
Motor de Simulación Euler para Primitivos 2D y 3D basado en Nodos y Scripts

Alejandro José Martínez de León



UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería



**Motor de Simulación Euler para Primitivos 2D y 3D basado
en Nodos y Scripts**

Trabajo de graduación presentado por Alejandro José Martínez de León
para optar al grado académico de Licenciado en Ingeniería en Ciencias
de la Computación y Tecnologías de la Información

Guatemala,

2025

UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería



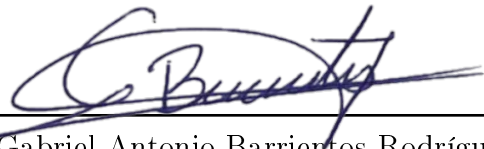
**Motor de Simulación Euler para Primitivos 2D y 3D basado
en Nodos y Scripts**

Trabajo de graduación presentado por Alejandro José Martínez de León
para optar al grado académico de Licenciado en Ingeniería en Ciencias
de la Computación y Tecnologías de la Información

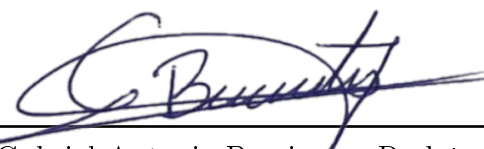
Guatemala,

2025

Vo.Bo.:

(f) 
PhD. Gabriel Antonio Barrientos Rodríguez

Tribunal Examinador:

(f) 
PhD. Gabriel Antonio Barrientos Rodríguez


(f)
Ing. Sandra Lucrecia Gomez Rodas

Fecha de aprobación: Guatemala, Noviembre de 2025.

Lista de figuras	III
Resumen	IV
Abstract	VI
1. Introducción	1
2. Justificación	3
3. Objetivos	4
3.1. Objetivo general	4
3.2. Objetivos específicos	4
4. Marco teórico	5
4.1. Simulación	5
4.1.1. Casos de uso	5
4.1.2. Ecuaciones diferenciales ordinarias	6
4.1.3. Integración Euler	7
4.2. Computación y programación	9
4.2.1. Discretización	9
4.2.2. Simulación en tiempo real	9
4.2.3. Visualización	9
4.2.4. Arquitectura de software	14
5. Metodología	17
5.1. Módulo base	17
5.2. Módulo motor	18
5.2.1. Diseño de la interfaz gráfica	18
5.2.2. Renderizado	18
5.2.3. Arquitectura de nodos	19
5.2.4. <i>Scripts</i> y extensión en C++	22
5.2.5. Archivos e interacción	25

5.2.6.	Optimización y expansión	26
5.3.	Demostraciones y casos de uso	27
5.3.1.	Únicamente nodos	27
5.3.2.	Únicamente <i>scripts</i>	27
5.3.3.	Una combinación de ambos	27
6.	Resultados	28
6.1.	Sistema de archivos	28
6.2.	Sistema de nodos	29
6.3.	Sistema de <i>scripting</i>	30
6.4.	Demostraciones	31
7.	Discusión de resultados	38
8.	Conclusiones	40
9.	Bibliografía	42

Lista de figuras

1.	Visualización del método Euler [5]	8
2.	Primitivos 2D [12]	10
3.	Primitivos 3D [17]	11
4.	Objetos complejos [24]	13
5.	Vertex y Fragment Shader [26]	14
6.	Grafo de nodos simple	15
7.	Sistema de <i>scripting</i> - clase de interfaz	24
8.	Sistema de archivos - nodo	26
9.	Sistema de archivos - punteros	26
10.	Demostración: sistema de archivos	29
11.	Demostración: sistema de nodos	30
12.	Demostración: sistema de <i>scripting</i>	31
13.	Demostración 1	32
14.	Demostración 2	32
15.	Demostración 3	33
16.	Demostración 4	33
17.	Demostración 5	34
18.	Demostración 6	34
19.	Demostración 7	35
20.	Demostración 8	35
21.	Demostración 9	36
22.	Demostración 10	36
23.	Demostración 11	37
24.	Demostración 12	37

Un motor de simulación Euler es una herramienta computacional diseñada para crear, controlar y ejecutar simulaciones interactivas. Este tipo de motor combina un sistema de nodos visuales que representan operaciones, transformaciones o entidades con lenguajes de *scripting* para permitir una mayor flexibilidad y dinamismo en la definición de comportamientos. Dicho motor permite a usuarios de distintos niveles de conocimiento poder visualizar y aprender distintos conceptos matemáticos físicos y de programación, en entornos bidimensionales y tridimensionales. Los nodos, *scripts* y el código abierto permiten la modificación y extensión del programa para cumplir con las necesidades de los usuarios en distintos campos, como la matemática y la programación.

Los primitivos 2D y 3D son formas geométricas básicas utilizadas como bloques de construcción en entornos gráficos y simulaciones. En 2D, estas primitivas incluyen elementos como círculos, rectángulos y líneas, mientras que en 3D se extienden a esferas, cubos, cilindros y planos. Estas formas permiten representar objetos más complejos y sirven como base para aplicar transformaciones, colisiones y dinámicas físicas.

Los sistemas de nodos ofrecen una interfaz visual donde los usuarios pueden construir lógica y flujos de datos conectando nodos que representan funciones, variables o entidades del entorno simulado. Esta técnica facilita el desarrollo visual, especialmente para usuarios sin experiencia avanzada en programación.

El *scripting* complementa el sistema de nodos al permitir una programación más precisa y controlada mediante lenguajes como C++ o Python. Esto permite crear simulaciones complejas con lógica condicional, manipulación de datos y eventos personalizados.

El uso del método de Euler para simular movimientos y colisiones de estos elementos tiene limitaciones computacionales, ya que su precisión disminuye con interacciones rápidas o fuerzas intensas, lo que puede causar errores acumulativos y comportamientos inestables si no se ajusta adecuadamente el tamaño del paso temporal. Al ser un motor generalizado, muchas optimizaciones no son posibles de implementar lo cual lo limita en cuanto grandes escalas dependiendo el sistema del usuario que lo utilice.

El resultado esperado es tener una herramienta que permita crear escenarios sencillos, vi-

sualmente descriptivos con los nodos, que se puedan expandir y refinar en el área de *scripting*. La simulación más sencilla sería tener una partícula que es afectada por gravedad, mientras que su expansión sería implementar un sistema de múltiples partículas que interactúan entre sí y con una escena.

An Euler simulation engine is a computational tool designed to create, control, and execute interactive simulations. This type of engine combines a system of visual nodes — representing operations, transformations, or entities, with scripting languages to allow greater flexibility and dynamism in defining behaviors. Such an engine enables users of various knowledge levels to visualize and learn different mathematical, physical, and programming concepts in two- and three-dimensional environments. The nodes, scripts, and open-source nature of the code allow modification and extension of the program to meet the needs of users in different fields, such as mathematics and programming.

2D and 3D primitives are basic geometric shapes used as building blocks in graphical environments and simulations. In 2D, these primitives include elements such as circles, rectangles, and lines, while in 3D they extend to spheres, cubes, cylinders, and planes. These shapes make it possible to represent more complex objects and serve as the foundation for applying transformations, collisions, and physical dynamics.

Node systems provide a visual interface where users can build logic and data flows by connecting nodes that represent functions, variables, or entities within the simulated environment. This approach facilitates visual development, especially for users without advanced programming experience.

Scripting complements the node system by allowing more precise and controlled programming through languages such as C++ or Python. This enables the creation of complex simulations with conditional logic, data manipulation, and custom events.

The use of the Euler method to simulate motion and collisions of these elements has computational limitations, as its accuracy decreases with rapid interactions or strong forces. This can lead to cumulative errors and unstable behavior if the time step size is not properly adjusted. Since it is a generalized engine, many optimizations cannot be implemented, which limits its scalability depending on the user's system.

