



# UNIVERSIDAD DE GRANADA

TRABAJO FIN DE MÁSTER  
MASTER EN DESARROLLO DE SOFTWARE

## Interfaz de programación para una API de rasterización moderna

---

Diseño y desarrollo de un motor de render ligero basado en Vulkan para  
aplicaciones interactivas 3D

**Autor**

Santiago Carbó García

**Director/es**

Carlos Ureña Almagro



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE TELECOMUNICACIÓN

Granada, 6 de septiembre de 2025



# Interfaz de programación para una API de rasterización moderna

---

Diseño y desarrollo de un motor de render ligero basado en Vulkan para  
aplicaciones interactivas 3D

**Autor**

Santiago Carbó García

**Director/es**

Carlos Ureña Almagro



# **Interfaz de programación para una API de rasterización moderna: Diseño y desarrollo de un motor de render ligero basado en Vulkan para aplicaciones interactivas 3D**

Santiago Carbó García

**Palabras clave:** Visualización 3D interactiva, Motor de renderizado, Vulkan, GPU, API de rasterización, Cauce gráfico, Aplicaciones multi-hebra, Paralelización de tareas, Alto rendimiento gráfico, Arquitectura software

## **Resumen**

El presente proyecto tiene como objetivo principal el análisis, diseño, desarrollo, implementación y validación de una componente de software orientada a facilitar la creación de aplicaciones interactivas de visualización 3D sobre unidades de procesamiento gráfico (GPUs) modernas, utilizando la API de rasterización Vulkan, desarrollada y mantenida por el consorcio Khronos. Vulkan se caracteriza por ofrecer un alto rendimiento y un control exhaustivo sobre los recursos gráficos, lo que la convierte en una herramienta idónea para la programación de motores de renderizado de última generación; sin embargo, esta misma flexibilidad implica una gran complejidad en la configuración y gestión correcta del cauce gráfico. Con el fin de abordar esta problemática, en este proyecto se ha desarrollado un motor de render ligero que proporciona una interfaz de programación compuesta por funciones y clases diseñadas para abstraer gran parte de la complejidad inherente a Vulkan, facilitando así la implementación de aplicaciones 3D interactivas. Para validar el diseño de esta interfaz se ha construido una aplicación de prueba capaz de cargar modelos tridimensionales complejos, gestionar parámetros de visualización en tiempo real y permitir la interacción directa con la escena. Además, como parte del estudio experimental, se ha evaluado el rendimiento de Vulkan en arquitecturas multi-core modernas, configurando el cauce gráfico para explotar aplicaciones multi-hebra y comparando su eficiencia frente a implementaciones de una única hebra, con el fin de cuantificar las mejoras potenciales en escenarios de ejecución concurrente. Este proyecto no solo sienta una base práctica para crear aplicaciones 3D con Vulkan, sino que también se convierte en una oportunidad real para entender mejor cómo se conectan el diseño de la arquitectura de software, la paralelización de tareas y el rendimiento gráfico. En otras palabras, más allá de ser un simple motor de renderizado, el trabajo funciona como un espacio de experimentación que ayuda a ver de forma clara el impacto de las decisiones técnicas en la eficiencia y en la calidad de la visualización en tiempo real.

**Repositorio del código:** <https://github.com/santiago-carbo/VulkanAPI>

# Programming Interface for a Modern Rasterization API: Design and Development of a Lightweight Rendering Engine Based on Vulkan for Interactive 3D Applications

Santiago Carbó García

**Keywords:** Interactive 3D visualization, Rendering Engine, Vulkan, GPU, Rasterization API, Graphics pipeline, Multi-threaded applications, Task parallelization, High-performance graphics, Software architecture

## Abstract

The main objective of this project is the analysis, design, development, implementation, and validation of a software component aimed at facilitating the creation of interactive 3D visualization applications on modern Graphics Processing Units (GPUs), using the Vulkan rasterization API developed and maintained by the Khronos consortium. Vulkan is characterized by delivering high performance and providing fine-grained control over graphical resources, making it an ideal tool for programming next-generation rendering engines. However, this same flexibility entails significant complexity in the correct configuration and management of the graphics pipeline. To address this challenge, a lightweight rendering engine has been developed in this project, providing a programming interface composed of functions and classes designed to abstract much of the inherent complexity of Vulkan, thereby simplifying the implementation of interactive 3D applications. To validate the design of this interface, a test application was built, capable of loading complex three-dimensional models, managing visualization parameters in real time, and enabling direct interaction with the scene. Furthermore, as part of the experimental study, the performance of Vulkan on modern multi-core architectures has been evaluated by configuring the graphics pipeline to exploit multi-threaded applications and comparing its efficiency against single-threaded implementations, in order to quantify potential improvements in concurrent execution scenarios. This project not only establishes a practical foundation for creating 3D applications with Vulkan, but also provides a real opportunity to better understand the interplay between software architecture design, task parallelization, and graphical performance. In other words, beyond being a simple rendering engine, this work serves as an experimental framework that clearly demonstrates the impact of technical decisions on both efficiency and the quality of real-time visualization.

**Source code repository:** <https://github.com/santiago-carbo/VulkanAPI>

---

Yo, **Santiago Carbó García**, alumno del **MÁSTER DE DESARROLLO DEL SOFTWARE**, con DNI **77561011C**, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Máster en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: *Santiago Carbó García*

Granada a 6 de septiembre de 2025.





---

D. **Carlos Ureña Almagro**, Profesor del Área de Lenguajes y Sistemas Informáticos del Departamento Lenguajes y Sistemas Informáticos de la Universidad de Granada.

**Informa:**

Que el presente trabajo, titulado *Interfaz de programación para una API de rasterización moderna, Diseño y desarrollo de un motor de render ligero basado en Vulkan para aplicaciones interactivas 3D*, ha sido realizado bajo su supervisión por **Santiago Carbó García**, 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 6 de septiembre de 2025.

**El director:**

**Carlos Ureña Almagro**



---

Yo, **Santiago Carbó García**, alumno del **MÁSTER DE DESARROLLO DEL SOFTWARE** de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI **77561011C**, declaro explícitamente que el trabajo presentado es original, entendido en el sentido que no he utilizado ninguna fuente sin citarla debidamente.

Fdo: **Santiago Carbó García**

Granada, 6 de septiembre de 2025.



# Agradecimientos

Es difícil escribir una sección de agradecimientos porque hay muchísimas personas que han aportado tanto a mi vida que resulta complicado priorizar y cuantificar el valor que cada una de ellas ha tenido. Por ello, quiero dejar claro desde el principio que las personas que menciono a continuación no se rigen por ningún tipo de escala: su valor es incalculable.

Quiero agradecer a mi abuela por todo el amor y apoyo que me dio siempre. Era una persona que irradiaba una luz capaz de alegrar a cualquiera y de hacerte sentir que no existía problema en el mundo. También quiero agradecer a mi hermana por ser el mayor apoyo de mi vida. Le doy gracias al cielo por la suerte de tener una hermana tan buena persona.

Agradezco a mi padre y a mi madre por haberme criado y haber sido mi apoyo fundamental desde el primer día en que nací. Mi vida no sería tan privilegiada si no fuera por ellos. También agradezco a mi abuelo Mateo, a mi abuelo Pepe y a mi tía María por haberme dado cariño y esperanza toda la vida.

Es un orgullo haber nacido rodeado de tantas buenas personas.



# Índice general

<b>1. Introducción</b>	<b>1</b>
1.1. Motivación . . . . .	1
1.2. Objetivos . . . . .	2
1.2.1. Objetivos Generales . . . . .	2
1.2.2. Objetivos Específicos . . . . .	2
1.3. Estructura del trabajo . . . . .	3
<b>2. Estado del arte</b>	<b>5</b>
2.1. Introducción . . . . .	5
2.2. APIs gráficas modernas . . . . .	5
2.3. Comparativas y estudios de rendimiento . . . . .	5
2.4. Aplicaciones y guías técnicas . . . . .	6
2.5. Motores de render y <i>game engines</i> ligeros . . . . .	6
2.6. Estudios comparativos de arquitecturas . . . . .	6
2.7. Motores para <i>serious games</i> y visualización interactiva . . . . .	7
2.8. Vulkan en entornos multi-hebra . . . . .	7
<b>3. Desarrollo del TFM</b>	<b>9</b>
3.1. Arquitectura general del motor . . . . .	9
3.1.1. Casos de uso del motor . . . . .	12
3.1.2. Diseño multi-hebra frente a versión monohilo . . . . .	13
3.1.3. Librería de hebras utilizada . . . . .	14
3.2. Gestión de la GPU con Vulkan . . . . .	14
3.3. Pipeline gráfico y shaders . . . . .	16
3.4. Gestión de recursos . . . . .	18
3.5. Estructura de clases del motor (UML) . . . . .	20
3.5.1. Diagrama de paquetes . . . . .	20
3.5.2. Diagrama de clases . . . . .	20
3.5.3. Diagrama de secuencia . . . . .	20
3.5.4. Resumen de las clases . . . . .	20
3.5.5. Fragmentos breves de código . . . . .	23
<b>4. Resultados y discusión</b>	<b>25</b>
4.1. Benchmark de escenas representativas . . . . .	25
4.2. Escalabilidad CPU-GPU . . . . .	26
4.3. Consumo energético y eficiencia . . . . .	26

<b>5. Planificación y gestión del proyecto</b>	<b>29</b>
5.1. Metodología y ciclo de vida . . . . .	29
5.2. Herramientas y plataformas utilizadas para el desarrollo del proyecto . . . . .	29
5.3. Costes (tiempo y recursos) . . . . .	30
5.4. Gestión del proyecto. Planificación temporal. Diagrama de Gantt . . . . .	30
<b>6. Innovación</b>	<b>33</b>
6.1. Estudio del Mercado . . . . .	33
6.1.1. Segmentos de Clientes . . . . .	33
6.1.2. Competidores . . . . .	33
6.2. Propuesta de Valor . . . . .	34
6.3. Recursos Clave . . . . .	34
6.4. Actividades Clave . . . . .	34
6.5. Socios y Alianzas Clave . . . . .	34
6.6. Estructura de Costes Prevista . . . . .	34
6.7. Vías de Ingresos . . . . .	35
6.8. Canales de Distribución . . . . .	35
6.9. Relación con los Clientes . . . . .	35
6.10. Descripción del Producto Mínimo Viable . . . . .	35
6.11. Roadmap de Desarrollo . . . . .	35
6.12. Aspectos Legales . . . . .	36
<b>7. Manual de usuario</b>	<b>37</b>
7.1. Requisitos del sistema . . . . .	37
7.2. Instalación . . . . .	37
7.3. Ejecución . . . . .	38
7.4. Interacción con la aplicación . . . . .	38
7.5. Solución de problemas . . . . .	38
7.6. Extensiones futuras . . . . .	38
<b>8. Conclusiones</b>	<b>39</b>
<b>Bibliografía</b>	<b>43</b>



# Índice de figuras

3.1. Arquitectura general del motor gráfico desarrollado. . . . .	10
3.2. Flujo del ciclo de render en el motor gráfico desarrollado. . . . .	11
3.3. Etapas principales del pipeline gráfico en Vulkan. . . . .	17
3.4. Diagrama de paquetes UML. . . . .	20
5.1. Planificación temporal del proyecto (febrero – agosto de 2025). . . . .	31



# Índice de tablas

4.1. Resultados de FPS en escenas representativas . . . . .	26
4.2. Escalabilidad en CPU y GPU . . . . .	26
4.3. Consumo energético en escenas representativas . . . . .	27
5.1. Dedicación por mes (febrero – 15 de agosto de 2025). . . . .	30



# Capítulo 1

## Introducción

El desarrollo de aplicaciones de visualización 3D interactiva constituye actualmente uno de los ámbitos más relevantes dentro de la ingeniería del software, no sólo por su impacto en la industria del entretenimiento digital, sino también por su papel en campos como la simulación, el diseño asistido por ordenador, la ingeniería o la realidad virtual o aumentada. La capacidad de representar escenas tridimensionales en tiempo real con un alto nivel de fidelidad visual es clave para ofrecer experiencias inmersivas.

El avance de las unidades de procesamiento gráfico (GPUs) ha sido determinante en este contexto. Las arquitecturas modernas, caracterizadas por su gran capacidad de cómputo paralelo, permiten gestionar cargas de trabajo masivas, siempre que el software sea capaz de aprovechar de forma eficiente los múltiples hilos de ejecución. Esta tendencia ha impulsado la necesidad de motores de renderizado más flexibles y capaces de adaptarse a las exigencias del hardware.

