeta build:

```
cmake -B build
cmake --build build -j$(nproc)
```

4. Por último si se quiere ejecutar el proyecto:

```
./build/copper
```

Esto construirá el ejecutable `copper`, incluyendo todas las dependencias y archivos fuente indicados en `CMakeLists.txt`.

3.4.3. Función del sistema de construcción

El sistema CMake facilita:

- La gestión automática de dependencias externas.
- La configuración multiplataforma.
- La compilación eficiente de todos los módulos del programa, asegurando que las dependencias y rutas de inclusión están correctamente resueltas.
- La integración de recursos y librerías de terceros necesarios para la ejecución (por ejemplo, ImGui para la interfaz gráfica, Dawn para el acceso a WebGPU).

La configuración descrita en el archivo `CMakeLists.txt` permite reproducir el proceso de compilación en distintos sistemas, facilitando el trabajo colaborativo y la portabilidad del proyecto.

Capítulo 4

Implementación del módulo Ventana

4.1. Introducción

El módulo **Ventana** constituye el punto de entrada visual y de interacción para el sistema. 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 windowWidth, windowHeight`: 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.

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 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.

Capítulo 5

Implementación de la cámara

Las cámaras son un componente esencial en cualquier sistema de renderizado 3D, ya que permiten definir el punto de vista desde el que se observa la escena. En Copper, la cámara está implementada a través de las clases `Camera` y `CameraController`, las cuales gestionan tanto la transformación de la vista como la interacción del usuario. A continuación, se describe detalladamente cómo se representa y utiliza la cámara en Copper, desde la construcción de la matriz de vista hasta su uso en los shaders mediante el sistema de `Uniforms`.

5.1. La matriz de vista en Copper

En Copper, la cámara se representa principalmente mediante una matriz de vista (`view_matrix`), que transforma las coordenadas de los objetos del espacio de mundo al espacio de la cámara. La clase `Camera` mantiene la posición del ojo (`eye`), el punto de interés (`center`), y el vector de orientación vertical (`up`), que se usan para construir la matriz de vista.

$$V = \begin{pmatrix} \vec{x}_x & \vec{x}_y & \vec{x}_z & c_x \\ \vec{y}_x & \vec{y}_y & \vec{y}_z & c_y \\ \vec{z}_x & \vec{z}_y & \vec{z}_z & c_z \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{5.1}$$

Aquí, $\vec{x}$, $\vec{y}$ y $\vec{z}$ son los vectores base del espacio de la cámara, y $\vec{c}$ es la posición de la cámara. Los vectores base son vectores unitarios que definen la orientación de la cámara; son ortogonales entre sí. Los vectores $\vec{x}$, $\vec{y}$ y $\vec{z}$ también se conocen como los vectores derecho, arriba y adelante respectivamente. El vector hacia adelante siempre apunta en la dirección en la que la cámara está mirando. En la 5.1 se muestra un diagrama de la vista de la cámara, que muestra cómo están orientados los vectores base en el espacio. La posición de la cámara $\vec{c}$ es la posición de la cámara, en coordenadas de cámara.

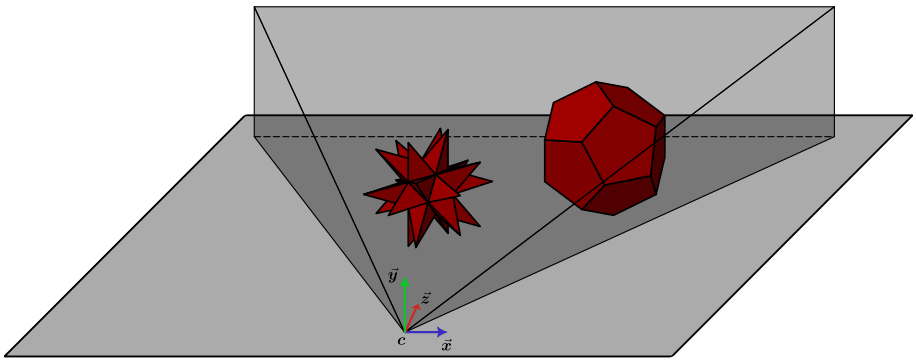

Figura 5.1: Vectores base del espacio de la cámara, $\vec{x}$, $\vec{y}$, $\vec{z}$ y posición c.

La matriz de vista se actualiza usando la función `glm::lookAt` de la biblioteca GLM:

```
void Camera::update_view_matrix() {
    this->view_matrix = glm::lookAt(
        this->eye,
        this->center,
        this->up
    );
}
```

Listing 5.1: Actualización de la matriz de vista en Copper

Esta función genera una matriz de 4x4 que realiza una traslación y rotación para situar el origen de coordenadas en la posición de la cámara y orientar los ejes de acuerdo a los vectores especificados. Los métodos `get_right()`, `get_up()` y `get_forward()` permiten obtener los vectores base del espacio de la cámara directamente desde la matriz de vista transpuesta.