The expected outcome is a tool that allows the creation of simple, visually descriptive scenarios using nodes, which can then be expanded and refined in the scripting area. The simplest simulation would involve a particle affected by gravity, while an extended version

would implement a system of multiple particles interacting with each other and with a scene.

La simulación computacional se ha consolidado como una herramienta fundamental para la enseñanza y comprensión de conceptos complejos en ciencias, matemáticas e ingeniería. Sin embargo, existe una brecha significativa entre las herramientas profesionales de simulación, que requieren conocimientos técnicos avanzados, y las necesidades de estudiantes y educadores que buscan experimentar con fenómenos dinámicos sin enfrentar barreras tecnológicas prohibitivas. Este trabajo presenta el desarrollo de un motor de simulación computacional basado en el método de integración numérica de Euler [1] que aborda esta problemática mediante una arquitectura innovadora dual: un sistema visual de nodos que permite construir lógica mediante grafos, y un sistema de *scripting* en C++ que facilita la extensión y personalización avanzada. El motor permite crear, visualizar y analizar simulaciones de primitivos geométricos 2D y 3D como círculos, rectángulos y esferas, proporcionando una plataforma accesible para usuarios de múltiples niveles de experiencia que pueden comenzar con herramientas visuales intuitivas y progresar hacia programación personalizada según sus necesidades.

La arquitectura del sistema se estructura en dos módulos principales que trabajan sinérgicamente. El módulo base integra bibliotecas especializadas para operaciones matemáticas vectoriales y para la interfaz gráfica, proporcionando infraestructura fundamental que incluye estructuras de datos observables, manejo de concurrencia y sistemas de logging. El módulo motor constituye el núcleo del sistema, implementando un editor visual de nodos. El sistema de *scripting* complementa esta arquitectura permitiendo carga dinámica de bibliotecas DLL compiladas en C++, ofreciendo rendimiento nativo, acceso directo a estructuras internas del motor. El renderizado emplea el pipeline programable de OpenGL con shaders personalizables en GLSL, mientras que el sistema de persistencia utiliza un formato de archivo jerárquico legible por humanos que emplea mapeo de punteros para preservar conexiones entre nodos.

El desarrollo culminó en un sistema funcional validado mediante doce demostraciones exhaustivas que abarcan desde visualizaciones simples hasta simulaciones complejas de partículas, evidenciando la versatilidad del motor para escalar desde casos educativos básicos

hasta aplicaciones técnicamente sofisticadas. Los resultados demuestran que la herramienta cumple satisfactoriamente con sus objetivos de simulación, visualización y gestión de escenas, proporcionando reproducibilidad y facilitando el aprendizaje progresivo de conceptos fundamentales en física computacional y métodos numéricos. El código fuente está disponible como proyecto de código abierto para fomentar transparencia académica y colaboración comunitaria, con una versión estable para Windows 11 y potencial de extensión multiplataforma. [2]

Este trabajo representa una contribución significativa al campo de herramientas educativas computacionales, demostrando que es posible crear sistemas técnicamente rigurosos que permanecen accesibles, equilibrando simplicidad de uso mediante programación visual con capacidad de profundización mediante *scripting* avanzado, estableciendo así fundamentos sólidos para futuras mejoras en métodos de integración, sistemas de evaluación más complejos y ampliaciones de los demás módulos.

Desde educación básica hasta cursos universitarios como Modelación y Simulación, el aprendizaje depende de la habilidad de todos los involucrados de poder imaginar o visualizar cómo funcionan los conceptos que se quieren impartir. En esto surge una necesidad de un motor de simulación que sea accesible para usuarios de múltiples niveles de conocimiento.

Andamiaje cognitivo progresivo El diseño dual (nodos/*scripts*) facilita una curva de aprendizaje natural: los usuarios pueden comenzar experimentando con nodos pre-construidos para desarrollar intuición, y progresivamente avanzar hacia la programación de comportamientos personalizados, consolidando su comprensión técnica.

Puente entre teoría y práctica La visualización interactiva de primitivos geométricos (esferas, cubos, cilindros, planos) moviéndose según leyes físicas programadas permite a los estudiantes verificar hipótesis, experimentar con parámetros y desarrollar intuición física que complementa el conocimiento teórico.

Fundamento en métodos numéricos Al centrarse en el método de Euler, el proyecto no solo sirve como herramienta de simulación, sino como plataforma educativa para comprender los principios fundamentales del análisis numérico, preparando a los estudiantes para métodos más sofisticados como Runge-Kutta o Verlet [3].

3.1. Objetivo general

Desarrollar un motor de simulación Euler basado en nodos y *scripts* que permita la creación modular de simulaciones físicas con primitivos 2D y 3D, reduciendo la barrera de entrada para desarrolladores de todo nivel.

3.2. Objetivos específicos

- Desarrollar una herramienta de simulación y visualización 2D y 3D para elementos primitivos con capacidad de guardar y cargar la escena.
- Diseñar la herramienta adecuadamente para la enseñanza en distintos campos, especialmente Física Matemática y Computación.
- Demostrar el funcionamiento, capacidad y flexibilidad de la herramienta por medio de múltiples archivos y *scripts* de ejemplo.

Un motor de simulación constituye una infraestructura computacional especializada diseñada para modelar, predecir y visualizar el comportamiento de sistemas dinámicos complejos mediante representaciones numéricas aproximadas. En el contexto de la computación gráfica y la física computacional, estos sistemas permiten la representación de interacciones complejas que se rigen bajo leyes físicas naturales o arbitrarias definidas por un programador, incluyendo fenómenos como el movimiento, la gravitación, las colisiones, la fricción, etc.

Los motores de simulación utilizan modelos matemáticos sofisticados que aproximan las leyes fundamentales del mundo físico o implementan reglas específicas definidas por el desarrollador. Estas simulaciones han demostrado ser fundamentales tanto en entornos académicos como en la industria del entretenimiento, con aplicaciones que se extienden desde videojuegos hasta visualización científica, ingeniería aeroespacial y medicina.

El objetivo principal de un motor de simulación es mantener y actualizar coherentemente el estado de múltiples objetos a lo largo del tiempo, respondiendo a fuerzas externas, entradas del usuario, condiciones ambientales y otras variables programadas del sistema.

4.1. Simulación

Las simulaciones computacionales han revolucionado múltiples campos del conocimiento y la industria. En el ámbito académico y científico, estas herramientas han permitido avances significativos en la comprensión de fenómenos complejos que serían imposibles o extremadamente costosos de estudiar experimentalmente.

4.1.1. Casos de uso

En la industria aeroespacial, las simulaciones de dinámica de fluidos computacional han permitido mejoras sustanciales en el diseño de vehículos aéreos y espaciales mediante el

modelado preciso de turbulencia, flujo de aire y transferencia de calor. Estas simulaciones han reducido significativamente los costos de desarrollo y testing físico, permitiendo iteraciones de diseño más rápidas y económicas [4].

En el sector automotriz, las simulaciones de crash-testing y análisis de elementos finitos han mejorado la seguridad vehicular mientras reducen la necesidad de prototipos físicos costosos. La simulación de sistemas de suspensión, aerodinámica vehicular y comportamiento de neumáticos ha optimizado tanto el rendimiento como la eficiencia energética.

El ámbito del entretenimiento, aunque menos crítico desde una perspectiva científica, se ha beneficiado enormemente de estas tecnologías. La industria cinematográfica utiliza simulaciones avanzadas para crear efectos especiales realistas, desde simulación de fluidos y partículas hasta destrucción de estructuras complejas. En videojuegos, las simulaciones físicas proporcionan interactividad realista e inmersión, desde sistemas de partículas hasta simulación de telas y cuerpos rígidos.