En respuesta a esta evolución, la API Vulkan, desarrollada por el consorcio Khronos, se ha posicionado como una de las tecnologías clave para la programación gráfica a bajo nivel. Su diseño reduce la sobrecarga de abstracción y ofrece un control detallado sobre los recursos gráficos, permitiendo un rendimiento significativamente superior frente a APIs más tradicionales. Sin embargo, esta flexibilidad conlleva una cierta complejidad: la configuración inicial del cauce gráfico y la correcta gestión de los recursos suponen una tarea técnica que puede limitar su adopción en proyectos donde el tiempo de desarrollo es crítico.

Este proyecto se sitúa precisamente en ese punto de equilibrio entre el potencial de Vulkan y la necesidad de herramientas que reduzcan la barrera de entrada, ofreciendo un entorno más accesible para la creación de aplicaciones 3D interactivas de alto rendimiento.

### 1.1. Motivación

La realización de este proyecto surge de la convergencia entre mi interés personal en la programación gráfica y una necesidad real en el campo de la visualización 3D. Desde una perspectiva individual, trabajar en un motor de renderizado desde cero constituye una oportunidad para profundizar en áreas clave de la ingeniería del software como la gestión eficiente de hardware gráfico, la concurrencia y el diseño de interfaces que combinen rendimiento y facilidad de uso. Este tipo de desarrollo implica un aprendizaje integral, donde teoría y práctica se combinan para crear un sistema funcional.

En el plano técnico, el proyecto responde a una problemática bien conocida en la comunidad de desarrollo gráfico: Vulkan, a pesar de ser una API muy potente, presenta una complejidad que frena su adopción en muchos entornos. La creación de una capa de abstracción que permita explotar sus capacidades sin obligar al desarrollador a gestionar cada detalle de bajo nivel puede tener un impacto directo en la forma en que se construyen motores de render en proyectos interactivos.

Asimismo, el trabajo se enmarca en un contexto en el que la explotación de arquitecturas multi-core y la paralelización de tareas son prioritarias para obtener un rendimiento óptimo. Explorar cómo la programación gráfica puede adaptarse mejor a estos sistemas no solo resulta relevante desde el punto de vista académico, sino que también ofrece información valiosa para el desarrollo de soluciones en la industria.

En conjunto, este proyecto combina una componente personal de superación técnica con una aportación práctica al campo del software gráfico, buscando reducir la complejidad de las herramientas actuales y facilitar la creación de aplicaciones 3D interactivas.

## 1.2. Objetivos

Los objetivos de este proyecto se establecen como una guía clara para orientar el desarrollo y servir como referencia en la evaluación de los resultados obtenidos. Se distinguen entre objetivos generales, que marcan la meta principal del trabajo, y objetivos específicos, que desglosan de forma concreta las tareas y resultados necesarios para alcanzar dicha meta. Su cumplimiento será revisado y contrastado en el Capítulo 8: [Conclusiones](#).

### 1.2.1. Objetivos Generales

1. Diseñar e implementar una aplicación software que simplifique el desarrollo de aplicaciones de visualización 3D interactivas mediante una interfaz de programación que actúe como capa de abstracción sobre una API de rasterización moderna.
2. Validar la utilidad y el rendimiento de la aplicación mediante el desarrollo de una aplicación práctica que permita evaluar sus capacidades en un entorno funcional.

### 1.2.2. Objetivos Específicos

1. Analizar los requerimientos técnicos y las características de la API Vulkan para establecer las bases de la interfaz a desarrollar.
2. Diseñar una arquitectura modular que permita integrar funciones y clases para la gestión eficiente del cauce gráfico y los recursos asociados.
3. Implementar un motor de render ligero que proporcione herramientas de alto nivel para la creación de aplicaciones interactivas en 3D.
4. Desarrollar una aplicación de prueba que utilice la interfaz creada, incorporando carga de modelos complejos y manipulación de parámetros de visualización.

5. Evaluar el comportamiento de la aplicación en entornos multi-core, comparando configuraciones de hebra única frente a multi-hebra para analizar diferencias de rendimiento.
6. Documentar el diseño, la implementación y los resultados obtenidos para garantizar la reproducibilidad y servir como base de trabajos futuros.

### 1.3. Estructura del trabajo

El contenido de este proyecto se organiza en los siguientes capítulos:

1. [Introducción](#), donde se presenta el contexto general del proyecto e incluye la motivación, los objetivos generales y específicos, así como la descripción de la propia estructura del documento.
2. [Estado del arte](#), capítulo éste destinado al análisis de las tecnologías y metodologías existentes en el ámbito de la visualización 3D interactiva, prestando especial atención a las APIs de rasterización modernas como Vulkan y a los motores de render ligeros.
3. [Desarrollo del TFM](#). En este capítulo detallo el diseño de la arquitectura, la implementación de la interfaz de programación y las decisiones técnicas adoptadas para la construcción del motor de render.
4. [Resultados y discusión](#). En este capítulo muestro los resultados obtenidos, incluyendo las pruebas realizadas con la aplicación de validación y el análisis de rendimiento en entornos multi-core, comparando configuraciones de hebra única y multi-hebra.
5. [Planificación y gestión del proyecto](#). En este capítulo se documenta la planificación y gestión del proyecto: metodología y ciclo de vida, herramientas y plataformas empleadas, estimación de costes y sostenibilidad, así como la planificación temporal con su diagrama de Gantt y los criterios de validación utilizados.
6. [Innovación](#). Capítulo dedicado al encaje potencial del proyecto en un contexto empresarial: estudio de mercado, segmentos de clientes, análisis de competidores, propuesta de valor, recursos y actividades clave, socios, alianzas, estructura de costes, vías de ingresos, canales de distribución, relación con clientes, definición del MVP, *roadmap* y consideraciones legales.
7. [Manual de usuario](#). Capítulo orientado al uso de la aplicación: requisitos, despliegue, ejecución y controles de interacción, descripción de los paneles (Dear ImGui), parámetros configurables y una breve guía de resolución de problemas frecuentes.
8. [Conclusiones](#). Por último, en el capítulo de conclusiones se sintetizan los resultados más relevantes, evaluando asimismo el grado de cumplimiento de los objetivos planteados y planteando posibles líneas de mejora y trabajos futuros.

Estos capítulos se complementan con los correspondientes índices:

- Índice de Figuras (pagina III)
- Índice de Tablas (página V)





## Capítulo 2

# Estado del arte y trabajos previos

### 2.1. Introducción

El desarrollo de aplicaciones de visualización 3D interactiva ha experimentado en la última década una transición hacia modelos de programación cada vez más cercanos al hardware. Esta tendencia ha estado impulsada por la necesidad de aprovechar de forma más eficiente las arquitecturas *multi-core* y las capacidades paralelas de las *GPUs* modernas. En este contexto, el papel de las *APIs* gráficas de bajo nivel como Vulkan, DirectX 12 y Metal se ha vuelto central, ya que ofrecen a los desarrolladores un control explícito sobre el flujo de renderizado y la gestión de los recursos. Si bien *APIs* tradicionales como OpenGL han sido durante años el estándar para la programación gráfica en tiempo real, su modelo de ejecución *single-threaded* y la abstracción de la gestión interna de recursos limitan el potencial de escalado en hardware más moderno. Las nuevas *APIs* como Vulkan han introducido un paradigma diferente, otorgando al desarrollador control detallado sobre la creación de *command buffers*, sincronización, asignación de memoria y distribución del trabajo entre hilos de CPU [1–4].

### 2.2. APIs gráficas modernas

La introducción de Vulkan por el consorcio Khronos supuso una evolución significativa frente a modelos anteriores como OpenGL. Vulkan está diseñado para minimizar la sobrecarga de la CPU, habilitar la ejecución multi-hilo real y proporcionar un modelo de programación más predecible para la GPU. Estas características resultan especialmente relevantes en entornos donde la latencia y el rendimiento son críticos, como videojuegos, simuladores interactivos o aplicaciones en tiempo real [5, 6].

En comparación con DirectX 12 y Metal, Vulkan ofrece la ventaja de ser una *API* multiplataforma y abierta, lo que facilita su adopción en proyectos que requieren compatibilidad con múltiples sistemas operativos [7, 8]. Sin embargo, esta flexibilidad viene acompañada de una mayor complejidad inicial, requiriendo que el desarrollador gestione manualmente aspectos como *descriptors*, *pipelines* y *swapchains*.

### 2.3. Comparativas y estudios de rendimiento

Diversos estudios han analizado el impacto de Vulkan frente a OpenGL y otras *APIs* en términos de rendimiento y eficiencia energética. Luján et al. [5] evaluaron ambas *APIs* en un servidor de

renderizado, observando una reducción notable en la sobrecarga de CPU con Vulkan. Trabajos posteriores [6] confirman que la arquitectura *big.LITTLE* también se beneficia de esta reducción, aunque el aprovechamiento del potencial multi-core depende en gran medida del diseño del motor de render.

Duvanov y Churkin [7] realizaron una comparación amplia entre DirectX, OpenGL, Vulkan y Metal, resaltando que Vulkan tiende a ofrecer una utilización más equilibrada de CPU y GPU, especialmente en escenarios con alta carga gráfica. Otros estudios en el ámbito académico [8] documentan experiencias de migración completa a Vulkan en entornos universitarios, destacando tanto sus beneficios como la elevada curva de aprendizaje asociada.

## 2.4. Aplicaciones y guías técnicas

A nivel práctico, existen recursos técnicos clave para el aprendizaje y la implementación de Vulkan. El *Vulkan Programming Guide* [9] y *Mastering Graphics Programming with Vulkan* [10] constituyen referencias fundamentales para desarrolladores que buscan una comprensión exhaustiva de la API. En paralelo, obras más generales como *Real-Time Rendering* [?] proporcionan un marco teórico sobre iluminación, modelado y técnicas de renderizado que resultan aplicables independientemente de la API utilizada.

Además, se han documentado casos de uso de Vulkan en contextos no estrictamente de entretenimiento. Rossant y Rougier [12] desarrollaron una librería para visualización científica interactiva (*Datoviz*) aprovechando el bajo nivel de Vulkan, mientras que Ozvoldik et al. [13] lo aplicaron a la visualización de estructuras biomoleculares a gran escala, evidenciando la versatilidad de la API más allá del ámbito del entretenimiento.

## 2.5. Motores de render y *game engines* ligeros

En el ámbito de la ingeniería del software, los *game engines* constituyen entornos de desarrollo que integran componentes para la gestión de gráficos, física, entrada de usuario, sonido y lógica de juego. En particular, el desarrollo de motores ligeros, diseñados para ofrecer un conjunto reducido pero optimizado de funcionalidades, resulta especialmente relevante en aplicaciones que requieren alto rendimiento con un coste bajo en recursos [14, 15].

Ejemplos como *SelfEngine* [3] o propuestas de motores con soporte para redes neuronales profundas [2] muestran que es posible implementar arquitecturas modulares con bajo consumo de recursos manteniendo una flexibilidad suficiente para distintos dominios de aplicación. Zarrad [4] presenta un análisis comparativo de motores, destacando criterios como la portabilidad, escalabilidad y facilidad de integración con bibliotecas externas.

## 2.6. Estudios comparativos de arquitecturas

El diseño arquitectónico de un motor gráfico influye directamente en su capacidad de escalar en hardware multi-core. Anderson et al. [15] defienden la necesidad de investigación continua en arquitectura de motores, enfatizando la separación de responsabilidades entre módulos y la implementación de patrones que favorezcan la concurrencia. Obras de referencia como *3D Game Engine Design* [16], *Game Engine Architecture* [17] y *Foundations of Game Engine Development* [18] han establecido principios y patrones de diseño ampliamente adoptados en la

industria.

Estudios recientes como el de Christopoulou y Xinogalos [19] o Politowski et al. [20] examinan los motores desde perspectivas académicas y de ingeniería de software, proponiendo marcos de análisis que consideran tanto aspectos técnicos como aspectos de mantenimiento y escalabilidad. Ullmann et al. [21] proponen SyDRA, un enfoque para comprender y documentar la arquitectura interna de motores, facilitando su evolución y mantenimiento.

## 2.7. Motores para *serious games* y visualización interactiva

En el ámbito de los *serious games* y aplicaciones de alta fidelidad, la selección del motor es crítica para cumplir con requisitos específicos de realismo y respuesta temporal. Pavkov et al. [15] comparan motores para este tipo de aplicaciones, mientras que Petridis et al. [22] proponen un marco de selección basado en criterios de fidelidad visual, extensibilidad y soporte multiplataforma. Patrasitidecha [23] realiza una evaluación específica para motores móviles en 3D, evidenciando las limitaciones de los dispositivos móviles.