5.2. Control de la cámara y actualización de la vista

La clase `CameraController` gestiona la interacción del usuario para modificar la vista de la cámara mediante el ratón y el scroll, permitiendo rotar, trasladar y acercar/alejar la cámara respecto al centro de la escena. El controlador calcula la nueva orientación usando cuaterniones y actualiza la matriz de vista de la cámara en función de los movimientos del usuario.

```
void CameraController::update_view_matrix() {
    auto matrix = glm::transpose(
        mat4_cast(this->total_rotation)
    );

    auto right = glm::vec3(matrix[0]);
```

```
    auto up = glm::vec3(matrix[1]);
    auto view_dir = glm::vec3(matrix[2]);

    auto camera_center = this->center;
    auto translation = glm::vec3(
        glm::dot(right, camera_center),
        glm::dot(up, camera_center),
        glm::dot(view_dir, camera_center));

    matrix[0][3] = translation.x;
    matrix[1][3] = translation.y;
    matrix[2][3] = translation.z - this->radius;

    this->camera->set_view_matrix(glm::transpose(matrix));
}
```

Listing 5.2: Actualización de la vista desde CameraController

5.3. Paso de la matriz de cámara al shader: Uniforms

Para que el shader pueda utilizar la información de la cámara, la matriz de vista se pasa como parte de la estructura `Uniforms`. En Copper, antes de cada renderizado, la matriz de vista se actualiza y se copia a los datos de `uniformsData`, que luego se transfieren a la GPU:

```
uniformsData.mvp_matrix = this->camera->get_view_matrix();
device.GetQueue().WriteBuffer(uniformsBuffer, 0, &uniformsData,
    sizeof(Uniforms));
```

Listing 5.3: Actualización del uniform con la matriz de vista

En el shader (WGSL), se accede a esta matriz a través del binding de `Uniforms`:

```
struct Uniforms {
    mvp_matrix: mat4x4<f32>,
    //...
};
@group(0) @binding(0) var<uniform> uniforms: Uniforms;

@fragment
fn fragmentMain(in: VertexOutput) -> @location(0) vec4<f32> {
    let cam = transpose(uniforms.mvp_matrix);
    let right = cam[0].xyz;
    let up = cam[1].xyz;
    let forward = cam[2].xyz;
    let eye = cam[0].w * right + cam[1].w * up + cam[2].w *
    forward;
    //...
}
```

Listing 5.4: Uso de la matriz de cámara en el shader

La operación `transpose(uniforms.mvp_matrix)` permite extraer los vectores principales de la cámara desde la matriz de vista, siguiendo la convención de OpenGL y GLM.

5.3.1. Motivos para utilizar una matriz 4×4 en la cámara de Copper

Aunque el cálculo de los rayos de visión en Copper se realiza utilizando únicamente los vectores principales de la cámara (derecha, arriba y adelante), es decir, operando esencialmente en tres dimensiones, la matriz de cámara (`view_matrix`) se mantiene en formato 4×4 durante todo el pipeline, a pesar de poder simplificarse a una matriz 3×4. Esta decisión se ha tomado por razones de compatibilidad, flexibilidad y coherencia con las convenciones de las principales APIs y bibliotecas gráficas.

La matriz de proyección tradicional en gráficos 3D requiere el uso de coordenadas homogéneas (x, y, z, w) y matrices 4×4 para poder representar la perspectiva correctamente[Ahn08]. Sin embargo, en Copper, no se aplica una matriz de proyección, si no que el cálculo de la perspectiva se realiza directamente en el shader, utilizando únicamente los vectores de la cámara y los parámetros de `aspect_ratio` y `FOV`.

En resumen, **Copper calcula la perspectiva y el ray tracing únicamente en tres dimensiones, pero mantiene la matriz de cámara en formato 4×4 por compatibilidad, facilidad de desarrollo y flexibilidad futura**. Este diseño permite adaptar el sistema a cambios sin necesidad de reescribir partes significativas del código y sigue las convenciones del desarrollo gráfico moderno.

Capítulo 6

Implementación y diseño de los controles

6.1. 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, y se divide en dos apartados: controles de interfaz gráfica (ImGui) y controles de gizmo (GizmoControls).

6.2. Controles de interfaz gráfica: ImGui

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

6.2.1. 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.

Figura 6.1: Interfaz gráfica ImGui general.