4.1.2. Ecuaciones diferenciales ordinarias

Las ecuaciones diferenciales ordinarias (EDO) constituyen la base matemática fundamental para modelar el comportamiento dinámico de sistemas físicos, biológicos, económicos y computacionales a lo largo del tiempo [1]. Una EDO establece una relación matemática entre una función desconocida y sus derivadas, describiendo cómo cambia una cantidad específica con respecto al tiempo u otra variable independiente.

Matemáticamente, una EDO de primer orden se puede expresar como:

$$\frac{dy}{dt} = f(t, y) \quad (1)$$

donde:

- y representa la función desconocida
- t es la variable independiente (típicamente el tiempo)
- $f(t, y)$ describe la tasa de cambio instantánea del sistema.

En simulaciones físicas, estas ecuaciones permiten predecir la evolución en el tiempo de un sistema a partir de un conjunto bien definido de condiciones iniciales. Por ejemplo, en la simulación del movimiento de un proyectil bajo la acción gravitacional, las EDO modelan las relaciones fundamentales entre posición, velocidad y aceleración:

$$\frac{d^2x}{dt^2} = 0 \quad (2)$$

$$\frac{d^2y}{dt^2} = -g \quad (3)$$

- (2) Ésta ecuación que describe un movimiento horizontal simple
- (3) Ésta ecuación que describe un movimiento vertical simple bajo gravedad

En simulaciones térmicas, las EDO describen la variación temporal de temperatura mediante la ecuación de conducción de calor. En sistemas de partículas, modelan la dinámica de múltiples entidades interactuantes mediante sistemas acoplados de ecuaciones diferenciales. Y es así que este tipo de ecuaciones se pueden utilizar para simular todo tipo de propiedades y valores, y su evolución a lo largo del tiempo.

Dado que la mayoría de las EDO que describen sistemas físicos realistas no poseen soluciones analíticas exactas, las simulaciones computacionales recurren a métodos numéricos de integración para aproximar sus soluciones en pasos temporales discretos. Esta discretización temporal convierte el problema matemático continuo en uno computacionalmente tratable, permitiendo su implementación eficiente en motores de simulación.

4.1.3. Integración Euler

El método de Euler, desarrollado por Leonhard Euler en el siglo XVIII, representa una de las técnicas de integración numérica más fundamentales y ampliamente utilizadas para resolver ecuaciones diferenciales ordinarias en sistemas de simulación [3].

Este método numérico estima el valor futuro de una variable de estado (como posición, velocidad o temperatura) utilizando la derivada actual y el valor presente del sistema. La formulación matemática básica del método de Euler es:

$$x(t + \Delta t) = x(t) + \Delta t \cdot f(x(t), t) \quad (4)$$

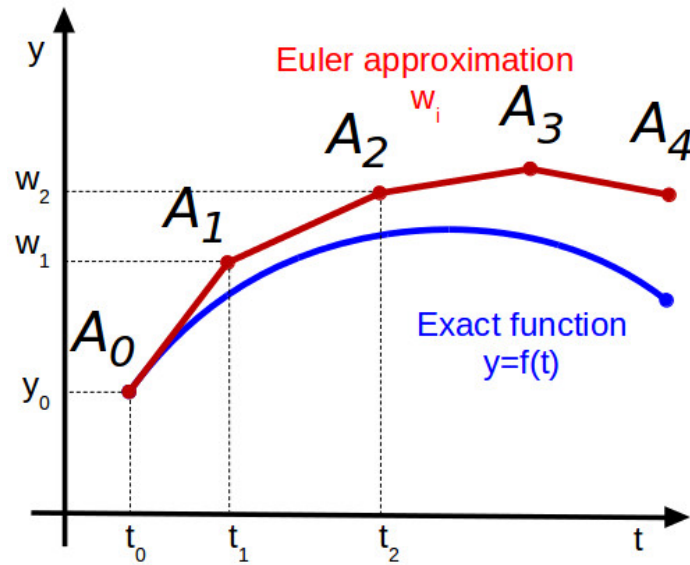
donde:

- $x(t)$ representa el estado actual del sistema
- $x(t + \Delta t)$ es el estado estimado en el siguiente paso temporal
- Δt es el tamaño del paso de integración
- $f(x(t), t)$ es la función que describe la derivada del sistema

Esta aproximación se puede observar claramente en la Fig. 1 donde:

- la línea azul representa la función exacta
- la línea roja representa cinco pasos Euler (A0 - A4), y la aproximación que esto genera.

Figura 1: Visualización del método Euler [5]



Ventajas del Método de Euler

La simplicidad conceptual y computacional del método de Euler lo convierte en una elección popular para motores de simulación en tiempo real, donde la eficiencia computacional frecuentemente toma precedencia sobre la precisión numérica absoluta. [6]

- **Simplicidad de implementación:** requiere únicamente operaciones aritméticas básicas.
- **Bajo costo computacional:** complejidad lineal en operaciones por paso temporal $O(n)$.
- **Estabilidad conceptual:** comportamiento predecible y comprensible.

Limitaciones y consideraciones

La principal desventaja del método de Euler se encuentra en que su estabilidad numérica puede verse comprometida cuando el paso de integración (Δt) es excesivamente grande o cuando el sistema presenta alta sensibilidad a las condiciones iniciales [7]. En sistemas altamente dinámicos, esto puede resultar en acumulación de errores numéricos o comportamientos no físicos.

A pesar de estas limitaciones, el método de Euler permanece ampliamente utilizado en simulaciones interactivas debido a su implementación sencilla, bajo costo computacional y comportamiento predecible en la mayoría de escenarios prácticos.

4.2. Computación y programación

La computación proporciona los fundamentos teóricos y prácticos que permiten simular fenómenos físicos complejos mediante representaciones digitales. A través de técnicas numéricas y estructuras de datos especializadas, es posible modelar la evolución de sistemas dinámicos de forma precisa y eficiente dentro de un entorno computacional.

4.2.1. Discretización

La discretización es el proceso esencial mediante el cual los sistemas continuos se convierten en representaciones discretas, es decir, en valores separados y finitos que permiten su utilización en el ámbito computacional. En simulaciones físicas, esto implica la conversión de ecuaciones diferenciales continuas en aproximaciones discretas que pueden ser evaluadas en computadoras digitales. [8]

Discretización temporal: el tiempo continuo se divide en intervalos discretos uniformes (Δt), donde cada paso temporal representa una aproximación del estado del sistema en un instante específico. La selección del tamaño del paso temporal implica un balance crítico entre precisión numérica y eficiencia computacional. Pasos temporales pequeños proporcionan mayor precisión pero requieren más cálculos, mientras que pasos grandes reducen el costo computacional a expensas de la exactitud.

Discretización espacial: los objetos y el espacio de simulación se representan mediante estructuras de datos discretas como grillas uniformes, árboles octales o estructuras jerárquicas de volúmenes delimitadores [9]. Esta discretización espacial facilita la detección eficiente de colisiones, consultas de proximidad y optimización de rendering.

4.2.2. Simulación en tiempo real

La simulación en tiempo real impone restricciones temporales estrictas donde cada cuadro de simulación debe completarse dentro de un presupuesto temporal fijo, típicamente 16.67 milisegundos para 60 cuadros por segundo o 33.33ms para 30 cuadros por segundo. Una simulación compleja que se desea correr en tiempo real requiere de optimizaciones extremas y simplificaciones al modelo, lo cual no es ideal para simulaciones exactas o de muy alta calidad, por lo que varios motores ofrecen distintas capacidades como offline rendering; una técnica en donde cada paso de la simulación puede tomar el tiempo necesario para resolverse, lentamente progresando y guardando los resultados para visualizar a futuro, culminando en un resultado mucho más preciso pero tardado y pesado [10].

4.2.3. Visualización

Los objetos simulados se representan mediante primitivos geométricos 2D y 3D, que constituyen representaciones básicas de formas geométricas estándar optimizadas para cálculos eficientes [11].