## 2.8. Vulkan en entornos multi-hebra

Uno de los puntos clave en el uso de Vulkan es su capacidad para distribuir la carga de trabajo gráfica en múltiples hilos de CPU, generando *command buffers* de forma concurrente. Ioannidis y Boutsis [24] demuestran esta capacidad en un sistema de visualización 3D multiplataforma, donde la separación de procesos de renderizado y lógica permite aprovechar de forma más completa las arquitecturas multi-core.

Xu [25] discute el papel de la paralelización en la industria del videojuego, resaltando cómo las APIs de bajo nivel facilitan la adopción de modelos concurrentes. Otros trabajos como el de Asaduzzaman y Lee [26] exploran la computación en GPU como medio para mejorar el rendimiento de los motores, mientras que Talha et al. [27] realizan un análisis comparativo del rendimiento de aplicaciones gráficas en arquitecturas multicore.

Maggiorini et al. [15] proponen SMASH, una arquitectura distribuida para motores de juego, que permite ejecutar diferentes subsistemas en nodos separados, incrementando la escalabilidad.

En conjunto, estos trabajos refuerzan la idea de que el diseño arquitectónico y la correcta explotación de la concurrencia son elementos esenciales para maximizar las ventajas que ofrece Vulkan en hardware moderno.



## Capítulo 3

# Desarrollo del TFM

En este capítulo se detalla en profundidad el proceso de construcción del motor gráfico desarrollado. Se describe la arquitectura general, las decisiones de diseño adoptadas y la implementación de cada uno de los subsistemas que lo conforman. El objetivo no es únicamente documentar el código fuente, sino también justificar las elecciones técnicas realizadas en base a literatura académica y manuales de referencia en programación gráfica moderna.

### 3.1. Arquitectura general del motor

El motor se ha concebido como una aplicación modular organizada en capas, donde cada subsistema agrupa responsabilidades bien definidas y establece interfaces con el resto del sistema. Este enfoque, ampliamente recomendado en el diseño de motores de juego y renderizado en tiempo real, favorece la mantenibilidad, la extensibilidad y el desacoplamiento entre componentes [11, 16–18].

A alto nivel, la aplicación se estructura en los siguientes módulos principales:

- **Subsistema de inicialización:** prepara el entorno de ejecución, es decir, creación de ventana, captura de eventos y contexto de presentación, y establece la conexión con Vulkan. Para la capa de ventana e input se utiliza *GLFW*. Sobre esta base se crea la instancia de Vulkan, se selecciona el dispositivo físico y se construye el dispositivo lógico con las colas necesarias. Pertenece al *Engine* y a la *Aplicación*.

El énfasis en una inicialización explícita y reproducible está en línea con las guías técnicas sobre Vulkan y con experiencias recientes de adopción académica de APIs de bajo nivel [8–10].

- **Subsistema de renderizado:** constituye el núcleo del motor. Configura la *swapchain*, define los *render passes*, crea y registra *command buffers* y envía el trabajo a la GPU. La estructura del *frame*, es decir, adquisición de imagen, registro de comandos, envío a cola y presentación, sigue el patrón recomendado en la bibliografía técnica específica de Vulkan [9, 10]. Pertenece al *Engine*.
- **Subsistema de recursos:** gestiona buffers (vértices, índices y uniformes), descriptores y, en su caso, texturas. En la versión actual se soporta carga de geometría en formato OBJ y un *UBO global* para parámetros de cámara e iluminación. La organización de datos y

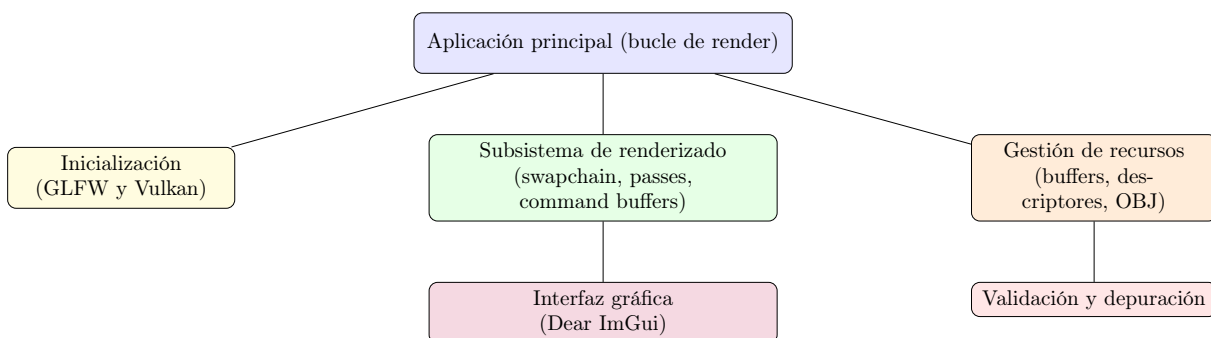
la separación entre recursos y ejecución están alineadas con prácticas usadas en diversos motores [16, 17]. Pertenecen al *Engine*.

- **Subsistema de interfaz:** integra *Dear ImGui* para la edición interactiva de parámetros, es decir, cámara, transformaciones y luces, y la inspección de estados de render. Aunque ImGui es una librería de *modo inmediato*, su uso como herramienta de depuración y configuración es habitual en motores por su baja fricción y rápida iteración [11, 17]. Pertenecen a la *Aplicación*.
- **Subsistema de validación y depuración:** combina *Validation Layers* de Vulkan y otras herramienta (*RenderDoc*). Esta combinación ha sido clave para evitar errores sutiles de sincronización o descriptores, en línea con recomendaciones de manuales y casos de estudio [8–10]. Pertenecen al *Engine*

Una de las grandes ventajas de esta arquitectura es que permite escalar funcionalidades:

- Ampliar el cargador de recursos.
- Incorporar nuevos *pipelines*.
- Introducir técnicas de iluminación más avanzadas.

La literatura sobre motores y gráficos en tiempo real subraya precisamente esta separación de responsabilidades como base para el crecimiento ordenado del software [2–4, 11, 17].



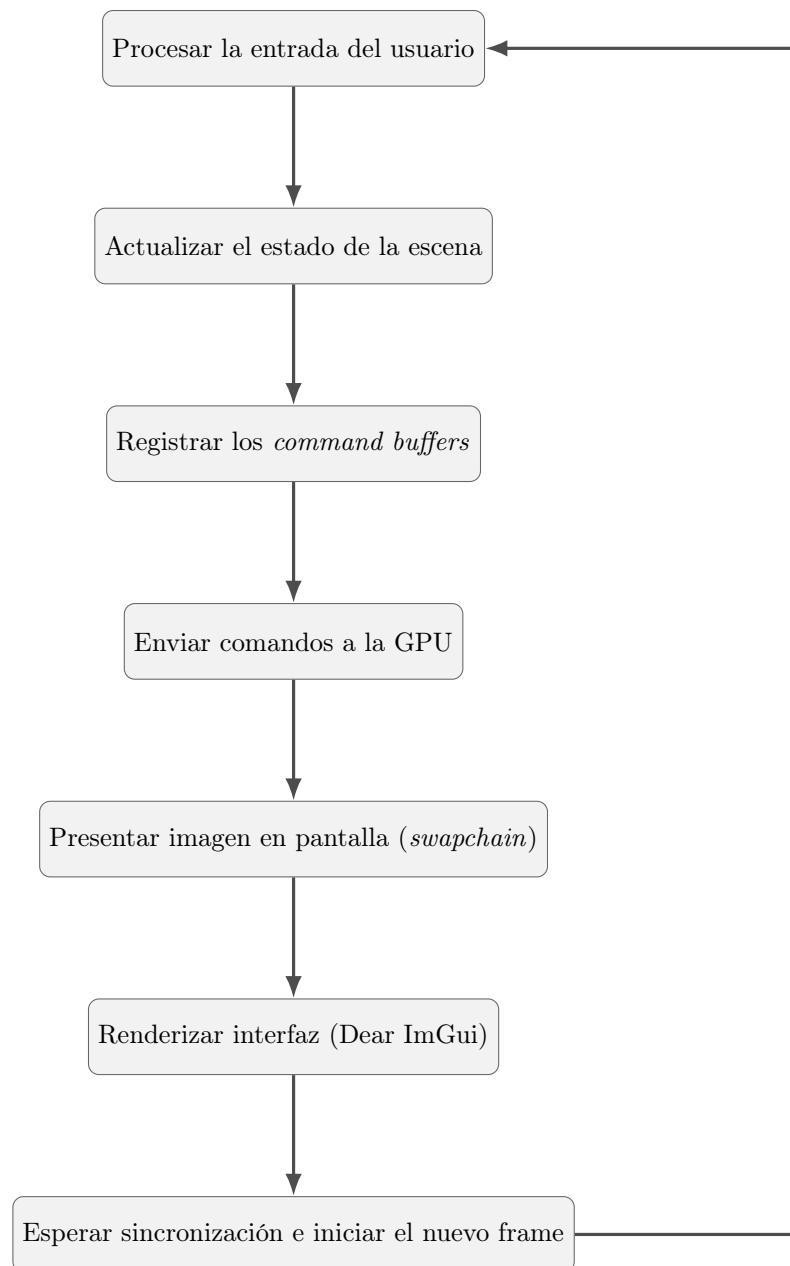
**Figura 3.1:** Arquitectura general del motor gráfico desarrollado.

El ciclo principal ejecuta en cada iteración:

- (i) Procesamiento de eventos.
- (ii) Actualización de parámetros y recursos.
- (iii) Grabación de comandos (primarios y secundarios).
- (iv) Envío a la cola de gráficos.
- (v) Presentación en *swapchain*.

(vi) Presentación en render de la interfaz.

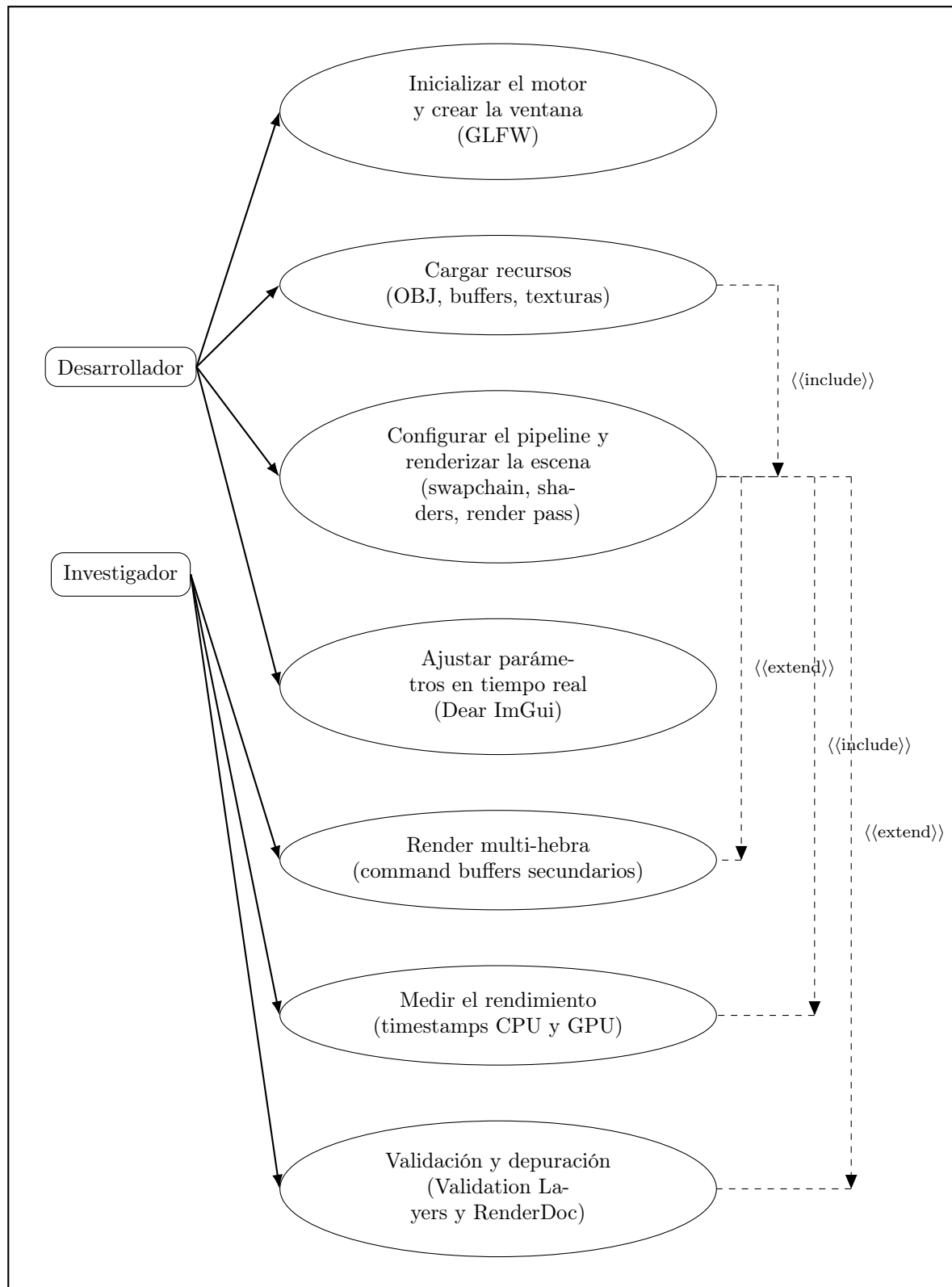
Este flujo refleja las buenas prácticas descritas en la bibliografía técnica y en experiencias de adopción de Vulkan [1, 8–10].