6.2.2. 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, 5.0
    f);
```

Listing 6.1: Edición de propiedades de un objeto SDF seleccionado

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

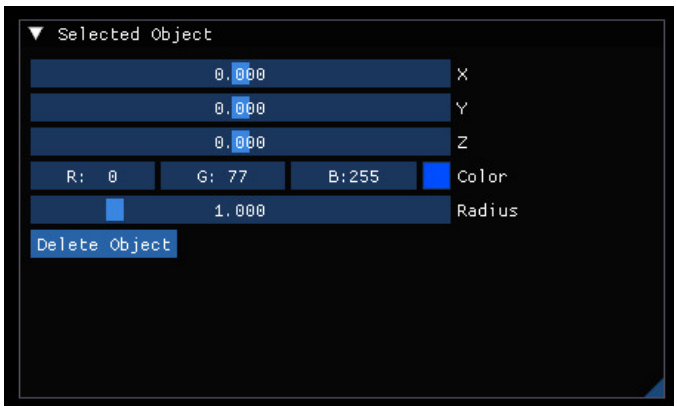

Figura 6.2: Interfaz gráfica ImGui del objeto seleccionado.

6.3. 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).

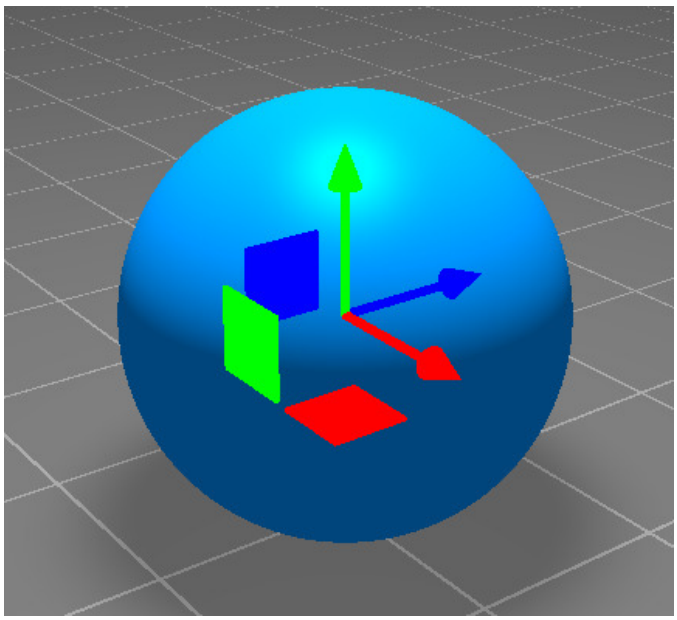

Figura 6.3: Visualización de gizmo de manipulación.

6.3.1. 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`.

6.3.2. Gizmo de movimiento lineal en ejes X, Y y Z

Para implementar el gizmo de movimiento lineal, es necesario implementar un sistema que detecte la cantidad de movimiento en cada eje con respecto al desplazamiento del ratón. Esto significa transformar unas coordenadas *screen space* (2D) a coordenadas *world space* (3D).

El sistema implementado se basa en la intersección entre dos líneas en world space, una línea que representa al eje sobre el que queremos mover el objeto, y otra línea, que representa el rayo *casteado* desde la posición del ratón. Este último rayo parte de la posición de la cámara y se extiende hacia el plano de la escena. La sección de *raymarching* explica como transformar una coordenada UV de la pantalla a rayos casteados desde la posición de la cámara.

Como en 3D es difícil hacer que dos líneas se crucen, y la precisión perdida en las aproximaciones de coma flotante podría afectar a los cálculos. Hemos decidido utilizar una expresión para el punto más cercano entre dos líneas. Estas expresiones son ampliamente conocidas y se puede encontrar en múltiples fuentes. A continuación se deja un desarrollo propio.

Sean dos líneas en el espacio 3D, definidas por sus ecuaciones paramétricas:

$$\begin{cases} p_1 = k_1 + l_1\vec{v}_1, \\ p_2 = k_2 + l_2\vec{v}_2. \end{cases} \tag{6.1}$$

Dónde k_1 y k_2 son dos puntos por los que pasan las líneas, $\vec{v}_1$ y $\vec{v}_2$ son sus vectores directores, y l_1 y l_2 son los parámetros que recorren las líneas, dos números reales. La distancia entre dos puntos arbitrarios en cada línea, p_1 y p_2, se puede expresar como:

$$d = \|p_1 - p_2\| = \|(k_1 + l_1 * \vec{v}_1) - (k_2 + l_2 * \vec{v}_2)\|. \tag{6.2}$$

Descomponiendo los vectores y puntos en sus componentes y tomando el cuadrado de la distancia, obtenemos:

$$\begin{aligned} d^2 = &\left(k_{1_x} + l_1 v_{1_x} - k_{2_x} - l_2 v_{2_x}\right)^2 \\ &+ \left(k_{1_y} + l_1 v_{1_y} - k_{2_y} - l_2 v_{2_y}\right)^2 \\ &+ \left(k_{1_z} + l_1 v_{1_z} - k_{2_z} - l_2 v_{2_z}\right)^2 . \end{aligned} \tag{6.3}$$

Teniendo en cuenta que solo tenemos dos variables, ambas al cuadrado, podemos ver que d^2 es una parábola tridimensional. Como la parábola es positiva, todos los puntos están por encima de $d = 0$, solo tiene un mínimo, el mínimo global. Nuestro problema tiene por tanto una sola solución, que podemos encontrar buscando este mínimo, (no tendremos que descartar puntos de silla).

El mínimo lo encontraremos en el punto en el que la derivada de d^2 con respecto a l_1 y l_2 sea cero.

$$\begin{aligned} \frac{\partial d^2}{\partial l_1} = 0 = &\, 2\left(k_{1_x} + l_1 v_{1_x} - k_{2_x} - l_2 v_{2_x}\right) v_{1_x} \\ &+2\left(k_{1_y} + l_1 v_{1_y} - k_{2_y} - l_2 v_{2_y}\right) v_{1_y} \\ &+2\left(k_{1_z} + l_1 v_{1_z} - k_{2_z} - l_2 v_{2_z}\right) v_{1_z}, \end{aligned} \tag{6.4}$$

$$\begin{aligned} \frac{\partial d^2}{\partial l_2} = 0 = &-2\left(k_{1_x} + l_1 v_{1_x} - k_{2_x} - l_2 v_{2_x}\right) v_{2_x} \\ &-2\left(k_{1_y} + l_1 v_{1_y} - k_{2_y} - l_2 v_{2_y}\right) v_{2_y} \\ &+2\left(k_{1_z} + l_1 v_{1_z} - k_{2_z} - l_2 v_{2_z}\right) v_{2_z}. \end{aligned} \tag{6.5}$$

Desarrollando estas ecuaciones podemos obtener:

$$l1 = \frac{l_2 \vec{v}_2 \cdot \vec{v}_2 - k_1 \cdot k_2 + k_2 \cdot k_2}{v_1 \cdot v_2}, \tag{6.6}$$

$$l2 = \frac{\frac{\vec{v}_1 \cdot \vec{v}_1}{\vec{v}_1 \cdot \vec{v}_2}(k_2 \cdot \vec{v}_2 - k_1 \cdot \vec{v}_2) + k_1 \cdot \vec{v}_1 - k_2 \cdot \vec{v}_1}{\vec{v}_1 \cdot \vec{v}_2 - \frac{(\vec{v}_2 \cdot \vec{v}_2)(\vec{v}_1 \cdot \vec{v}_1)}{\vec{v}_1 \cdot \vec{v}_2}}, \tag{6.7}$$

los parámetros que debemos introducir en las ecuaciones paramétricas de las líneas para obtener los puntos más cercanos entre ambas.

6.3.3. Gizmo de movimiento en planos XY, XZ y YZ

El movimiento en un plano del gizmo (por ejemplo, el plano XY) utiliza un enfoque geométrico distinto al del movimiento lineal. En lugar de encontrar la distancia mínima entre dos líneas, el sistema calcula la intersección directa entre el rayo del ratón y el plano de movimiento.

El proceso, implementado en las funciones `GizmoControls::update` y `gizmoIntersection`, es el siguiente:

1. **Definición del plano**: Cuando se selecciona un gizmo de plano (por ejemplo, el plano XY), el sistema lo define mediante un punto que contiene (el centro del objeto, `objectCenter`) y su vector normal. Es importante destacar que, por convención en el código, el vector almacenado en `currentGizmo` para un plano es su normal. Por ejemplo, para el plano XY, la normal es el eje Z (0, 0, 1).

2. **Intersección Rayo-Plano**: Con cada movimiento del ratón, se vuelve a castear un rayo desde la cámara (`ro`, `rd`). La función `planeLineIntersection` calcula el punto exacto donde este rayo intersecta el plano infinito definido en el paso anterior. La ecuación para la intersección de una línea y un plano es una fórmula geométrica estándar.

3. **Cálculo del Desplazamiento**: El sistema calcula el vector de desplazamiento (`diffV`) restando el punto de intersección inicial (`firstIntersectionPoint`, calculado en `initDrag`) del punto de intersección actual.

   ```
   glm::vec3 diffV = intersection - this->firstIntersectionPoint;
   ```

4. **Aplicación del Movimiento Restringido**: El desplazamiento `diffV` se suma a la posición inicial del objeto (`initialObjectCenter`) para obtener la nueva posición teórica. Sin embargo, para asegurar que el objeto solo se mueva en el plano seleccionado, se aplica una máscara vectorial. Esta máscara se crea restando el vector normal del plano al vector identidad `glm::vec3(1.0)`.

   - Para el plano XY, la normal es (0, 0, 1). La máscara es (1, 1, 1) - (0, 0, 1) = (1, 1, 0).
   - Para el plano XZ, la normal es (0, 1, 0). La máscara es (1, 1, 1) - (0, 1, 0) = (1, 0, 1).
   - Para el plano YZ, la normal es (1, 0, 0). La máscara es (1, 1, 1) - (1, 0, 0) = (0, 1, 1).

   Al multiplicar la nueva posición por esta máscara, se anula cualquier movimiento en la dirección de la normal, restringiendo el desplazamiento exclusivamente al plano deseado.

Este método garantiza que, aunque el ratón se mueva en un espacio 2D y el rayo viaje en 3D, el objeto seleccionado se deslice perfectamente sobre el plano elegido, proporcionando un control intuitivo y preciso.

Capítulo 7

Implementación de los Shaders

7.1. Introducción

En este capítulo se describe la implementación de los shaders, explicando cómo se genera el código, su funcionamiento, y el proceso mediante el cual se parametriza y escribe el código según la escena y los objetos definidos por el usuario.

7.2. Arquitectura General

Copper utiliza renderizado basado en shaders escritos en WGSL (WebGPU Shading Language), generando el código de manera dinámica a partir de la descripción de la escena. La clase principal responsable de esta tarea es `Coder`, que compone el código shader en tiempo de ejecución según los objetos y operaciones definidos.

7.3. Funcionamiento General

El ciclo de generación y uso de shaders sigue el siguiente flujo:

1. El usuario define objetos geométricos (esferas, cajas, conos, cilindros) y operaciones booleanas (unión, intersección, sustracción suave, etc.) a través de la interfaz gráfica.
2. Cada vez que la escena cambia, la clase `Coder` reconstruye el código shader combinando funciones SDF (Signed Distance Function) para cada objeto y operación.
3. El código generado incluye la lógica para renderizado normal y picking (selección de objetos).

4. El shader se compila y se utiliza en la pipeline gráfica de WebGPU.

7.4. Generación Dinámica del Código Shader

La clase `Coder` mantiene una lista de objetos y sus propiedades (tipo, posición, tamaño, color, operación). Cuando la escena cambia, el método `generateShaderCode()` produce el código WGSL correspondiente. El proceso es el siguiente:

- Para cada objeto, se genera una función SDF específica que calcula la distancia y el color.
- Se combinan las SDFs utilizando las operaciones booleanas indicadas.
- Se añade lógica para picking, de modo que cada objeto pueda ser seleccionado mediante raycasting.
- Si hay objetos auxiliares como gizmos, se añaden funciones SDF para estos elementos.
- El código generado se concatena junto con una base estándar que define la estructura de los shaders, funciones de iluminación, ray marching, etc.

El siguiente fragmento ilustra la parte del código shader que se puede generar dinámicamente, en concreto los sdf de los objetos:

```
fn sdf(pos: vec3<f32>) -> DistanceColor {
    var result = DistanceColor(1e6, vec3<f32>(0.0, 0.0, 0.0),
    -1);
    var sphere0: DistanceColor = sdf_sphere(pos, uniforms.
    position, uniforms.size[0], uniforms.color, 0);
    result = opSmoothUnion(result, sphere0, 0.6);
    // ... otros objetos y operaciones
    return result;
}
```

Listing 7.1: Fragmento de código WGSL generado

7.5. Personalización y Parametrización

La generación del shader se adapta a los parámetros definidos por el usuario:

- **Parámetros geométricos**: posición, tamaño y color de cada objeto se insertan directamente en el código WGSL, en el caso de ser objeto seleccionado se obtienen mediante los `uniforms`.

- **Operaciones**: el tipo de operación (unión, intersección, sustracción suave) determina qué función de combinación se utiliza.
- **Picking**: se genera una función especial que permite identificar qué objeto ha sido seleccionado por el usuario.
- **Gizmos**: si el usuario selecciona un objeto, se generan SDFs adicionales para los gizmos de manipulación.

7.6. Funciones SDF y Operaciones Booleanas

Cada tipo de objeto tiene su propia función SDF:

- `sdf_sphere`: calcula la distancia de un punto al borde de una esfera.
- `sdf_box`: calcula la distancia a una caja.
- `sdf_cone`, `sdf_cylinder`: para conos y cilindros.
- `sdf_plane`: para planos.

Las combinaciones entre objetos se realizan con operadores booleanos suaves:

```
1 fn opSmoothUnion(d1: DistanceColor, d2: DistanceColor, k: f32)
    -> DistanceColor { ... }
2 fn opSmoothIntersect(d1: DistanceColor, d2: DistanceColor, k:
    f32) -> DistanceColor { ... }
3 fn opSmoothSubtract(d1: DistanceColor, d2: DistanceColor, k:
    f32) -> DistanceColor { ... }
```

Listing 7.2: Operadores booleanos suaves

7.7. Renderizado y Ray Marching

El sistema de renderizado se basa en la técnica de **ray marching** implementada en el shader principal. Este método permite recorrer la escena pixel a pixel, determinando la distancia a la superficie más cercana en cada dirección de visión. El proceso está diseñado para trabajar con funciones de distancia (SDF, *Signed Distance Function*) que representan primitivas geométricas y operaciones booleanas entre ellas.

En cada fragmento (píxel), el shader realiza los siguientes pasos:

- Se construye el rayo de visión a partir de la posición de la cámara y la dirección correspondiente al píxel actual, utilizando los datos del buffer de uniformes (`Uniforms`) y la matriz de vista-proyección (`mvp_matrix`). La dirección del rayo se normaliza y se ajusta según el field-of-view (FOV) y el aspecto de la ventana.

- El rayo se avanza iterativamente en la escena, sumando en cada paso la distancia mínima a la superficie más cercana, obtenida evaluando la función SDF combinada de todos los objetos (`sdf_combined`). Este proceso se repite hasta que la distancia a la superficie es menor que un umbral mínimo (`MIN_DISTANCE`) o se alcanza el límite máximo de pasos (`MAX_MARCHING_STEPS`) o de distancia (`MAX_DISTANCE`), como se observa en la función `ray_march()` del shader generado en `Coder.cpp`.
- Cuando se detecta una intersección con una superficie, se calcula la normal local en ese punto mediante diferencias finitas (`calculate_normal()`), evaluando la variación de la SDF en las tres direcciones espaciales. Esta aproximación utiliza pequeñas perturbaciones (`eps`) y resta los valores de la SDF para obtener la derivada parcial en cada eje.
- El color final se determina aplicando el modelo de iluminación Blinn-Phong sobre el punto de intersección, normal calculada y dirección de visión.

7.7.1. Cálculo de Normales

El cálculo de la normal en Copper se realiza en el shader mediante la función `calculate_normal(p)`, donde p es el punto de la superficie detectado por el ray marching. La normal se calcula como:

$$\vec{n} = \text{normalize}(\begin{bmatrix} \text{sdf\_combined}(p+\epsilon_x) - \text{sdf\_combined}(p-\epsilon_x) \\ \text{sdf\_combined}(p+\epsilon_y) - \text{sdf\_combined}(p-\epsilon_y) \\ \text{sdf\_combined}(p+\epsilon_z) - \text{sdf\_combined}(p-\epsilon_z) \end{bmatrix}) \quad (7.1)$$

donde ϵ es un pequeño desplazamiento en cada eje. De esta forma, se obtiene la dirección normalizada perpendicular a la superficie. Esta técnica es estándar en renderizado SDF y permite obtener normales suaves y precisas sin necesidad de almacenar datos adicionales.

7.7.2. Modelo de Iluminación Blinn-Phong

Copper emplea el modelo de iluminación Blinn-Phong, implementado en la función `blinn_phong_lighting()` del shader. Este modelo calcula el color visible en cada punto de la superficie atendiendo a tres componentes:

- **Ambiental**: Representa la luz ambiental global, definida en Copper como una fracción constante de la luz blanca. (`ambient = 0.2 * light_color`; donde `light_color` es generalmente $(1.0, 1.0, 1.0)$).
- **Difusa**: Calculada según el ángulo entre la normal de la superficie y la dirección de la luz (`light_dir`), usando el producto escalar. Cuanto más alineados estén, mayor será la intensidad difusa.

- **Especular**: Usa el vector `halfway` (media entre la dirección de la luz y la dirección de la vista) para calcular el brillo especular, elevado a una potencia para simular materiales brillantes.

El color final en cada punto se calcula como:

$$\text{color\_final} = \text{base\_color} \times (\text{ambient} + (1.0 + \text{shadow}) \times (\text{diffuse} + \text{specular}))$$

donde `shadow` es el factor de atenuación por sombras, calculado mediante un nuevo ray marching desde el punto de intersección hacia la luz, comprobando si existen obstáculos en ese trayecto (`calculate_shadow()`).

7.7.3. Cálculo de Sombra

La sombra en Copper se calcula mediante un segundo proceso de ray marching, iniciado desde el punto de intersección en la dirección de la luz. En cada paso, se evalúa la distancia a la superficie más cercana, y se acumula el valor mínimo relativo a la distancia recorrida, lo que permite obtener sombras suaves y realistas. Si el rayo encuentra una superficie antes de alcanzar la fuente de luz, la sombra se intensifica.

Esto es optimizable, ya que se podría añadir el ray marching de las sombras al proceso de ray marching principal, evitando así la necesidad de un segundo recorrido por la escena.

7.8. Picking en el shader

La técnica de **picking** en Copper permite identificar el objeto SDF seleccionado por el usuario, usando un shader especializado que recorre la escena y devuelve el identificador (`id`) del objeto más próximo al rayo lanzado desde el cursor.

El proceso es el siguiente:

1. Al pulsar el botón izquierdo del ratón, se genera una pipeline de picking y se ejecuta el shader picking.

2. En el shader, se calcula el rayo a partir de la posición del cursor, transformada a coordenadas normalizadas y ajustada por la matriz de vista de cámara.

3. Se realiza ray marching sobre la escena, evaluando la función SDF y almacenando el `id` del objeto más cercano.

4. El resultado se escribe en un buffer (una textura de un solo canal entero), que se lee desde la CPU para determinar el objeto seleccionado.

El núcleo del shader de picking se encuentra en la función `fragmentPickingMain`:

```
@fragment
fn fragmentPickingMain(in: VertexOutput) -> @location(0) i32{
    let cam = transpose(uniforms.mvp_matrix);
    let right = cam[0].xyz;
    let up = cam[1].xyz;
    let forward = cam[2].xyz;
    let eye = cam[0].w * right + cam[1].w * up + cam[2].w *
    forward;
    let m = uniforms.mouse_position * 2.0 - vec2<f32>(1.0, 1.0)
    ;
    let mouse_uv = vec2<f32>(m.x, -m.y / uniforms.aspect_ratio)
    ;
    let rd = normalize(mouse_uv.x * right + mouse_uv.y * up +
    FOV * forward);
    let ro = eye;
    let dc = ray_march_picking(ro, rd);
    return dc.id;
}
```

Listing 7.3: Picking shader fragment

La función `ray_march_picking` recorre la escena utilizando la función SDF de picking, que para cada objeto evalúa la distancia y el identificador:

```
fn ray_march_picking(ro: vec3<f32>, rd: vec3<f32>) ->
    DistanceColor {
    var total_distance: f32 = 0.0;
    for (var i: i32 = 0; i < MAX_MARCHING_STEPS &&
    total_distance < MAX_DISTANCE; i++) {
        let pos = ro + rd * total_distance;
        let dc = sdf_picking(pos);
        if (dc.distance < MIN_DISTANCE) {
            return dc;
        }
        total_distance += dc.distance;
    }
    return DistanceColor(1e6, vec3<f32>(0.0), -1);
}
```

Listing 7.4: Ray marching picking loop

La función `sdf_picking` está generada dinámicamente y recorre todos los objetos, devolviendo el identificador del más cercano:

```
fn sdf_picking(pos: vec3<f32>) -> DistanceColor {
    var result = DistanceColor(1e6, vec3<f32>(0.0, 0.0, 0.0),
    -1);
    // Para cada objeto:
    // var sphere0: DistanceColor = sdf_sphere(...);
    // ...
    if (sphere0.distance < result.distance) {
        result = sphere0;
    }
    // ...
```

```
10      return result;
11  }
```

Listing 7.5: SDF picking para todos los objetos

El pipeline de picking escribe el `id` del objeto seleccionado en la textura de picking, que luego es leída por la CPU para actualizar el estado de selección en el sistema.

Capítulo 8

Funcionalidades adicionales implementadas

8.1. 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 y su integración en los distintos módulos.

8.2. 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`).

8.2.1. 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.

8.2.2. 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.

8.2.3. 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);
}
```

8.3. 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.

8.3.1. Pruebas de rendimiento con `addTest`

Se han realizado pruebas de rendimiento utilizando el método `addTest` de `Coder`, el cual añade un conjunto de esferas a la escena para evaluar el comportamiento del sistema bajo diferentes cargas de objetos. Los resultados obtenidos, variando el número de objetos y la presencia del plano de suelo, son los siguientes:

- 20 objetos:
  Sin suelo: 240 FPS
  Con suelo: 120 FPS
- 50 objetos:
  Sin suelo: 180 FPS
  Con suelo: 60 FPS
- 100 objetos:
  Sin suelo: 100 FPS
  Con suelo: 30 FPS
- 200 objetos:
  Sin suelo: 25 FPS
  Con suelo: 10 FPS

Estos resultados muestran cómo el rendimiento (FPS) se ve afectado por el número de objetos y la presencia del suelo en la escena. Se observa una disminución progresiva de los FPS a medida que aumenta el número

de objetos, siendo más acusada cuando se activa el plano de suelo, esto es causado principalmente por la forma actual de calcular las sombras, ya que el suelo tiene una gran superficie y cada intersección requiere un cálculo adicional.