Primitivos 2D

En el dominio bidimensional los primitivos más comunes son las líneas, círculos y triángulos como se puede observar en la Fig. 2.

Figura 2: Primitivos 2D [12]



Los primitivos fundamentales son convenientes ya que sus funciones matemáticas para representación visual son simples y rápidas de calcular:

- **Líneas y segmentos:** un segmento de línea se define mediante dos puntos extremos $\mathbf{p}_0 = (x_0, y_0)$ y $\mathbf{p}_1 = (x_1, y_1)$. Su representación paramétrica está dada por:

$$\mathbf{p}(t) = (1 - t)\mathbf{p}_0 + t\mathbf{p}_1, \quad t \in [0, 1] \quad (5)$$

El renderizado se realiza mediante algoritmos de rasterización como el algoritmo de Bresenham [13].

- **Círculos:** un círculo con centro $\mathbf{c} = (x_c, y_c)$ y radio r satisface la ecuación implícita:

$$(x - x_c)^2 + (y - y_c)^2 = r^2 \quad (6)$$

Su representación paramétrica es:

$$\begin{cases} x(\theta) = x_c + r \cos(\theta) \\ y(\theta) = y_c + r \sin(\theta) \end{cases}, \quad \theta \in [0, 2\pi] \quad (7)$$

El renderizado eficiente se logra mediante el algoritmo del punto medio [14].

- **Polígonos:** triángulos, cuadriláteros y polígonos regulares para aproximación de formas complejas. Un polígono se define por una secuencia ordenada de vértices $\{\mathbf{v}_i\}_{i=1}^n$ donde $\mathbf{v}_i = (x_i, y_i)$. Su renderizado se realiza mediante:

- *Triangulación:* descomposición en triángulos primitivos usando algoritmos como ear clipping [15].
- *Relleno:* mediante scan-line rendering o algoritmos de llenado por semillas [16].

El área de un polígono simple se calcula mediante la fórmula del determinante:

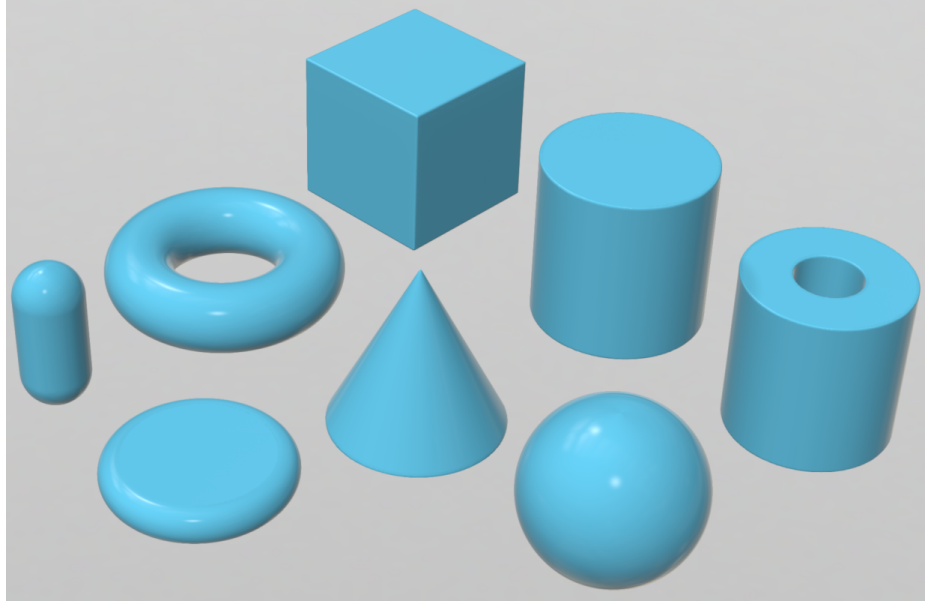
$$A = \frac{1}{2} \left| \sum_{i=0}^{n-1} (x_i y_{i+1} - x_{i+1} y_i) \right| \quad (8)$$

Los primitivos fundamentales proporcionan las bases para la representación gráfica en espacios planos y constituyen elementos esenciales en interfaces de usuario, visualización de datos y gráficos vectoriales.

Primitivos 3D

En el dominio tridimensional los primitivos más comunes son las esferas, cajas, cilindros, y otros, como se observa en la Fig. 3.

Figura 3: Primitivos 3D [17]



Al igual que las figuras bidimensionales éstas tienen fórmulas matemáticas y métodos de representación bien definidos y altamente optimizados:

- **Cajas** (Cuboides): definidas por dos puntos extremos $\mathbf{p}_{\min} = (x_{\min}, y_{\min}, z_{\min})$ y $\mathbf{p}_{\max} = (x_{\max}, y_{\max}, z_{\max})$ que forman una caja alineada con los ejes (AABB). Su volumen es:

$$V = (x_{\max} - x_{\min})(y_{\max} - y_{\min})(z_{\max} - z_{\min}) \quad (9)$$

El renderizado se realiza mediante la proyección de sus 8 vértices y 12 aristas al espacio de pantalla usando transformaciones de vista y proyección [18].

- **Esferas**: una esfera con centro $\mathbf{c} = (x_c, y_c, z_c)$ y radio r se define implícitamente por:

$$(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 = r^2 \quad (10)$$

Su representación paramétrica en coordenadas esféricas es:

$$\begin{cases} x(\theta, \phi) = x_c + r \sin(\phi) \cos(\theta) \\ y(\theta, \phi) = y_c + r \sin(\phi) \sin(\theta) \\ z(\theta, \phi) = z_c + r \cos(\phi) \end{cases} \quad (11)$$

donde $\theta \in [0, 2\pi]$ y $\phi \in [0, \pi]$. El renderizado se implementa mediante teselación en triángulos o mediante ray tracing directo [19].

- **Cilindros y conos:** un cilindro de radio r y altura h alineado con el eje z se parametriza como:

$$\begin{cases} x(\theta, z) = r \cos(\theta) \\ y(\theta, z) = r \sin(\theta) \\ z = z \end{cases}, \quad \theta \in [0, 2\pi], z \in [0, h] \quad (12)$$

Un cono con vértice en el origen y apertura α satisface:

$$x^2 + y^2 = (z \tan \alpha)^2, \quad z \in [0, h] \quad (13)$$

El renderizado emplea teselación adaptativa basada en la curvatura aparente [20].

- **Mallas poligonales:** representación mediante conjuntos de vértices $\mathcal{V}$, aristas $\mathcal{E}$ y caras $\mathcal{F}$, típicamente triángulos. Una cara triangular con vértices $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2$ se renderiza calculando su normal:

$$\mathbf{n} = \frac{(\mathbf{v}_1 - \mathbf{v}_0) \times (\mathbf{v}_2 - \mathbf{v}_0)}{\|(\mathbf{v}_1 - \mathbf{v}_0) \times (\mathbf{v}_2 - \mathbf{v}_0)\|} \quad (14)$$

y aplicando modelos de iluminación como Phong o physically-based rendering (PBR) [21].

El pipeline de renderizado 3D transforma coordenadas del espacio de modelo al espacio de pantalla mediante:

$$\mathbf{p}_{\text{pantalla}} = \mathbf{M}_{\text{viewport}} \cdot \mathbf{M}_{\text{proj}} \cdot \mathbf{M}_{\text{view}} \cdot \mathbf{M}_{\text{model}} \cdot \mathbf{p}_{\text{local}} \quad (15)$$

donde cada matriz $\mathbf{M}$ representa una transformación específica del pipeline gráfico [22].