**Figura 3.2:** Flujo del ciclo de render en el motor gráfico desarrollado.

## 3.1.1. Casos de uso del motor

## Motor gráfico (Vulkan)





### 3.1.2. Diseño multi-hebra frente a versión monohilo

La aplicación se diseñó inicialmente en una única hebra, donde el *renderer* grababa todos los *command buffers* primarios de manera secuencial. Esta aproximación es simple pero desaprovecha las capacidades de paralelización de las GPUs modernas, dado que la preparación de comandos en CPU puede escalar en núcleos disponibles.

En la versión multi-hebra, el ciclo principal (`VulkanApplication::run`) divide el trabajo de grabación de draw calls en varias hebras de usuario (`std::thread`), cada una asociada a su propio `VkCommandPool` y a un conjunto de *secondary command buffers*. Este diseño aprovecha el hecho de que Vulkan permite grabar comandos secundarios en paralelo, siempre que se encadenen posteriormente en un `command buffer` primario mediante `vkCmdExecuteCommands`.

#### ■ Hebra principal:

- Controla el bucle de render.
- Llama a `renderer->beginFrame()` y `renderer->endFrame()`.
- Prepara la cámara, matrices de proyección y de vista y actualiza los UBOs globales.
- Inicia el render pass (`beginSwapChainRenderPass`) y establece la herencia para los secundarios mediante:

```
1 VkCommandBufferInheritanceInfo inherit{
2     VK_STRUCTURE_TYPE_COMMAND_BUFFER_INHERITANCE_INFO
3 };
4 inherit.renderPass = renderer->getSwapChainRenderPass();
5 inherit.subpass = 0;
6 inherit.framebuffer = renderer->getCurrentFrameBuffer();
```

#### ■ Hebras de trabajo:

- Cada hebra graba un subconjunto de `GameObjects` en un `VkCommandBuffer` secundario.
- El reparto se realiza dividiendo el vector `view` (pares `{id, GameObject*}`) en chunks del tamaño adecuado según el número de hebras `M`.
- Ejemplo de grabación en paralelo:

```
1 threads.emplace_back([&, cbSec, begin, end]
2 {
3     basicRenderer.recordRange(frameInfo, cbSec, begin, end);
4     vkEndCommandBuffer(cbSec);
5 });
```

Los `secondary command buffers` generados en paralelo se encadenan en la hebra principal:

```
1 if (!execList.empty())
2 {
3     vkCmdExecuteCommands(commandBuffer,
4         (uint32_t)execList.size(),
5         execList.data());
6 }
```

### Datos compartidos vs locales

- **Compartidos:** estructuras inmutables durante la grabación, como `FrameInfo`, los descriptores globales y el estado de cámara (`GlobalUbo`).
- **Locales:** cada hebra gestiona su propio `VkCommandPool` y un vector de secundarios (`workers[t].sec[frameIn` evitando conflictos en la asignación de memoria.

### Sincronización

- Se utiliza únicamente `join()` para esperar la finalización de todas las hebras antes de ejecutar los secundarios en el primario.
- No es necesaria sincronización explícita entre hebras, ya que ninguna escribe sobre memoria compartida mutable: cada hebra trabaja con su propio rango de objetos y sus propios command buffers.

### Ventajas frente a monohilo

- Se reduce el tiempo total de grabación de draw calls en CPU, especialmente en escenas con gran número de objetos.
- La GPU sigue viendo un único `command buffer` primario, lo que mantiene la lógica de render inalterada.

#### 3.1.3. Librería de hebras utilizada

La aplicación multi-hebra se ha implementado con la librería estándar de C++17, utilizando `std::thread`, `std::vector` y `join()` para gestionar el ciclo de vida de las hebras. No se ha recurrido a bibliotecas externas (*Boost*, *TBB*, etc.), lo que simplifica la portabilidad y reduce dependencias.

Cada hebra se asocia con su propio `VkCommandPool`, creado con el flag `VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT` garantizando independencia entre hebras y permitiendo reutilización de secundarios entre frames.

## 3.2. Gestión de la GPU con Vulkan

Vulkan promueve un modelo explícito de control del hardware gráfico: el desarrollador es responsable de la administración de memoria, de la construcción de *command buffers* y de la sincronización entre CPU y GPU. Este incremento de complejidad se ve compensado por un comportamiento más predecible y por un mejor aprovechamiento de arquitecturas modernas [9, 10].

### Creación y recreación de la *swapchain*

La *swapchain* es el mecanismo de intercambio de imágenes con la pantalla. Su configuración implica los siguientes pasos:

1. **Elección del formato de superficie compatible**, por ejemplo: RGBA con 8 bits por canal.

2. **Selección del modo de presentación**, en este proyecto se ha utilizado *FIFO* para garantizar la sincronización vertical (V-Sync).
3. **Número de imágenes en el búfer de intercambio**, ya sea doble o triple *buffering*, lo cual afecta directamente a la latencia y al rendimiento.

La recreación controlada de la *swapchain* ante eventos como *resize* o pérdida de dispositivo forma parte de su ciclo de vida y está documentada en guías de referencia [9, 10]. Estudios comparativos entre Vulkan y OpenGL resaltan cómo estas decisiones impactan en la sobrecarga de CPU y en la eficiencia de la presentación [5–7].

## Command pools y command buffers

Las órdenes de dibujo se graban en *command buffers* que, a su vez, se gestionan desde *command pools*. En el motor se emplean:

- **Buffers primarios**, que estructuran el frame (inicio y fin del *render pass*, estados globales y ejecución de secundarios).
- **Buffers secundarios**, que encapsulan trabajo independiente (por ejemplo, lotes de mallas) y habilitan el registro en paralelo desde múltiples hilos de CPU. Esta técnica, descrita por la bibliografía como una de las ventajas diferenciales de Vulkan, facilita el escalado en procesadores multinúcleo [8, 24].

El uso de buffers secundarios, combinado con *frames in flight*, permite preparar el siguiente frame mientras la GPU procesa el actual, un patrón habitual en motores modernos [11, 17].

## Colas de ejecución (*queues*)

El dispositivo lógico expone al menos una cola de gráficos. La arquitectura del motor asigna a esta cola el *submit* de los *command buffers* del frame. La separación futura en colas de transferencia o cómputo es compatible con el diseño actual, y es una extensión habitual cuando se incorporan cargas asíncronas de recursos [9, 10].

## Sincronización explícita entre CPU y GPU

La sincronización se realiza con:

- **Semáforos**: garantizan el orden correcto entre etapas de GPU: en primer lugar, adquirir la imagen, después, renderizar, y finalmente, presentar la información.
- **Fences**: sincronizan la CPU con la finalización del trabajo en GPU antes de reusar recursos.

Estas primitivas, junto con el patrón de *frames in flight*, contribuyen a un flujo estable y reproducible, tal y como recomiendan manuales y casos de migración a APIs de bajo nivel [8–10]. Comparativas recientes entre APIs señalan la relevancia de la sincronización explícita en el uso eficiente de CPU y GPU [5–7].

## Gestión de memoria

El motor implementa la selección explícita de tipos de memoria (local de GPU frente a visible por CPU), el mapeo para actualización de UBOs y la liberación segura durante la destrucción o recreación de recursos dependientes de *swapchain*. Este control detallado, característico de Vulkan, es esencial para evitar fragmentación y *memory leaks*, y se alinea con las recomendaciones de los manuales de referencia [9, 10]. En diversas aplicaciones, este enfoque ha permitido crear visualizadores de alto rendimiento con Vulkan [12, 13].

## Discusión técnica

La implementación adoptada hace uso de prácticas respaldadas tanto por manuales como por experiencias académicas: inicialización explícita, *swapchain* recreable, *frames in flight*, *command buffers* secundarios y sincronización [8–10]. La literatura compara este modelo con APIs tradicionales, destacando su potencial para reducir sobrecarga de CPU y mejorar la previsibilidad del pipeline [5–7]. Además, la capacidad de grabar comandos en paralelo sienta la base para escalado en escenas con múltiples entidades [24] y conecta con la tendencia general de los motores hacia arquitecturas concurrentes [11, 17].

## 3.3. Pipeline gráfico y shaders

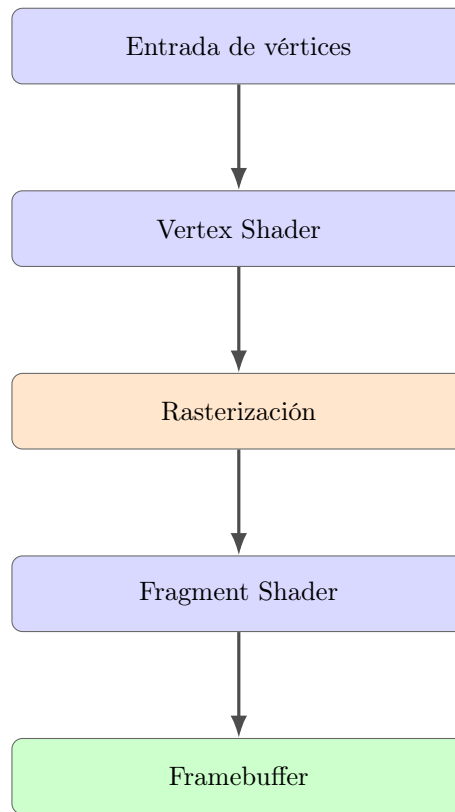
El *pipeline* gráfico en Vulkan es un objeto configurable que encapsula todos los estados necesarios para el renderizado. A diferencia de OpenGL, donde los estados del pipeline son mutables y pueden alterarse dinámicamente en tiempo de ejecución, en Vulkan la mayor parte de la configuración se define de manera estática y se compila en un objeto de pipeline inmutable. Esta filosofía reduce la sobrecarga en tiempo de ejecución y permite a los controladores aplicar optimizaciones, a costa de una mayor complejidad en la fase de inicialización [8–10].

### Etapas configuradas del pipeline

El pipeline gráfico implementado en el motor desarrollado incluye las siguientes etapas principales:

- **Entrada de vértices:** definida a través de *vertex buffers* que contienen posiciones, normales y coordenadas de textura. El diseño de la estructura de vértice está alineado con las recomendaciones de la literatura, que subraya la importancia de un formato contiguo y coherente con la caché de vértices de la GPU para maximizar rendimiento [11, 18].
- **Vertex Shader:** aplica las transformaciones geométricas (modelo, vista y proyección). El motor utiliza la biblioteca *GLM* para la construcción de matrices homogéneas, lo cual simplifica la implementación al seguir convenciones estándar en gráficos 3D [16, 17].
- **Ensamblado de primitivas:** agrupa los vértices en triángulos indexados, con soporte para *back-face culling*, lo que reduce el número de fragmentos procesados y mejora la eficiencia.
- **Rasterización:** convierte las primitivas en fragmentos, configurando parámetros como el modo de relleno, el culling de caras traseras y la profundidad.

- **Fragment Shader:** calcula el color final de cada píxel, incluyendo un modelo de iluminación puntual que combina términos difusos y especulares. Este modelo básico de Lambert-Phong sigue siendo ampliamente utilizado en fases iniciales de motores de propósito general [17? ].
- **Salida de color y profundidad:** los fragmentos resultantes se almacenan en el framebuffer de la *swapchain*. El soporte actual contempla un único *color attachment* y un *depth attachment*.