Las especificaciones del sistema utilizado para realizar estas pruebas son las siguientes:

- CPU: AMD Ryzen 5 5600X
- GPU: AMD Radeon RX 6800
- RAM: 32 GB
- Sistema operativo: Arch Linux

Capítulo 9

Conclusiones y mejoras futuras

En este documento se ha presentado el sistema Copper, un entorno de desarrollo para la creación y manipulación de escenas 3D basadas en Signed Distance Fields (SDF). A lo largo de los capítulos, se ha detallado la arquitectura, implementación y funcionamiento de sus principales componentes, incluyendo la gestión de controles, la implementación de shaders y el sistema de picking.

Entre las conclusiones más relevantes, se destacan las siguientes:

- La utilización de SDF permite una representación eficiente y flexible de la geometría 3D, facilitando operaciones complejas como la unión, intersección y sustracción de objetos.
- La integración de herramientas de interfaz gráfica (ImGui) y gizmos proporciona una experiencia de usuario intuitiva y directa para la manipulación de la escena.
- El sistema de picking implementado permite seleccionar objetos de manera precisa, incluso en escenas complejas, mejorando la interactividad del entorno.

En cuanto a mejoras futuras, se proponen las siguientes líneas de trabajo:

- Ampliación de las capacidades de edición en tiempo real, permitiendo a los usuarios modificar más propiedades y mayor variedad de objetos.
- Implementación de un sistema de materiales y texturas más avanzado, que permita una mejor representación visual de los objetos en la escena.
- Mejora del sistema de renderizado, optimizando el uso de la GPU y añadiendo mejoras como el antialiasing.

- Estudiar la portabilidad del sistema a web, debido a la elección de tecnologías como WebGPU es un área de desarrollo evidente.
- Añadir un sistema de exportación de la escena a formatos de malla estándar (como OBJ o GLTF) para facilitar la interoperabilidad con otras herramientas y flujos de trabajo. Con este objetivo, se ha estudiado la aplicación del algoritmo Marching Cubes, que permite convertir las representaciones basadas en SDF a mallas poligonales compatibles con estos formatos.

Estas mejoras contribuirán a consolidar a Copper como una herramienta potente y versátil para la creación de escenas 3D, ampliando sus posibilidades y mejorando la experiencia del usuario.

Bibliografía

[Ric73] A. Ricci. "A Constructive Geometry for Computer Graphics". En: *The Computer Journal* 2 (mayo de 1973), págs. 157-160. DOI: http://dx.doi.org/10.1093/comjnl/16.2.157.

[Whi80] Turner Whitted. "An improved illumination model for shaded display". En: *Communications of the ACM* 23.6 (1980), págs. 343-349. DOI: 10.1145/358876.358882.

[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.

[Ahn08] Song Ho Ahn. *OpenGL Projection Matrix*. 2008. URL: http://www.songho.ca/opengl/gl_projectionmatrix.html.

[Quí08] Iñigo Quílez. *Raymarching Distance Fields*. Accessed: 2025-08-11. 2008. URL: https://iquilezles.org/articles/raymarchingdf/.

[PJH16] Matt Pharr, Wenzel Jakob y Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. 3rd. Morgan Kaufmann, 2016. ISBN: 978-0128006450. URL: https://www.pbrt.org.

[Cor24] Omar Cornut. *Dear ImGui*. Accessed: 2025-08-11. 2024. URL: https://github.com/ocornut/imgui.

[Cre24] G-Truc Creation. *OpenGL Mathematics (GLM)*. Accessed: 2025-08-11. 2024. URL: `https://github.com/g-truc/glm`.

[Dev24] GLFW Developers. *GLFW Documentation*. Accessed: 2025-08-11. 2024. URL: `https://www.glfw.org/docs/latest/`.

[GC24] Google y Chromium Community. *Dawn: WebGPU implementation*. Accessed: 2025-08-11. 2024. URL: `https://github.com/google/dawn`.

[Kit24] Kitware. *CMake Documentation*. Accessed: 2025-08-11. 2024. URL: `https://cmake.org/cmake/help/latest/`.

Apéndice A

Escenas

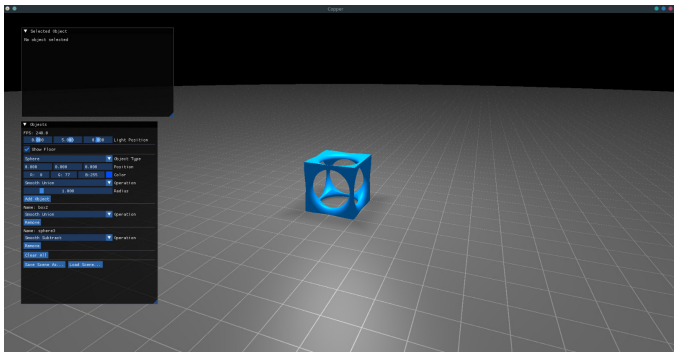

Figura A.1: Combinación de cubo y esfera mediante SDF.