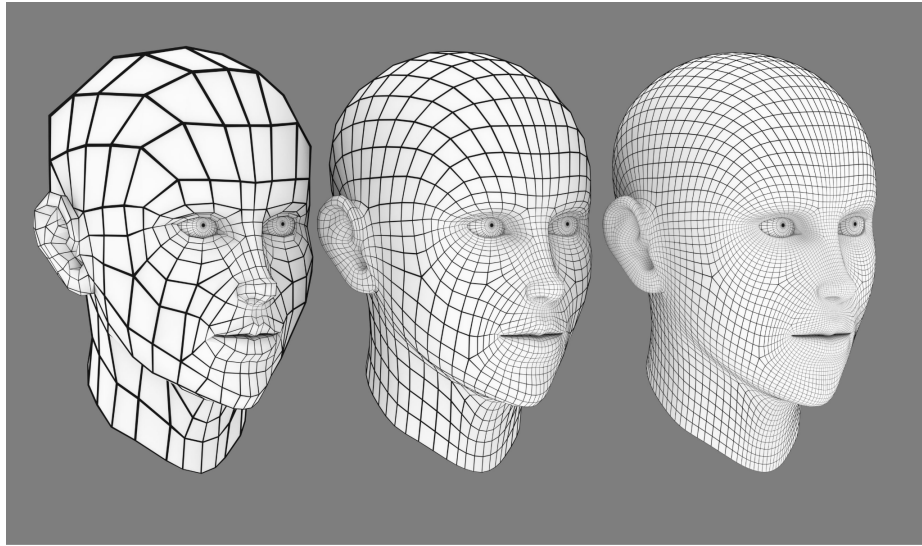
Los primitivos 3D extienden los conceptos bidimensionales incorporando profundidad volumétrica, siendo fundamentales para simulación física, renderización fotorrealista y diseño asistido por computadora.

Objetos complejos

Los objetos geométricos complejos se categorizan principalmente en:

Mallas poligonales: representaciones discretas de superficies mediante colecciones de triángulos interconectados como se observa en la Fig. 4. Los triángulos constituyen el primitivo fundamental debido a que tres puntos siempre definen un plano único, garantizando consistencia geométrica [23].

Figura 4: Objetos complejos [24]

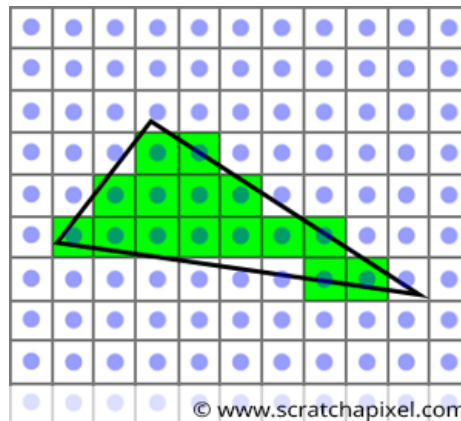


Superficies paramétricas: representaciones continuas mediante funciones matemáticas como NURBS (Non-Uniform Rational B-Splines) y Bézier surfaces, utilizadas para modelado preciso de formas orgánicas y manufacturadas.

Renderizado

El renderizado convierte las representaciones geométricas abstractas en imágenes visuales mediante el pipeline gráfico, que incluye las etapas de procesamiento de vértices, ensamblaje de primitivos, rasterización y procesamiento de fragmentos [25]. Una figura geométrica como un triángulo, entonces pasa un proceso que se llama *vertex shading* el cual lo proyecta a la pantalla, y luego *fragment shading* lo cual indica a esa sección de la pantalla como debe dar color a ese objeto. En la Fig. 5 se puede observar una grilla de puntos, representando píxeles, la figura del triángulo que se desea visualizar, y en verde los píxeles que concuerdan con esa figura, cálculo que se realiza en el vertex shader, y se envía al fragment shader para determinar el color verde.

Figura 5: Vertex y Fragment Shader [26]



Vertex Shaders:

En esta etapa, cada vértice del modelo —que define las esquinas de una figura geométrica como un triángulo o un cuadrado— se procesa individualmente mediante un *vertex shader*. El *vertex shader* aplica transformaciones geométricas, como traslaciones, rotaciones y escalados, para posicionar el objeto correctamente en el espacio tridimensional. También se realiza la proyección de las coordenadas 3D a coordenadas 2D de la pantalla, lo que permite determinar dónde aparecerá el vértice en la imagen final. Además, puede incluir cálculos de iluminación por vértice, determinando cómo la luz afecta cada punto antes de interpolar esos valores a lo largo de la superficie. Por ejemplo, si un triángulo forma parte de un modelo 3D, el *vertex shader* se encarga de calcular dónde se proyectarán sus tres vértices en la pantalla y cómo responderán ante las fuentes de luz presentes en la escena.

Fragment Shaders:

Cada fragmento pasa por el *fragment shader*, que calcula el color final de ese punto de la imagen. El *fragment shader* puede incluir técnicas de sombreado avanzadas, como Phong shading, Blinn-Phong, o el uso de texturas para añadir detalles visuales. También puede aplicar efectos como transparencias, reflejos o sombras, según las propiedades del material y las condiciones de iluminación de la escena.

4.2.4. Arquitectura de software

La arquitectura de software define la estructura fundamental de un sistema en términos de sus componentes, sus relaciones y los principios que guían su diseño y evolución. En el contexto de un motor de simulación, esta arquitectura debe permitir modularidad, escalabilidad y extensibilidad para manejar sistemas complejos de forma eficiente.

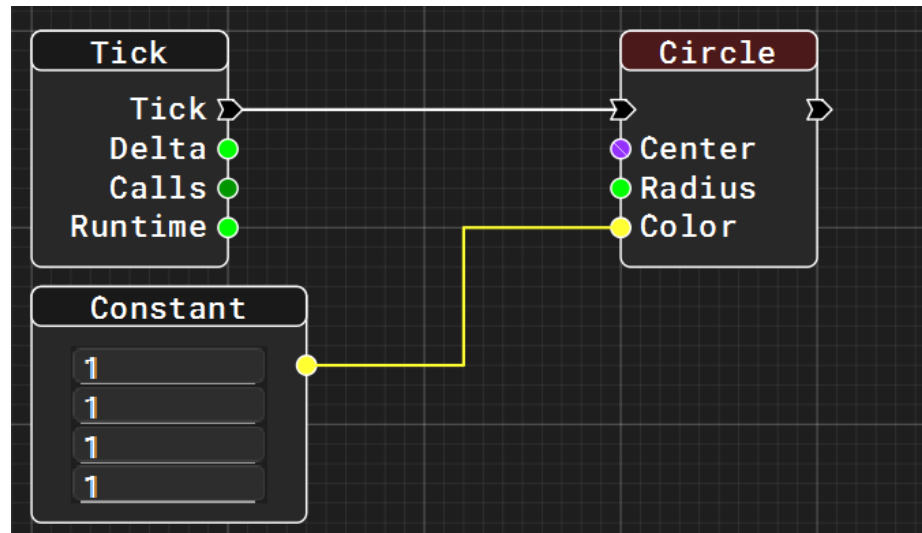
Nodos y Flow Based Programming

Para gestionar la complejidad lógica del motor de forma visual y modular, especialmente para usuarios sin conocimiento profundo de programación, se implementa el paradigma de

Flow-Based Programming (FBP) [27]. Este paradigma permite construir sistemas complejos conectando bloques funcionales (nodos) mediante grafos dirigidos acíclicos.

Cada nodo representa una unidad de operación atómica con entradas y salidas claramente definidas. Por ejemplo, un nodo de multiplicación vectorial tiene dos entradas de vectores tridimensionales y una salida de un vector resultante. El sistema completo se describe como el flujo de datos entre estos bloques interconectados, como se puede observar en la Fig. 6.

Figura 6: Grafo de nodos simple



En la Fig. 6 se puede observar un grafo simple de tres nodos. *Tick* que emite una señal de ejecución, *Circle* que recibe esa señal lo cual indica al sistema que debe mostrar un círculo en pantalla, y *Constant* que es un valor constante representando el color blanco en formato RGBA y transmitiendo esa data al nodo *Circle*.