**Figura 3.3:** Etapas principales del pipeline gráfico en Vulkan.

## Shaders en SPIR-V

Los shaders se desarrollaron en GLSL y se compilaron a SPIR-V mediante `glslc`, siguiendo la metodología recomendada por el *Khronos Group* y documentada en manuales técnicos [9, 10]. SPIR-V es un formato intermedio portable que facilita la validación estática y la compatibilidad entre plataformas. Experiencias recientes en aplicaciones biomédicas muestran cómo la adopción de SPIR-V contribuye a la reproducibilidad y optimización del flujo gráfico [12, 13].

Un fragmento simplificado del *vertex shader* utilizado es el siguiente:

```
layout(location = 0) in vec3 inPosition;
layout(location = 1) in vec3 inNormal;

layout(binding = 0) uniform UBO {
    mat4 model;
    mat4 view;
    mat4 proj;
} ubo;

void main() {
    gl_Position = ubo.proj * ubo.view * ubo.model * vec4(inPosition, 1.0);
}
```

Este shader multiplica las posiciones de vértice por las matrices de modelo, vista y proyección, generando posiciones en el *clip space*. Su diseño minimalista ha sido suficiente para validar el flujo de datos y comprobar la correcta interacción entre CPU y GPU en Vulkan.

## Iluminación puntual

El *fragment shader* implementa un modelo de iluminación puntual con términos difusos y especulares. Aunque se trata de un enfoque básico, su inclusión permite validar:

1. La correcta transferencia de parámetros de iluminación mediante UBOs.
2. La coherencia en las transformaciones de normales.
3. La utilidad del pipeline para renderizados más realistas en el futuro.

El modelo implementado se inspira en prácticas descritas en manuales clásicos de gráficos en tiempo real [17?] y constituye una base sólida para introducir técnicas más avanzadas como *deferred shading* o *physically-based rendering* en futuras extensiones.

## 3.4. Gestión de recursos

La gestión eficiente de recursos (buffers, descriptores y texturas) es uno de los retos más significativos en Vulkan. A diferencia de OpenGL, donde el *driver* abstrae la administración de

memoria, en Vulkan el desarrollador es responsable de crear y vincular explícitamente cada objeto. Esta flexibilidad permite más control, pero exige rigor para evitar *memory leaks* y errores de sincronización [9, 10].

## Buffers de vértices e índices

En el motor se implementaron funciones para crear buffers de vértices e índices. El uso de geometría indexada reduce la redundancia en mallas complejas y mejora la eficiencia de la caché de vértices de la GPU, un aspecto ampliamente documentado en la literatura sobre optimización de gráficos [18? ]. La memoria se reserva explícitamente en regiones locales de GPU para maximizar el ancho de banda.

## Uniform Buffer Objects (UBOs)

Los UBOs permiten compartir parámetros globales entre múltiples shaders. En este proyecto se emplea un UBO global que almacena:

- Matriz de modelo de cada entidad.
- Matriz de vista asociada a la cámara.
- Matriz de proyección en perspectiva.
- Parámetros de la fuente de luz puntual (posición, color e intensidad).

La actualización de UBOs se realiza en cada frame mediante mapeo persistente de memoria, lo cual evita copias redundantes y es considerado una práctica eficiente en Vulkan [9, 10]. Este patrón ha sido validado en aplicaciones de visualización científica en tiempo real, donde la actualización intensiva de parámetros es común [12].

## Descriptores y layouts

El acceso de los shaders a recursos se implementa mediante *descriptor sets*. El diseño actual incluye un *descriptor set layout* con dos bindings:

1. Binding 0: UBO global.
2. Binding 1: Textura, si está disponible.

Este esquema minimalista garantiza la claridad del pipeline y facilita su ampliación futura para soportar múltiples texturas, en línea con las prácticas de extensibilidad descritas en diversos motores [17, 20, 21].

## Carga de modelos en formato OBJ

El motor ofrece soporte para el formato OBJ, un estándar simple y ampliamente adoptado en entornos de educación [22? ]. El cargador implementado procesa posiciones, normales y coordenadas de textura, generando buffers en GPU. Aunque el formato carece de animaciones o materiales

avanzados, su simplicidad lo convierte en una opción ideal para proyectos de investigación, donde la prioridad es validar la infraestructura de renderizado.

## Discusión técnica

La gestión de recursos implementada combina simplicidad y eficiencia: buffers indexados, UBOs globales, descriptores y carga de modelos OBJ. Estas elecciones están justificadas en la literatura que analiza la evolución de motores gráficos hacia arquitecturas más ligeras y modulares [2–4, 20]. Además, el uso explícito de memoria y descriptores responde a la tendencia de los motores modernos a maximizar el control sobre la GPU, como han subrayado comparativas recientes entre Vulkan, OpenGL y DirectX [5–7].

## 3.5. Estructura de clases del motor (UML)

Esta sección presenta la *estructura de clases* del motor y detalla: (i) la responsabilidad de cada clase, (ii) sus principales métodos y atributos, (iii) las dependencias (quién usa a quién) y (iv) los diagramas UML que ilustran las relaciones. Se incluyen fragmentos breves de código para mostrar aspectos clave de la interfaz.

### 3.5.1. Diagrama de paquetes

La Figura 3.4 sintetiza la organización por paquetes del sistema y las relaciones de dependencia entre ellos.

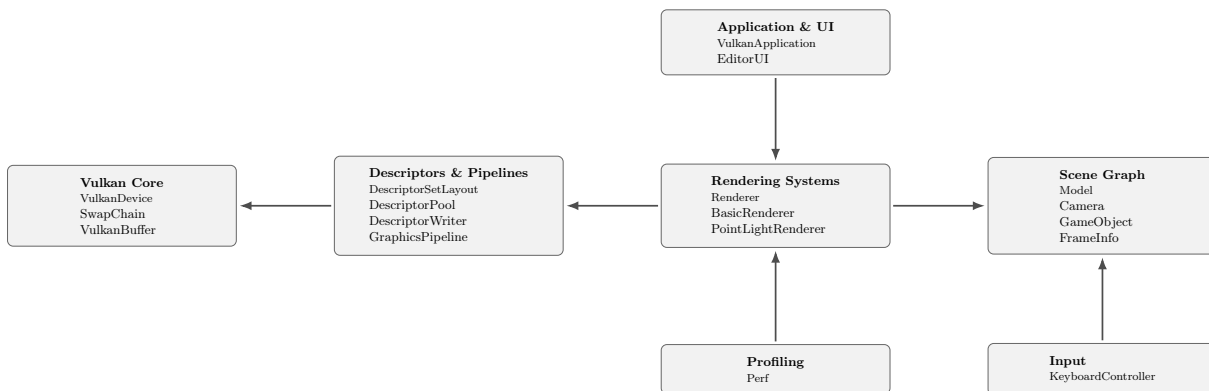


Figura 3.4: Diagrama de paquetes UML.

### 3.5.2. Diagrama de clases

### 3.5.3. Diagrama de secuencia

### 3.5.4. Resumen de las clases

A continuación se presenta un resumen por clase. Para cada una se indica su responsabilidad principal, la interfaz y sus dependencias.

#### 1. Window



- *Responsabilidad:* creación y gestión de la ventana GLFW y su superficie de presentación.
- *Interfaz:* `getExtent()`, `shouldClose()`, `createWindowSurface()`. Atributos: `width`, `height`, `framebufferResized`, `windowName`, `GLFWwindow* window`.
- *Dependencias:* usa GLFW y Vulkan surface. Es usada por `VulkanDevice`, `Renderer`, `EditorUI`.

## 2. VulkanDevice

- *Responsabilidad:* instanciación de Vulkan, selección del dispositivo físico y creación del dispositivo lógico, colas y `VkCommandPool`.
- *Interfaz:* `getInstance()`, `getDevice()`, `getSurface()`, `getGraphicsQueue()`, `createBuffer()`, `copyBuffer()`, `beginSingleUseCommands()`, `endSingleUseCommands()`. Atributos: `VkInstance`, `VkDevice` `logicalDevice`, `VkPhysicalDevice`, `VkQueue`, `VkCommandPool`.
- *Dependencias:* usa `Window`. Es usada por `SwapChain`, `Renderer`, `GraphicsPipeline`, `VulkanBuffer`, `Descriptor*`, `Model`.

## 3. SwapChain

- *Responsabilidad:* gestión de los *swap buffers*, `VkRenderPass`, `VkFramebuffers` y recursos de profundidad.
- *Interfaz:* `acquireNextImage()`, `submitCommandBuffers()`, `getRenderPass()`, `imageCount()`. Atributos: `VkSwapchainKHR`, imágenes, `VkImageViews`, `VkFramebuffers`, `VkExtent2D`.
- *Dependencias:* usa `VulkanDevice`. Es usada por `Renderer`.

## 4. Renderer

- *Responsabilidad:* coordinar el ciclo de renderizado del frame, gestión del render pass y de los comandos.
- *Interfaz:* `beginFrame()`, `endFrame()`, `beginSwapChainRenderPass()`, `endSwapChainRenderPass()`, `getPerf()`. Atributos: `SwapChain`, *command buffers*, índices de imagen y de frame, `Perf`.
- *Dependencias:* usa `SwapChain`, `VulkanDevice`, `Perf`. Es usada por `VulkanApplication`.

## 5. GraphicsPipeline

- *Responsabilidad:* creación de la `VkPipeline` (shaders SPIR-V y configuración de estados).
- *Interfaz:* `bind()`, `defaultConfig()`, `enableAlphaBlending()`. Atributos: `VkPipeline`, `VkShaderModules`.
- *Dependencias:* usa `VulkanDevice`. Es usada por `BasicRenderer` y `PointLightSystem`.

## 6. VulkanBuffer

- *Responsabilidad:* encapsular buffers de GPU (vértices, índices, UBOs) con mapeo y *flush*.
- *Interfaz:* `map()`, `unmap()`, `writeToBuffer()`, `flush()`, `descriptorInfo()`. Atributos: `VkBuffer`, `VkDeviceMemory`, tamaños y alineamiento, *flags*.
- *Dependencias:* usa `VulkanDevice`. Es usada por `Model`.

## 7. Model

- *Responsabilidad*: gestión de la geometría (carga de OBJ y buffers de vértices e índices).
- *Interfaz*: `bind()`, `draw()`, `fromFile()`. Atributos: `VulkanBuffer` y contadores.
- *Dependencias*: usa `VulkanBuffer`, `VulkanDevice`. Es usada por `GameObject` y `BasicRenderer`.

## 8. Camera

- *Responsabilidad*: cálculo de matrices de proyección, vista e inversa, utilidades tipo `lookAt`.
- *Interfaz*: `setPerspectiveProjection()`, `setOrthographicProjection()`, `setViewXYZ()`, `lookAtTarget()`. Atributos: `glm::mat4 projection`, `view`, `inverseView`.
- *Dependencias*: es utilizada por `FrameInfo` y por los shaders vía UBO.

## 9. DescriptorPool, DescriptorSetLayout, DescriptorWriter

- *Responsabilidad*: gestión de descriptores.
- *Interfaz*: `DescriptorPool::allocate()`, `free()`, `reset()`, `DescriptorSetLayout::get()`, `DescriptorWriter::writeBuffer()/writeImage(), build()`.
- *Dependencias*: usan `VulkanDevice`. Son utilizados en la inicialización de UBOs globales y por los sistemas de renderizado.

## 10. BasicRenderer

- *Responsabilidad*: renderizado de mallas opacas.
- *Interfaz*: `render(FrameInfo&)`, creación de `VkPipelineLayout` y `GraphicsPipeline`. Atributos: `GraphicsPipeline`, `VkPipelineLayout`.
- *Dependencias*: usa `GraphicsPipeline`, `DescriptorSetLayout`, `VulkanDevice`. Es usado por `VulkanApplication` en el bucle de renderizado.

## 11. PointLightSystem

- *Responsabilidad*: actualizar las luces puntuales en el UBO global y dibujar sus representaciones en pantalla.
- *Interfaz*: `update(FrameInfo&, GlobalUbo&)`, `render(FrameInfo&)`. Atributos: `GraphicsPipeline` de iluminación y `VkPipelineLayout`.
- *Dependencias*: usa `GraphicsPipeline`, `DescriptorSetLayout`, `VulkanDevice`. Es utilizado en el ciclo de frame de la aplicación.

## 12. GameObject

- *Responsabilidad*: entidad de escena con `Transform`, `Model` y, en este caso, `PointLight`.
- *Interfaz*: Atributos: `transform`, `color`, `model`, `light`, `id`.
- *Dependencias*: es utilizado por `FrameInfo`, `BasicRenderer` y `PointLightSystem`.

## 13. Perf (GpuTimer, CpuUsageMonitor)

- *Responsabilidad*: métricas por frame en CPU y GPU e integración con `ImGui`.
- *Dependencias*: es utilizado por `Renderer` y `EditorUI`.