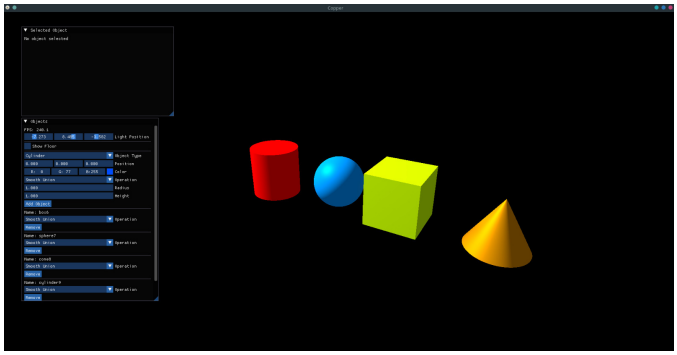

Figura A.2: Escena con objetos sin plano de referencia.

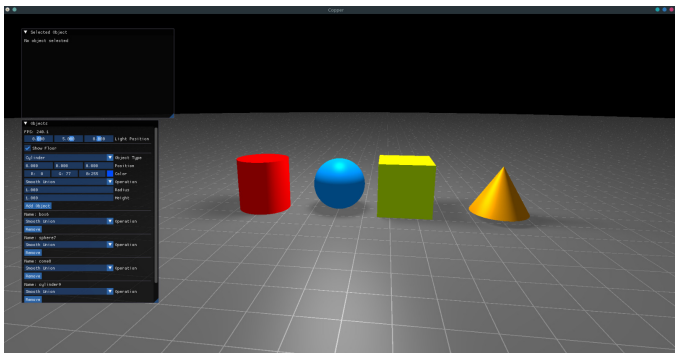

Figura A.3: Escena con objetos y plano de referencia activado.

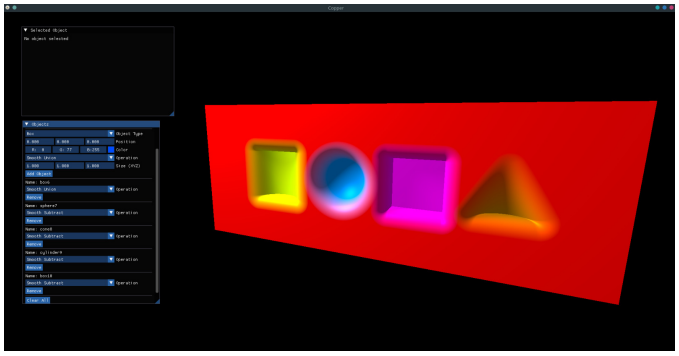

Figura A.4: Ejemplo de operación smooth subtract entre objetos.

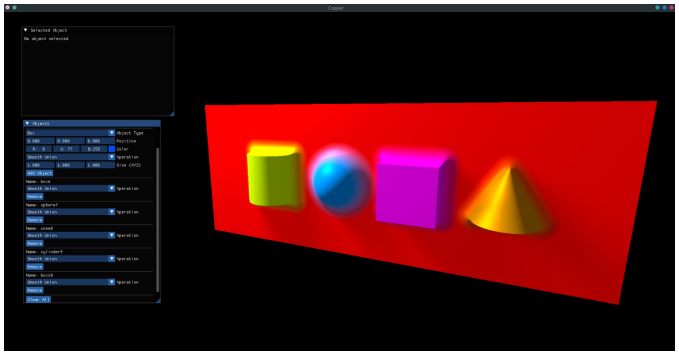

Figura A.5: Ejemplo de operación smooth union entre objetos.

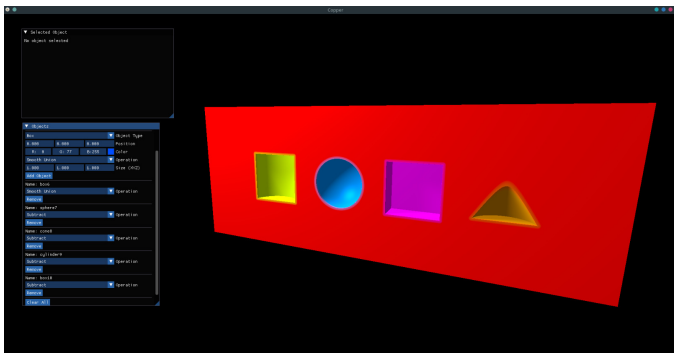

Figura A.6: Ejemplo de operación de sustracción entre objetos.

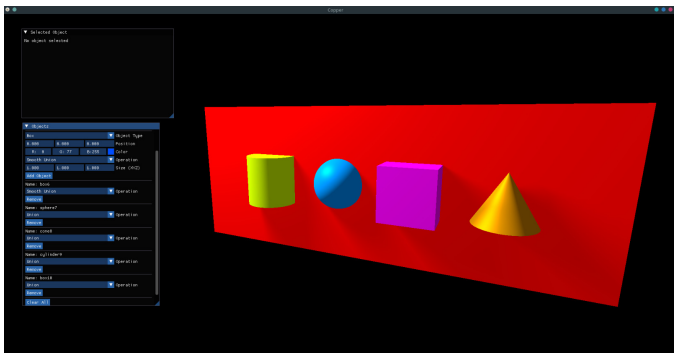

Figura A.7: Ejemplo de operación de unión entre objetos.

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..