Características del sistema de nodos:

- **Modularidad:** cada nodo encapsula una funcionalidad específica y bien definida.
- **Reutilización:** los nodos pueden ser reutilizados en múltiples contextos y configuraciones.
- **Depuración visual:** el flujo de datos es visualmente rastreable a través del grafo.

En el contexto de un motor de simulación, los nodos pueden representar:

- **Operaciones matemáticas:** suma, multiplicación, transformaciones matriciales.
- **Transformaciones físicas:** aplicación de fuerzas, integración temporal, detección de colisiones.
- **Condiciones lógicas:** evaluación de predicados, máquinas de estado finito.
- **Eventos y reacciones:** triggers, sistemas de partículas.

Scripting en C++

La implementación de sistemas de scripting en C++ como mecanismo para definir comportamientos de nodos y extender funcionalidades del motor proporciona una combinación única de rendimiento, flexibilidad y control de bajo nivel [28].

Ventajas del scripting en C++:

- **Rendimiento:** acceso directo a memoria, optimizaciones del compilador y control preciso de recursos.
- **Integración:** acceso nativo al código base del motor sin overhead de interpretación.
- **Flexibilidad:** capacidad de implementar algoritmos complejos sin restricciones del lenguaje.
- **Ecosistema:** aprovechamiento de bibliotecas existentes y herramientas de desarrollo maduras.

Implementación técnica: los sistemas de scripting en C++ suelen implementarse mediante:

- **Interfaces abstractas:** definición de contratos mediante clases base virtuales.
- **Carga dinámica:** utilización de bibliotecas dinámicas (DLL/SO) para hot-reloading de código.
- **Sistemas de reflexión:** implementación manual de metadatos para introspección de tipos.
- **Plugin architecture:** framework para carga y gestión de módulos externos.

Esta arquitectura facilita la creación de nodos personalizados en entornos basados en flujo, permitiendo expansión progresiva, desarrollo comunitario y especialización orientada al dominio específico del motor de simulación.

5.1. Módulo base

El primer módulo, denominado KL, actúa como la base fundamental del sistema. Contiene un conjunto de funciones esenciales de utilidad, principalmente orientadas a la aritmética, conversión y compatibilidad entre distintas librerías externas.

Entre las integraciones más relevantes se encuentran:

- GLM (OpenGL Mathematics): provee operaciones matemáticas y computacionales para gráficos, vectores, matrices y transformaciones. [29]
- Qt6: encargado de la interfaz gráfica de usuario (*GUI*, por sus siglas en inglés) y la comunicación entre el motor y el entorno visual. [30]

Además, KL incluye clases y herramientas de soporte que sirven como cimientos para el desarrollo del motor principal:

- Listas observables, útiles para mantener sincronización entre datos y vistas en la interfaz.
- Manejo de multi-threading mediante una cola de tareas, que permite distribuir eficientemente la carga de trabajo.
- Sistema de logging, fundamental para depuración y análisis de ejecución.
- Operadores y utilidades básicas que simplifican la escritura y mantenimiento del código.

Este módulo fue concebido inicialmente como una librería separada, con el propósito de ser reutilizable en otros proyectos. Sin embargo, en la medida que avanzó el desarrollo, se

integró estrechamente con el módulo principal, expandiendo sus capacidades con funciones de aplicación más específicas.

5.2. Módulo motor

El segundo módulo constituye el motor central del sistema, cuya construcción tomó aproximadamente cinco meses de desarrollo continuo.

5.2.1. Diseño de la interfaz gráfica

El primer paso fue diseñar el layout de la interfaz gráfica, definiendo con precisión los distintos paneles y su propósito. Cada panel cumple una función particular: desde la edición de nodos, visualización de resultados y gestión de scripts, hasta controles de depuración y monitoreo en tiempo real. Se utilizó exclusivamente la librería de QT6 para este propósito, al ser una de las librerías gráficas más completas y soportadas, al igual que tener una licencia abierta para proyectos no comerciales.

Los espacios de trabajo / paneles se consideraron para ser mínimos y obtener la operación completa de todo el sistema en una única pantalla. Dentro de ellos se tuvieron que considerar los dos espacios más importantes, el editor de nodos y la pantalla de visualización, y adicional pantallas de utilidad como un listado con los nodos disponibles para agregarlos al grafo, un listado para crear y modificar variables y dos listas para poder ver el historial de acciones, el cual está atado al sistema de *undo* / *redo*.

Adicional a los distintos espacios de trabajo también fue necesario el alto uso de la librería de QT para el sistema de nodos, ya que fue necesario configurar toda la visualización de los grafos, conexiones, nodos y demás, para poder trabajar con ellos e interactuar intuitivamente con todo el sistema.

5.2.2. Renderizado

Otra área importante de la interfaz gráfica es la visualización y renderizado de los objetos. Para todos los nodos de renderizado se utilizó el método de rasterización, el cual forma parte fundamental del pipeline gráfico de OpenGL. Este pipeline transforma los datos geométricos de los objetos en píxeles en la pantalla a través de varias etapas programables.

En las figuras 2D, la implementación de triángulos es la más sencilla, ya que solo requiere una llamada al *backend* gráfico (OpenGL) con los tres vértices del triángulo. El pipeline gráfico procesa estos vértices a través de la etapa de vertex shader y posteriormente los rasteriza. Este concepto se extiende para los cuadrados y rectángulos, simplemente utilizando dos triángulos para representar la figura, aprovechando así la naturaleza primitiva de OpenGL que trabaja eficientemente con triángulos.

Para los círculos se utilizó una optimización matemática, en la cual utilizando dos triángulos para formar un cuadrado, en el fragment shader (una etapa programable del pipeline

gráfico de OpenGL) se calcula la distancia a las orillas para determinar qué píxeles sí forman parte del círculo. El mismo método se extendió para las esferas, añadiendo para la forma 3D una fuente de luz para sombrear y dar volumen al objeto. Esta técnica aprovecha la flexibilidad del pipeline gráfico programable de OpenGL, específicamente la etapa de fragment shader donde se pueden realizar cálculos por píxel.

Todos estos fragment shaders que definen las figuras básicas están estandarizados y el motor los lee al momento de iniciar. El pipeline gráfico de OpenGL permite esta modularidad al separar el código de renderizado en shaders independientes. Por lo que modificar los archivos para una figura permite al usuario implementar su propio código de renderizado no sólo para el fondo y utilizando scripts, sino también para las figuras 2D y 3D básicas ya incluidas dentro del motor, personalizando así el comportamiento del pipeline gráfico según sus necesidades.

5.2.3. Arquitectura de nodos

Posteriormente, se abordó el aspecto más complejo del proyecto: la definición de la arquitectura de nodos y el flujo de información dentro del grafo acíclico dirigido. Este proceso, resultó ser el más demandante en tiempo y esfuerzo, dado que constituye el núcleo lógico del motor y determina su capacidad de expresividad y rendimiento.

La arquitectura de nodos se basa en el patrón de diseño de grafo de dependencias, donde cada nodo representa una operación o dato, y las conexiones entre nodos definen el flujo de ejecución y datos. Esta estructura permite la composición de comportamientos complejos a partir de bloques simples y reutilizables, facilitando el desarrollo visual de lógica computacional sin necesidad de escribir código tradicional.

Los principales objetivos en esta etapa fueron:

- Garantizar una evaluación rápida y correcta de los nodos, sin comprometer la escalabilidad.
- Diseñar un sistema flexible que permitiera la creación de nuevos nodos mediante scripts.
- Unificar ambos mundos (nodos y scripts), de forma que un nodo pudiera convertirse en el punto de entrada para la ejecución de scripts, permitiendo al usuario interactuar de manera natural con el motor.