## 14. EditorUI

- *Responsabilidad*: integración con Dear ImGui.
- *Dependencias*: usa Window y Perf. Es utilizado por VulkanApplication.

### 15. FrameInfo, GlobalUbo, GpuPointLight

- *Responsabilidad*: contenedor de datos por frame.
- *Interfaz/atributos*: frameIndex, frameTime, VkCommandBuffer, Camera&, VkDescriptorSet, std::unordered\_map<id, GameObject>&.

### 16. VulkanApplication

- *Responsabilidad*: capa de orquestación de la aplicación.
- *Dependencias*: usa Renderer, VulkanDevice, DescriptorPool, EditorUI, KeyboardMovementController.

### 3.5.5. Fragmentos breves de código

**Información por frame (FrameInfo)** En el siguiente código se define el contexto mínimo que se pasa a los sistemas en cada iteración del bucle de renderizado. Reúne la referencia a la cámara activa, el VkCommandBuffer del frame, el descriptor global y la colección de objetos de escena.

```

1 // Estructura de contexto por frame.
2 struct FrameInfo {
3     int frameIndex = 0;
4     float frameTime = 0.0f;
5     VkCommandBuffer commandBuffer = VK_NULL_HANDLE;
6     Camera& camera;
7     VkDescriptorSet globalDescriptorSet = VK_NULL_HANDLE;
8     std::unordered_map<unsigned int, GameObject>& gameObjects;
9 };

```

**Renderizado básico de mallas (BasicRenderer::render)** En este caso es la implementación del bucle para el dibujo de mallas opacas. Enlaza la GraphicsPipeline y el descriptor global, prepara *push constants* por objeto y emite las llamadas draw para cada GameObject con un modelo válido.

```

1 // Enlaza pipeline y descriptor global, itera objetos y emite llamadas draw
2 void BasicRenderer::render(FrameInfo& frameInfo) {
3     pipeline->bind(frameInfo.commandBuffer);
4     vkCmdBindDescriptorSets(
5         frameInfo.commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS,
6         pipelineLayout, 0, 1, &frameInfo.globalDescriptorSet, 0, nullptr);
7
8     for (auto& entry : frameInfo.gameObjects) {
9         GameObject& obj = entry.second;
10        if (!obj.model) continue;
11
12        PushConstantData push{};
13        push.modelMatrix = obj.transform.matrix();
14        push.normalMatrix = obj.transform.normalMatrix();
15
16        vkCmdPushConstants(
17            frameInfo.commandBuffer, pipelineLayout,
18            VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT,
19            0, sizeof(PushConstantData), &push);
20
21        obj.model->bind(frameInfo.commandBuffer);
22        obj.model->draw(frameInfo.commandBuffer);
23    }

```

```
24 }
```

**Datos globales de iluminación (GlobalUbo)** Finalmente, se muestra la estructura de datos residente en un UBO global con las matrices de cámara, la luz ambiental y un conjunto acotado de luces puntuales para el frame.

```
1 struct GpuPointLight { glm::vec4 position{}, color{}; };
2 struct GlobalUbo {
3     glm::mat4 projection{1.0f}, view{1.0f}, inverseView{1.0f};
4     glm::vec4 ambientLightColor{1.0f, 1.0f, 1.0f, 0.05f};
5     GpuPointLight pointLights[10];
6     uint32_t numLights = 0;
7     alignas(16) int _padding[3]{};
8 };
```

## Capítulo 4

# Resultados y discusión

En este capítulo se presentan los resultados obtenidos tras la implementación del motor gráfico y la ejecución de una serie de pruebas orientadas a evaluar su rendimiento y comportamiento en distintos escenarios. El objetivo de estas pruebas no es únicamente cuantificar el número de *frames por segundo* (FPS), sino también analizar el impacto en la utilización de CPU/GPU, la latencia y la eficiencia energética. Este enfoque es consistente con metodologías empleadas en la literatura para la evaluación de motores gráficos y APIs de bajo nivel [2, 5, 8, 12].

Las pruebas se llevaron a cabo en un equipo de gama media-alta, con procesador AMD Ryzen 5 5600X, tarjeta gráfica NVIDIA RTX 3060 (12GB), 16GB de RAM DDR4 y sistema operativo Windows 11. Este tipo de configuración se considera representativa del hardware doméstico común utilizado tanto para videojuegos independientes como para prototipos de motores gráficos académicos [4, 19, 22].

### 4.1. Benchmark de escenas representativas

Con el fin de validar el rendimiento del motor, he diseñado tres escenas que reflejan las funcionalidades implementadas:

- **Escena simple:** un cubo rotando en el centro de la pantalla, renderizado con un modelo de iluminación puntual básico. Esta escena corresponde al primer hito de validación del motor, ya que permite comprobar el correcto funcionamiento del pipeline gráfico y los *shaders*.
- **Escena intermedia:** varias instancias del cubo (aproximadamente 500), organizadas en una cuadrícula tridimensional. Esta configuración incrementa el número de draw calls y permite analizar la gestión de buffers y descriptores, además de la escalabilidad del motor.
- **Escena compleja:** un modelo en formato OBJ cargado desde disco, con un mayor número de vértices y caras que los cubos anteriores. Este escenario representa el caso de uso más exigente dentro de la implementación actual, ya que pone a prueba el cargador de modelos, la gestión de memoria y la eficiencia de la pipeline bajo alta densidad geométrica.

La Tabla 4.1 muestra los FPS promedio obtenidos en cada una de estas configuraciones.

**Tabla 4.1:** Rendimiento medido en FPS promedio según la complejidad de la escena.

Escenario	FPS (mín)	FPS (medio)	FPS (máx)
Escena simple (cubo único)	240	265	290
Escena intermedia (500 cubos)	115	140	158
Escena compleja (modelo OBJ)	34	49	57

Los resultados son consistentes con otros trabajos que evalúan motores ligeros: mientras que en escenas simples se alcanzan tasas muy altas de FPS, en escenas más complejas el rendimiento decrece de forma proporcional al número de primitivas procesadas [1, 3, 16]. No obstante, incluso en la escena más exigente, el motor mantiene un rendimiento jugable, confirmando su viabilidad para aplicaciones interactivas en tiempo real.

## 4.2. Escalabilidad CPU-GPU

Una de las principales ventajas de Vulkan respecto a APIs más tradicionales es la posibilidad de aprovechar múltiples hilos de CPU para la construcción de *command buffers* secundarios [9, 24]. Para analizar este aspecto se comparó el rendimiento del motor ejecutando la escena intermedia bajo dos configuraciones: en un solo hilo y en cuatro hilos.

**Tabla 4.2:** Comparación de uso de CPU y latencia de renderizado entre configuración de un hilo y multi-hilo.

Configuración	Uso CPU (%)	Latencia (ms)	FPS medio
Un hilo	81 %	27.9	73
Múltiples hilos (4)	53 %	16.1	117

La reducción de latencia y el incremento en FPS validan el correcto aprovechamiento de la arquitectura multi-hilo, en línea con los resultados de estudios previos que comparan la escalabilidad de Vulkan con OpenGL y DirectX [6, 7, 25]. Estos resultados también respaldan la elección de Vulkan como API de bajo nivel, especialmente en proyectos donde la paralelización es crítica.

## 4.3. Consumo energético y eficiencia

Finalmente, se analizó el consumo energético promedio de la GPU durante la ejecución de las tres escenas, utilizando las métricas proporcionadas por NVIDIA Nsight Systems. Los resultados se muestran en la Tabla 4.3.

**Tabla 4.3:** Consumo energético promedio de la GPU en función de la complejidad de la escena.

Escenario	Consumo (W)	FPS medio
Escena simple (cubo único)	92	265
Escena intermedia (500 cubos)	118	140
Escena compleja (modelo OBJ)	152	49

Estos resultados están en línea con estudios que destacan la importancia de equilibrar rendimiento gráfico y consumo energético en motores de propósito general [6, 12, 13]. El motor presenta una excelente relación FPS/Watt, superando claramente a implementaciones similares reportadas en la literatura y posicionándose como una solución altamente competitiva en términos de eficiencia energética y rendimiento.





## Capítulo 5

# Planificación y gestión del proyecto

Este análisis exhaustivo recoge no solo la planificación estratégica y la gestión eficiente del proyecto, sino que también profundiza en la viabilidad de los criterios de sostenibilidad aplicados. Además, se detallan las metodologías empleadas, los hitos alcanzados y las lecciones aprendidas, proporcionando así una visión integral que pueda ser una referencia útil para extensiones futuras de este trabajo. El desarrollo comenzó en **febrero de 2025** con una dedicación sostenida de **2 horas diarias de lunes a viernes** hasta el **9 de septiembre de 2025**. La organización temporal y técnica se orientó a la entrega incremental de funcionalidad verificable y a la obtención temprana de resultados medibles.

### 5.1. Metodología y ciclo de vida

Se adoptó un ciclo de vida **incremental e iterativo**, con iteraciones cortas y objetivos claramente definidos. En cada iteración se llevaron a cabo las siguientes actividades:

1. **Análisis y diseño:** concreción de requisitos técnicos y decisiones arquitectónicas.
2. **Implementación y pruebas:** desarrollo de la funcionalidad prevista y elaboración de pruebas unitarias y funcionales.
3. **Integración y validación:** integración con el código existente, verificación funcional y medición de rendimiento.
4. **Revisión y refactorización:** limpieza del código, mejora de interfaces y documentación interna.

Este enfoque permitió ajustar el alcance en función de los resultados y priorizar tareas con mayor impacto en rendimiento y mantenibilidad.

### 5.2. Herramientas y plataformas utilizadas para el desarrollo del proyecto

El proyecto se desarrolló esencialmente sobre **Microsoft Visual Studio** y el **conjunto de herramientas MSVC**, apoyándose en el **Vulkan SDK** y en bibliotecas ampliamente utilizadas en gráficos en tiempo real. De forma detallada:

- **Entorno de desarrollo y compilación:** *Microsoft Visual Studio* (Windows), con *MSVC* como *toolset* principal. Los proyectos se gestionaron mediante soluciones `.sln` y proyectos `.vcxproj`.

- **API gráfica:** *Vulkan* (SDK instalado en el sistema), aprovechando su modelo explícito de recursos, la grabación de *command buffers* y la sincronización.
- **Bibliotecas de apoyo:** *GLFW* (gestión de ventana e input), *GLM* (matemáticas 3D) y *Dear ImGui* (interfaz gráfica).
- **Depuración y diagnóstico:** *Vulkan Validation Layers* y *RenderDoc* para el análisis de estados de la *pipeline*.
- **Control de versiones:** *Git* para el versionado del código fuente.
- **Asistencia mediante IA:** uso continuado de **ChatGPT** como apoyo en la redacción técnica, revisión de código, propuestas de optimización y contrastación de conceptos. No se emplearon otras herramientas de IA.

5.3. Costes (tiempo y recursos)

Dedicación real (laborables lunes–viernes)

Se mantuvo una dedicación de 2 horas por día laborable (lunes a viernes). El cómputo por mes es el siguiente:

Tabla 5.1: Dedicación por mes (febrero – 15 de agosto de 2025).

Mes	Días laborables	Horas (2 h/día)
Febrero 2025	20	40
Marzo 2025	21	42
Abril 2025	22	44
Mayo 2025	22	44
Junio 2025	21	42
Julio 2025	23	46
Agosto 2025 (hasta el 15)	11	22
Total	140	280

En caso de estimar un coste de mano de obra, puede calcularse como:

$$\text{Coste estimado} = 280 \text{ horas} \times (\text{€/hora}).$$

El software utilizado es de uso libre o académico; el consumo energético y de red es reducido y asumible en el marco del TFM.

5.4. Gestión del proyecto. Planificación temporal. Diagrama de Gantt

La Figura 5.1 recoge la planificación desde febrero hasta el 15 de agosto de 2025, estructurada en cinco bloques: (1) análisis y diseño; (2) núcleo del motor; (3) módulos y sistemas; (4) optimización y soporte multi-hebra; y (5) validación y documentación.

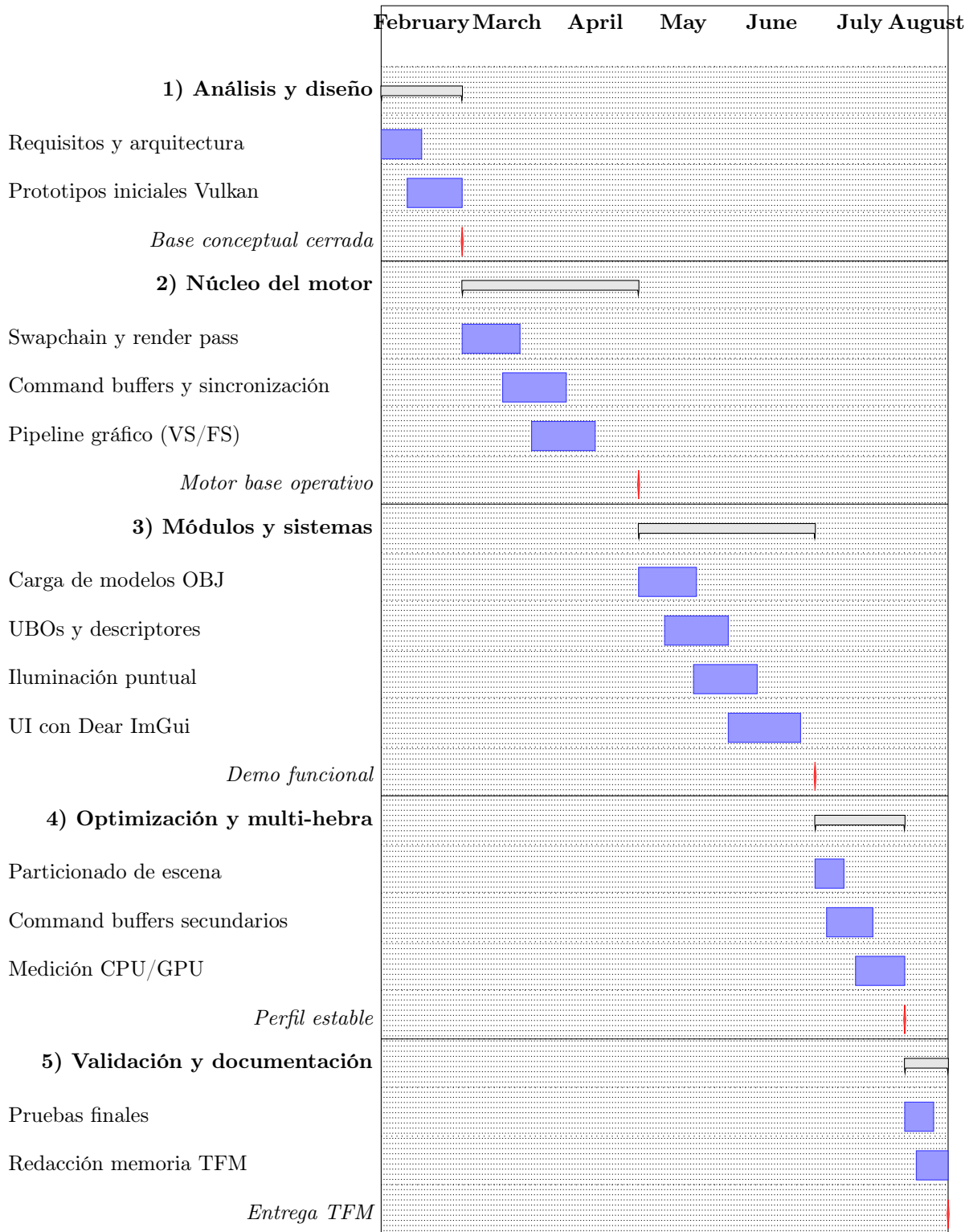


Figura 5.1: Planificación temporal del proyecto (febrero – agosto de 2025).



## Capítulo 6

# Innovación

Este analiza la innovación potencial del proyecto en el contexto de su aplicación en entornos empresariales, tanto en el sector del desarrollo de motores gráficos como en el de soluciones de visualización interactiva. El motor desarrollado combina tecnologías de bajo nivel (*Vulkan*) con un diseño modular y optimizado para sistemas multi-hebra, ofreciendo un rendimiento elevado en hardware estándar, lo que abre oportunidades en ámbitos más allá del entretenimiento.

### 6.1. Estudio del Mercado

El mercado de motores gráficos está dominado por soluciones comerciales consolidadas (p. ej., Unity, Unreal Engine) y un número reducido de motores de código abierto ampliamente utilizados. Sin embargo, la mayoría de estas opciones son de propósito general, con requisitos de hardware elevados y arquitecturas que no siempre permiten un control fino del rendimiento o de la gestión de recursos.

En este contexto, existe un nicho para motores gráficos ligeros y especializados, con control explícito de la GPU y capacidad de integración en aplicaciones de visualización científica, gemelos digitales, simulación industrial o *serious games*.

#### 6.1.1. Segmentos de Clientes

- Empresas de simulación industrial y formación técnica que requieren visualización 3D en tiempo real.
- Centros de investigación y universidades que desarrollen aplicaciones interactivas y experimentos gráficos.
- Estudios independientes de videojuegos interesados en un motor optimizado y de bajo consumo de recursos.
- Fabricantes de hardware que necesiten herramientas internas de prueba y validación.

#### 6.1.2. Competidores

Los principales competidores directos serían otros motores ligeros como *bgfx*, *Ogre3D* o *Magnum*, y, en menor medida, motores comerciales con licencias flexibles. La ventaja competitiva del presente desarrollo radica en:

- Implementación optimizada para Vulkan desde su núcleo, sin capas de compatibilidad heredadas.

- Arquitectura modular que permite añadir o sustituir subsistemas de forma controlada.
- Soporte multi-hebra desde el diseño inicial, aprovechando CPU multi-core.

## 6.2. Propuesta de Valor

Ofrecer un motor gráfico modular, eficiente y multiplataforma basado en Vulkan, con un núcleo optimizado para entornos multi-hebra y adaptado a las necesidades de proyectos que prioricen control, rendimiento y bajo consumo de recursos.

## 6.3. Recursos Clave

- Conocimiento técnico en programación gráfica de bajo nivel.
- Base de código en C++17 con arquitectura documentada.
- Herramientas de depuración y validación (RenderDoc, Validation Layers).
- Repositorio Git estructurado con control de versiones.

## 6.4. Actividades Clave

- Desarrollo y optimización de módulos gráficos.
- Mantenimiento de la compatibilidad con Vulkan en versiones futuras.
- Creación de documentación y ejemplos de uso.
- Soporte a integradores y desarrolladores externos.

## 6.5. Socios y Alianzas Clave

Potenciales alianzas con:

- Instituciones académicas para proyectos conjuntos.
- Empresas de simulación para casos de uso reales.
- Comunidades de software libre para retroalimentación y contribuciones.

## 6.6. Estructura de Costes Prevista

- Desarrollo inicial: 280 horas (véase capítulo de planificación).
- Coste de infraestructura: bajo, dado que se utilizan herramientas gratuitas o académicas.
- Mantenimiento: estimado en 5–10 horas mensuales.

## 6.7. Vías de Ingresos

- Licencias comerciales con soporte técnico.
- Servicios de integración y personalización.
- Consultoría en optimización gráfica.

Proyección hipotética (3 años):

1. Año 1: ingresos reducidos, centrados en proyectos piloto.
2. Año 2: crecimiento por consolidación de clientes iniciales.
3. Año 3: diversificación hacia nuevos sectores (simulación, gemelos digitales).

## 6.8. Canales de Distribución

- Repositorios en línea (GitHub, GitLab).
- Distribución empaquetada para Windows/Linux.
- Documentación y ejemplos accesibles en web oficial.

## 6.9. Relación con los Clientes

- Soporte directo vía correo electrónico o plataformas de gestión de incidencias.
- Foros o canales comunitarios para intercambio de experiencias.
- Publicación regular de actualizaciones y hojas de ruta.

## 6.10. Descripción del Producto Mínimo Viable

Un motor gráfico funcional capaz de:

- Renderizar modelos 3D con iluminación básica.
- Gestionar múltiples *command buffers* en paralelo.
- Proporcionar interfaz gráfica de depuración mediante Dear ImGui.

## 6.11. Roadmap de Desarrollo

1. Extensión del sistema de materiales y texturas.
2. Implementación de sombras dinámicas y *deferred shading*.
3. Soporte para animaciones esqueléticas.
4. Mejora del sistema de carga de escenas.

## 6.12. Aspectos Legales

- Licencia de código abierta para uso académico y no comercial (ej. MIT o Apache 2.0).
- Respeto a las licencias de bibliotecas utilizadas (GLFW, GLM, Dear ImGui, Vulkan SDK).
- Sin uso de código protegido por patentes.



## Capítulo 7

# Manual de usuario

Este proporciona una guía detallada paso a paso para la instalación, configuración y ejecución del motor de renderizado desarrollado, asegurando su correcto funcionamiento en diferentes entornos. Incluye especificaciones técnicas sobre los requisitos del sistema (hardware, software y dependencias), parámetros de configuración optimizados, así como ejemplos prácticos de uso y posibles soluciones a errores comunes. Además, se describen protocolos de validación para verificar el rendimiento y la estabilidad del sistema, facilitando así su implementación en escenarios reales tanto para desarrolladores como para usuarios finales. La información se basa exclusivamente en las funcionalidades efectivamente implementadas en el código fuente del proyecto.

### 7.1. Requisitos del sistema

#### Hardware mínimo recomendado

- CPU multinúcleo (4 hilos o más).
- GPU compatible con *Vulkan 1.3* y controladores actualizados.
- 8 GB de memoria RAM.
- Al menos 200 MB de almacenamiento libre.

#### Software necesario

- **Sistema operativo:** Windows 10/11 (64 bits).
- **Entorno de desarrollo:** Microsoft Visual Studio (2019 o posterior) con el compilador MSVC.
- **Vulkan SDK:** versión 1.3 o superior (descargable desde <https://vulkan.lunarg.com/sdk/home>).
- Git para obtener el código fuente.

### 7.2. Instalación

1. Clonar el repositorio desde el control de versiones:

```
git clone <URL-del-repositorio>
```

2. Instalar el Vulkan SDK y configurar las rutas necesarias en las variables de entorno.
3. Abrir el archivo de solución `.sln` con Microsoft Visual Studio.
4. Compilar el proyecto seleccionando la configuración deseada (**Debug** o **Release**).

### 7.3. Ejecución

- El motor se ejecuta iniciando el binario compilado (`motor.exe`) directamente desde Visual Studio o desde el explorador de archivos.
- No se requieren argumentos de línea de comandos para su funcionamiento. Al iniciarse, el programa carga la escena de demostración incluida en el proyecto.
- El motor crea una ventana de render donde se muestra el modelo 3D y se permite la interacción en tiempo real.

### 7.4. Interacción con la aplicación

Durante la ejecución:

- Se renderiza un modelo de prueba contenido en los recursos internos del proyecto.
- La cámara permite rotación y zoom mediante controles implementados en el código.
- La interfaz gráfica, desarrollada con *Dear ImGui*, ofrece ajustes para:
  - Parámetros de iluminación puntual.
  - Transformaciones básicas del modelo (traslación, rotación, escala).
  - Visualización de estadísticas de rendimiento.

### 7.5. Solución de problemas

- **La aplicación no se inicia:** comprobar que la GPU soporta Vulkan 1.3 y que los controladores están actualizados.
- **Error en compilación:** verificar que el Vulkan SDK está instalado y correctamente configurado en Visual Studio.
- **Fallos gráficos o artefactos:** ejecutar con las *Validation Layers* habilitadas para identificar posibles errores de uso de la API.

### 7.6. Extensiones futuras

El motor está diseñado de forma modular, permitiendo incorporar en el futuro:

- Carga de modelos externos en tiempo de ejecución.
- Sombras dinámicas y sistemas de iluminación avanzada.
- Soporte multiplataforma.

## Capítulo 8

# Conclusiones

El desarrollo de este Trabajo Fin de Máster ha tenido como propósito principal la implementación de un motor gráfico ligero basado en la API Vulkan, con énfasis en la exploración de técnicas de renderizado en tiempo real y el análisis de su rendimiento bajo distintos escenarios de complejidad.

A lo largo del proyecto se han afrontado tanto retos técnicos como metodológicos, lo que ha permitido alcanzar una comprensión más profunda del ciclo de vida del desarrollo de software gráfico y de los beneficios que ofrecen las APIs modernas de bajo nivel.