El paso más complicado era definir los puertos de los nodos, ya que hay dos tipos base fundamentales: puertos de ejecución (execution ports) y puertos de datos (data ports). Cada tipo tiene características y comportamientos distintos que determinan cómo fluye la información a través del grafo.

Los puertos de ejecución controlan el flujo de control del programa, similar a las líneas de código en programación tradicional, mientras que los puertos de datos transportan información entre nodos. Esta separación permite una clara distinción entre cuándo "qué" se ejecuta, facilitando la comprensión visual del comportamiento del sistema.

Puertos de ejecución

Los puertos de ejecución son relativamente sencillos en su implementación; si un puerto de ejecución de salida tiene una conexión a un puerto de entrada de otro nodo, cada vez que se ejecuta el nodo al que pertenece ese puerto de salida, llama a la función virtual `execute()` a través de la conexión, lo que desencadena la ejecución del nodo siguiente en la cadena.

Este mecanismo implementa un patrón de propagación de eventos donde la ejecución fluye secuencialmente de un nodo a otro, similar al flujo de control en programación imperativa. Un nodo puede tener múltiples puertos de ejecución de salida (por ejemplo, para implementar bifurcaciones condicionales), y cuando se ejecuta, puede elegir qué puerto activar basándose en su lógica interna. Este diseño permite implementar estructuras de control complejas como condicionales, bucles y manejo de eventos de forma visual e intuitiva.

Puertos de datos

Los puertos de datos son más complicados debido a la necesidad de manejar un sistema de tipos robusto y garantizar la seguridad de tipos en las conexiones. Existen distintos tipos de datos que estas conexiones pueden transportar: valores de punto flotante de precisión simple y doble, enteros con y sin signo de diferentes tamaños (8, 16, 32, 64 bits), vectores de 2, 3 y 4 dimensiones, matrices de transformación, listas dinámicas y estáticas, booleanos, cadenas de texto, y otros tipos de información más complejos como texturas, mallas geométricas y referencias a objetos.

Los datos no son interoperables de forma implícita, siguiendo el principio de tipado fuerte. Se tienen que hacer conversiones explícitas con ayuda de distintos nodos especializados (nodos de conversión o casting) si se quiere convertir un entero en un punto flotante, o un vector en sus componentes individuales. Esta decisión de diseño, aunque puede parecer más restrictiva, previene errores sutiles y hace que las transformaciones de datos sean explícitas y visibles en el grafo, mejorando la comprensibilidad y debuggeabilidad del sistema.

En general, el proceso de evaluación es similar al de puertos de ejecución pero con una diferencia fundamental: utiliza evaluación "pull" (por demanda) en lugar de "push" (por propagación). Si un puerto de entrada de datos está conectado a un puerto de salida, cuando el nodo necesita ese dato, a través de esa conexión va a solicitar al nodo fuente que calcule y retorne el dato requerido. Este enfoque permite la evaluación perezosa: un nodo solo se evalúa cuando su resultado es necesario, y puede cachear su resultado para evitar recálculos innecesarios si ninguna de sus entradas ha cambiado.

Cada puerto de datos implementa un mecanismo de invalidación de caché que se propaga cuando los valores de entrada cambian, asegurando que siempre se obtengan valores actualizados sin recalcular innecesariamente el grafo completo. Este sistema de caché inteligente es crucial para mantener el rendimiento en grafos complejos con operaciones costosas.

Evaluación y ejecución del grafo

Una vez establecidos estos dos tipos de flujo del sistema, el siguiente paso fue poder generar un grafo con múltiples nodos interconectados y que se ejecutaran de forma correcta, respetando las dependencias entre nodos y evitando condiciones de carrera. La utilización de punteros inteligentes (smart pointers) y referencias fue clave para este paso, permitiendo una gestión automática de memoria y evitando problemas de lifetime de objetos en un sistema tan dinámico.

Se implementó un sistema de gestión de ciclo de vida de nodos que maneja la creación, destrucción y serialización de nodos de forma segura. Los nodos se almacenan en un contenedor especializado que mantiene la coherencia del grafo incluso cuando se añaden o eliminan nodos durante la ejecución.

Nodos de sistema y eventos Así mismo se definieron distintos nodos de ejecución especializados para interactuar con el ciclo de vida del sistema y responder a eventos específicos:

- **Nodo Init Scene:** se ejecuta una sola vez al iniciar o reiniciar la simulación. Es ideal para inicializar variables, cargar recursos o configurar el estado inicial del sistema. Garantiza que la lógica de inicialización se ejecute antes que cualquier otro nodo del grafo.
- **Nodo Euler Tick:** se ejecuta en un bucle continuo cada frame de la simulación. Recibe como parámetro el tiempo delta (tiempo transcurrido desde el frame anterior), permitiendo implementar animaciones y comportamientos que dependen del tiempo de forma independiente de la velocidad de refresco. Este es el nodo más utilizado para implementar lógica continua y actualizar el estado de los objetos.
- **Nodos de eventos de usuario:** se implementaron nodos que responden a eventos de entrada como clicks del mouse, teclas presionadas, movimiento del mouse, scroll, y eventos de touch en dispositivos móviles. Estos nodos permiten crear interfaces interactivas y responder a las acciones del usuario de forma reactiva.

Adicionalmente, se implementaron nodos para acceder a datos internos del sistema, proporcionando información contextual crucial para la lógica del grafo:

- **Delta:** tiempo transcurrido entre el frame actual y el anterior, expresado en segundos. Esencial para crear animaciones suaves e independientes del framerate.
- **Calls:** número total de cuadros renderizados desde el inicio de la simulación. Útil para implementar comportamientos periódicos o basados en ciclos.
- **Runtime:** tiempo total transcurrido desde el inicio de la simulación en segundos. Permite implementar temporizadores y comportamientos que dependen del tiempo absoluto.

Este conjunto de nodos de sistema forma la base sobre la cual se construyen comportamientos más complejos, proporcionando al usuario las herramientas fundamentales para crear experiencias interactivas y dinámicas sin necesidad de programación tradicional. La combinación de estos nodos básicos con nodos personalizados definidos por el usuario permite una expresividad prácticamente ilimitada del sistema. `RetryClaude` does not have the ability to run the code it generates yet.

5.2.4. *Scripts* y extensión en C++

El segundo módulo más importante del proyecto, el sistema de scripting, tuvo una implementación bastante más sencilla en comparación con el sistema de nodos, aunque no estuvo exento de desafíos técnicos significativos. La parte más complicada para este tipo de sistemas es el hecho de que C++ es un lenguaje compilado, por lo cual cualquier *script* que se desee integrar también se debe precompilar, incluyendo las librerías base y el módulo motor para poder hacer las integraciones necesarias.

Este enfoque contrasta con lenguajes de scripting interpretados como Python o Lua, comúnmente utilizados en motores de juegos, pero ofrece ventajas significativas en términos de rendimiento: los *scripts* compilados se ejecutan a velocidad nativa sin overhead de interpretación, tienen acceso directo a las estructuras de datos del motor sin necesidad de capas de binding, y pueden aprovechar todas las optimizaciones del compilador de C++.

Arquitectura del sistema de *scripts*

Para estas integraciones se utilizaron ampliamente punteros inteligentes (smart pointers) y variables estáticas para que los *scripts* pudieran acceder a las variables y clases internas del motor de forma segura y eficiente. El uso de punteros compartidos (`std::shared_ptr`) garantiza que los recursos compartidos entre el motor y los *scripts* se gestionen correctamente sin problemas de ownership o memory leaks.

Las variables estáticas actúan como puntos de acceso globales a subsistemas críticos del motor, como el sistema de renderizado, el gestor de recursos, el sistema de input y el grafo de nodos. Esta arquitectura permite que los *scripts* accedan a funcionalidades avanzadas del motor manteniendo un bajo acoplamiento y facilitando la extensibilidad del sistema.