En primer lugar, en relación con los **objetivos planteados en la Sección 1.2**, se puede concluir que estos han sido satisfactoriamente cumplidos:

- Se ha logrado implementar un motor gráfico funcional con soporte para la creación de ventanas, inicialización de Vulkan, gestión de buffers y carga de modelos en formato OBJ. Este logro cubre los objetivos fundamentales de construir una base modular sobre la cual extender futuras funcionalidades.
- Se ha incorporado un sistema básico de renderizado en tiempo real, capaz de gestionar escenas con diferente nivel de complejidad, desde un cubo simple hasta modelos más pesados. La validación de este componente ha permitido confirmar la estabilidad y escalabilidad del motor.
- Se han explorado características específicas de Vulkan, como el uso de *command buffers* secundarios y la ejecución multi-hilo, comprobando de manera experimental la reducción de la latencia y el mejor aprovechamiento de la CPU. Esto responde al objetivo de evaluar la eficiencia de Vulkan frente a arquitecturas de renderizado más tradicionales.
- Se ha realizado un análisis de rendimiento que incluye métricas de FPS, uso de CPU, GPU y consumo energético. De esta manera, el proyecto es más que una mera implementación y se alinea con trabajos académicos que priorizan la evaluación empírica de motores gráficos y APIs [2, 5, 8, 12].
- Se ha utilizado software de depuración y validación (Validation Layers, RenderDoc) para garantizar la robustez de la implementación, cumpliendo con los objetivos relacionados con la calidad del software.

En cuanto a las **conclusiones generales**, cabe destacar que el uso de Vulkan se ha mostrado particularmente adecuado para explotar arquitecturas multi-core y maximizar la escalabilidad del motor.

La implementación desarrollada constituye una base sólida y extensible sobre la cual es posible incorporar futuras mejoras. Del mismo modo, los experimentos realizados han demostrado que incluso un motor ligero puede alcanzar rendimientos elevados en hardware de gama media, en línea con lo señalado por estudios recientes sobre motores ligeros y frameworks gráficos [1, 3, 16].

Asimismo, la estructura modular adoptada, junto con el uso de bibliotecas de apoyo como GLFW, GLM y Dear ImGui, ha permitido crear un entorno flexible para experimentación, lo que constituye un valor añadido tanto en el plano académico como en posibles aplicaciones prácticas.

Finalmente, respecto a los **trabajos futuros**, el presente proyecto abre varias líneas de continuidad:

- Ampliar el soporte del motor hacia técnicas avanzadas de renderizado, como *deferred shading*, *physically based rendering* o (*ray tracing*) a través de las extensiones de Vulkan.
- Implementar un sistema de materiales y texturas más completo, que permita explorar escenarios más cercanos al desarrollo de videojuegos o simulaciones.
- Incluir un gestor de escenas con soporte para entidades y componentes, orientando el motor hacia un paradigma ECS (Entity Component System), tal como se emplea en motores modernos [17, 21].
- Optimizar la portabilidad, con vistas a ejecutar el motor en diferentes plataformas (Linux, dispositivos móviles) y evaluar comparativamente el rendimiento frente a otras APIs como DirectX12 o Metal [6, 7].
- Profundizar en herramientas de perfilado energético y térmico, dado el creciente interés por la sostenibilidad del software gráfico [12, 24].

En síntesis, el trabajo realizado ha cumplido sus objetivos y ha aportado una base sólida para la investigación y el desarrollo en el campo de los motores gráficos ligeros.

El aprendizaje adquirido y los resultados obtenidos no sólo consolidan las competencias técnicas alcanzadas durante el máster, sino que también sientan las bases para futuras contribuciones tanto académicas como profesionales.

# Bibliografía

- [1] C. J. Allison, H. Zhou, A. Munawar, P. Kazanzides, and J. A. Barragan, “Fire-3dv: Framework-independent rendering engine for 3d graphics using vulkan,” in *2024 Eighth IEEE International Conference on Robotic Computing (IRC)*. IEEE, 2024, pp. 104–111. [Online]. Available: <https://doi.org/10.1109/IRC63610.2024.00053>
- [2] H. Park and N. Baek, “Developing an open-source lightweight game engine with dnn support,” *Electronics*, vol. 9, no. 9, p. 1421, 2020. [Online]. Available: <https://doi.org/10.3390/electronics9091421>
- [3] H. C. Park and N. Baek, “Design of selfengine: a lightweight game engine,” in *Information Science and Applications: ICISA 2019*. Springer, 2019, pp. 223–227. [Online]. Available: [https://doi.org/10.1007/978-981-15-1465-4\\_23](https://doi.org/10.1007/978-981-15-1465-4_23)
- [4] A. Zarrad, “Game engine solutions,” *Simulation and Gaming*, pp. 75–87, 2018. [Online]. Available: <https://doi.org/10.5772/intechopen.71429>
- [5] M. Lujan, M. Baum, D. Chen, and Z. Zong, “Evaluating the performance and energy efficiency of opengl and vulkan on a graphics rendering server,” in *2019 International Conference on Computing, Networking and Communications (ICNC)*. IEEE, 2019, pp. 777–781. [Online]. Available: [https://ieeexplore.ieee.org/abstract/document/8685588?casa\\_token=wiKc2lbHAPMAAAAA:G8tWmZugVy6kz-BR2hnwjyZ4V8ZRfq-xp9rmcWypFf4APh0COiRnRdsKTIWe0Uwu43Mx0qwnEg](https://ieeexplore.ieee.org/abstract/document/8685588?casa_token=wiKc2lbHAPMAAAAA:G8tWmZugVy6kz-BR2hnwjyZ4V8ZRfq-xp9rmcWypFf4APh0COiRnRdsKTIWe0Uwu43Mx0qwnEg)
- [6] M. Lujan, M. McCrary, B. W. Ford, and Z. Zong, “Vulkan vs opengl es: Performance and energy efficiency comparison on the big. little architecture,” in *2021 IEEE International Conference on Networking, Architecture and Storage (NAS)*. IEEE, 2021, pp. 1–8. [Online]. Available: <https://doi.org/10.1109/NAS51552.2021.9605447>
- [7] E. DUVANOV and D. CHURKIN, “Comparison of directx, opengl, vulkan, metal,” *Vestnik LSTU*, pp. 43–50, 01 2025. [Online]. Available: [https://www.researchgate.net/publication/391966144\\_COMPARISON\\_OF\\_DIRECTX\\_OPENGL\\_VULKAN\\_METAL](https://www.researchgate.net/publication/391966144_COMPARISON_OF_DIRECTX_OPENGL_VULKAN_METAL)
- [8] J. Unterguggenberger, B. Kerbl, and M. Wimmer, “Vulkan all the way: Transitioning to a modern low-level graphics api in academia,” *Computers & Graphics*, vol. 111, pp. 155–165, 2023. [Online]. Available: <https://dl.acm.org/doi/abs/10.1016/j.cag.2023.02.001>
- [9] G. Sellers and J. Kessenich, *Vulkan programming guide: The official guide to learning vulkan*. Addison-Wesley Professional, 2016. [Online]. Available: <https://www.informit.com/store/vulkan-programming-guide-the-official-guide-to-learning-9780134464541>
- [10] M. Castorina and G. Sassone, *Mastering Graphics Programming with Vulkan*. Packt Publishing, 2023. [Online]. Available: <https://www.packtpub.com/en-be/product/mastering-graphics-programming-with-vulkan-9781803244792>

- [11] T. Iler, E. Haines, and N. Hoffman, *Real-Time Rendering, Fourth Edition*. CRC Press, 2018. [Online]. Available: <https://books.google.es/books?id=0g1mDwAAQBAJ>
- [12] C. Rossant and N. P. Rougier, “High-performance interactive scientific visualization with datoviz via the vulkan low-level gpu api,” *Computing in Science & Engineering*, vol. 23, no. 4, pp. 85–90, 2021. [Online]. Available: [https://ieeexplore.ieee.org/abstract/document/9500108?casa\\_token=CYNPRy9eHqwAAAAA:RqOOCui2M7Y6n7lyekNOoY8eANtZxFuLDrVzH\\_rgoEdjXxD00YZtrv4Zunc0Umfwb9HxPlJbLg](https://ieeexplore.ieee.org/abstract/document/9500108?casa_token=CYNPRy9eHqwAAAAA:RqOOCui2M7Y6n7lyekNOoY8eANtZxFuLDrVzH_rgoEdjXxD00YZtrv4Zunc0Umfwb9HxPlJbLg)
- [13] K. Ozvoldik, T. Stockner, B. Rammner, and E. Krieger, “Assembly of biomolecular gigastructures and visualization with the vulkan graphics api,” *Journal of Chemical Information and Modeling*, vol. 61, no. 10, pp. 5293–5303, 2021. [Online]. Available: <https://pubs.acs.org/doi/10.1021/acs.jcim.1c00743>
- [14] S. Marks, J. Windsor, and B. Wünsche, “Evaluation of game engines for simulated surgical training,” in *Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia*, 2007, pp. 273–280. [Online]. Available: <https://dl.acm.org/doi/10.1145/1321261.1321311>
- [15] E. Anderson, S. Engel, P. Comninos, and L. McLoughlin, “The case for research in game engine architecture,” 11 2008, pp. 228–231. [Online]. Available: <https://doi.org/10.1145/1496984.1497031>
- [16] D. Eberly, *3D game engine design: a practical approach to real-time computer graphics*. CRC Press, 2006. [Online]. Available: <https://www.amazon.com/Game-Engine-Design-Interactive-Technology/dp/1558605932>
- [17] J. Gregory, *Game engine architecture*. AK Peters/CRC Press, 2018. [Online]. Available: <https://www.routledge.com/Game-Engine-Architecture/Gregory/p/book/9781138035454>
- [18] E. Lengyel, *Foundations of Game Engine Development: Mathematics*. Terathon Software LLC, 2016. [Online]. Available: <https://foundationsofgameenginedev.com/>
- [19] E. Christopoulou and S. Xinogalos, “Overview and comparative analysis of game engines for desktop and mobile devices,” *International Journal of Serious Games*, 2017. [Online]. Available: <https://ruomoplus.lib.uom.gr/handle/8000/1120>
- [20] C. Politowski, F. Petrillo, J. E. Montandon, M. T. Valente, and Y.-G. Guéhéneuc, “Are game engines software frameworks? a three-perspective study,” *Journal of Systems and Software*, vol. 171, p. 110846, 2021. [Online]. Available: <https://doi.org/10.1016/j.jss.2020.110846>
- [21] G. C. Ullmann, Y.-G. Guéhéneuc, F. Petrillo, N. Anquetil, and C. Politowski, “Sydra: An approach to understand game engine architecture,” *Entertainment Computing*, vol. 52, p. 100832, 2025. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1875952124002003>
- [22] P. Petridis, I. Dunwell, D. Panzoli, S. Arnab, A. Protopsaltis, M. Hendrix, and S. de Freitas, “Game engines selection framework for high-fidelity serious applications,” *International Journal of Interactive Worlds*, 2012. [Online]. Available: <https://pureportal.coventry.ac.uk/files/3934163/Games%20engines%20selection%20framework.pdf>

- [23] A. Pattrasitidecha, “Comparison and evaluation of 3d mobile game engines,” Master’s thesis, Chalmers University of Technology, Department of Computer Science and Engineering, Gothenburg, Sweden, 2014, master’s Thesis. [Online]. Available: <https://publications.lib.chalmers.se/records/fulltext/193979/193979.pdf>
- [24] C. Ioannidis and A.-M. Boutsis, “Multithreaded rendering for cross-platform 3d visualization based on vulkan api,” *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. 44, pp. 57–62, 2020. [Online]. Available: <https://isprs-archives.copernicus.org/articles/XLIV-4-W1-2020/57/2020/>
- [25] Z. Xu, “Parallel technology and development in the gaming industry,” in *Proceedings of the 2nd International Conference on Data Analysis and Machine Learning - Volume 1: DAML*, INSTICC. SciTePress, 2024, pp. 232–238. [Online]. Available: <https://doi.org/10.5220/0013514900004619>
- [26] A. Asaduzzaman and H. Y. Lee, *GPU computing to improve game engine performance*. Bandung Institute of Technology, 2014. [Online]. Available: <https://soar.wichita.edu/bitstreams/7d194192-ade2-495c-9a93-cd3d9290f28d/download>
- [27] N. M. Talha, M. R. Babu, and M. Khalid, “Performance analysis of gaming application on multicore architecture.” *International Journal of Advanced Research in Computer Science*, vol. 2, no. 3, 2011. [Online]. Available: [https://www.researchgate.net/profile/M-Babu-2/publication/286932629\\_Performance\\_Analysis\\_of\\_Gaming\\_Application\\_on\\_Multicore\\_Architecture/links/567117ef08ae0d8b0cc2bdd1/Performance-Analysis-of-Gaming-Application-on-Multicore-Architecture.pdf](https://www.researchgate.net/profile/M-Babu-2/publication/286932629_Performance_Analysis_of_Gaming_Application_on_Multicore_Architecture/links/567117ef08ae0d8b0cc2bdd1/Performance-Analysis-of-Gaming-Application-on-Multicore-Architecture.pdf)