Carga dinámica de DLLs en Windows

Para poder cargar un *script* al motor, éste se debe compilar a un archivo de librería dinámica con extensión `.dll` (Dynamic Link Library en Windows). El motor utiliza la API de Windows para carga dinámica de librerías, específicamente las funciones `LoadLibrary()`, `GetProcAddress()` y `FreeLibrary()`, que permiten cargar, acceder a símbolos y descargar DLLs en tiempo de ejecución sin necesidad de linkear estáticamente.

Este mecanismo de carga dinámica, conocido como "Windows Dynamic DLL Loading", ofrece varias ventajas cruciales para el sistema:

- **Hot-Reloading:** los *scripts* pueden ser recompilados y recargados mientras el motor está en ejecución, sin necesidad de reiniciar el programa completo. Esto acelera dramáticamente el ciclo de desarrollo, permitiendo iteraciones rápidas y pruebas inmediatas de cambios en el código.
- **Modularidad:** cada *script* es una unidad independiente que puede ser distribuida, actualizada o reemplazada sin afectar al motor base ni a otros scripts. Esto facilita la creación de plugins y extensiones modulares.
- **Versionado independiente:** diferentes versiones de un mismo *script* pueden coexistir, y el usuario puede elegir cuál cargar. Esto es útil para mantener compatibilidad hacia atrás o experimentar con diferentes implementaciones.
- **Reducción del tamaño del ejecutable:** las funcionalidades opcionales pueden distribuirse como DLLs separadas, manteniendo el ejecutable principal ligero y permitiendo descargas modulares.

El proceso de carga de un *script* sigue estos pasos técnicos:

1. **Compilación del *script*:** el usuario escribe su código C++ heredando de la clase base **SCRIPT** y lo compila como DLL usando el mismo compilador y configuración que el motor (para garantizar compatibilidad de ABI - Application Binary Interface).
2. **Carga de la DLL:** el motor utiliza `LoadLibrary()` para cargar la DLL en el espacio de memoria del proceso. Esta función retorna un handle (HMODULE) que identifica la librería cargada.
3. **Resolución de símbolos:** mediante `GetProcAddress()`, el motor busca funciones exportadas específicas en la DLL, particularmente una función factory que crea instancias de la clase *script*. Esta función debe estar marcada con `extern "C"` y `__declspec(dllexport)` para evitar name mangling y garantizar que sea visible desde fuera de la DLL.
4. **Instanciación:** se llama a la función factory obtenida, que crea dinámicamente una instancia del *script* y retorna un puntero a la interfaz base **SCRIPT**. Esta indirección a través de una función factory es necesaria porque no se pueden crear objetos de clases C++ directamente desde DLLs sin conocer su layout exacto en memoria.
5. **Registro en el sistema:** el *script* se registra en el sistema de nodos, haciéndolo disponible para su uso en el editor visual. Se almacena también el handle de la DLL para su posterior descarga.
6. **Descarga:** cuando el *script* ya no es necesario o se va a recargar, se llama a `FreeLibrary()` para liberar la DLL de memoria y permitir que el archivo sea modificado o reemplazado.

Interfaz de scripting y polimorfismo

Haciendo uso de una clase de interfaz abstracta con funciones virtuales puras, estas pueden ser sobrescritas por el usuario para modificar su funcionalidad e integrar toda

aquella lógica personalizada que se desee. Este patrón de diseño, conocido como Template Method Pattern, define el esqueleto del algoritmo de ejecución del *script* mientras permite que las subclases proporcionen implementaciones específicas de ciertos pasos.

El uso de funciones virtuales es crucial para el polimorfismo dinámico que permite al motor tratar todos los *scripts* de forma uniforme a través de la interfaz base **SCRIPT**, sin importar sus implementaciones concretas. Cuando el motor llama a un método virtual, el mecanismo de virtual table (vtable) de C++ resuelve dinámicamente a qué implementación llamar, permitiendo que cada *script* tenga su propio comportamiento único.

En la Fig. 7 se puede observar cómo está estructurada esta clase base de interfaz, y las funciones que tiene disponibles a la hora de crear el propio script.

Figura 7: Sistema de *scripting* - clase de interfaz

```
struct Script : SCRIPT {  
    Script(Session* session);  
  
    void onLoad() final override;  
  
    void onUnload() final override;  
  
    void exec(const Exec_I* port) final override;  
  
    Ptr_S<Variable> getData(const Data_O* port) final override;  
};
```

Anatomía de la clase base **SCRIPT** La estructura mostrada en la figura define el contrato que todo *script* debe cumplir. Analicemos cada componente:

- **Constructor `Script(Session* session)`:** recibe un puntero a la sesión actual del motor, que proporciona acceso al contexto de ejecución, incluyendo el grafo de nodos, el sistema de renderizado, y otros subsistemas. Este puntero se almacena internamente para uso posterior en los métodos del script. La sesión actúa como punto de entrada centralizado a todas las funcionalidades del motor.
- **Método `void onLoad()`:** función de callback que se invoca una sola vez después de que el *script* se vincula internamente al programa. Esta vinculación ocurre en dos momentos: cuando el usuario añade manualmente un nodo del *script* a la escena a través del editor, o durante la carga de un proyecto guardado mediante `File::loadBuild()`. Este método es el lugar apropiado para realizar inicializaciones pesadas, cargar recursos externos, configurar conexiones con otros sistemas, registrar eventos, o cualquier otra lógica de setup que deba ejecutarse antes de que el *script* comience a procesar datos. Es análogo a un constructor, pero con la garantía de que el motor está completamente inicializado y todos los sistemas están disponibles.
- **Método `void onUnload()`:** función de callback crítica invocada una sola vez antes de que el *script* sea destruido. Esta destrucción puede ocurrir durante el cierre nor-

mal del programa o cuando el usuario decide descargar manualmente el *script* para recompilarlo y recargarlo (hot-reload).

- **Método** `void exec(const Exec_I* port)` [OPCIONAL]: maneja la ejecución del nodo cuando se activa uno de sus puertos de ejecución de entrada. El parámetro `port` identifica cuál puerto específico fue activado, permitiendo que un *script* con múltiples puertos de entrada implemente comportamientos diferentes según cuál se ejecutó.

La responsabilidad del *script* es manejar manualmente la ejecución downstream (aguas abajo) de los puertos de ejecución de salida que desee activar. Esto se logra iterando sobre las conexiones del puerto de salida deseado y llamando a su método `execute()`. Alternativamente, si se desea activar todos los puertos de salida sin discriminación, se puede llamar al método de conveniencia `execAllDownstream()`, que es mínimamente más costoso en términos de rendimiento pero ofrece automatización completa.

Este control manual permite implementar lógica condicional compleja, donde ciertos puertos de salida se activan solo bajo condiciones específicas, habilitando bifurcaciones, loops y comportamientos complejos de flujo de control.

- **Método** `Ptr_S<Variable> getData(const Data_0* port)` [OPCIONAL]: maneja las solicitudes de datos provenientes de conexiones a los puertos de salida de datos del script. Cuando un nodo downstream necesita leer un valor de un puerto de salida de este script, este método se invoca con un puntero al puerto solicitado.

El *script* debe retornar un puntero inteligente compartido (`Ptr_S`, que es un typedef de `std::shared_ptr`) a un objeto `Variable` que encapsula el dato solicitado. La clase `Variable` es un wrapper polimórfico que puede contener cualquier tipo de dato soportado por el sistema (floats, ints, vectors, matrices, etc.).

El uso de punteros compartidos garantiza que la memoria del dato retornado se gestiona automáticamente y que el dato permanece válido mientras cualquier nodo lo esté utilizando, incluso si el *script* que lo generó se descarga.

Este método permite implementar cálculos complejos, generación procedimental de datos, transformaciones matemáticas, y cualquier otra lógica que produzca valores utilizables por otros nodos del grafo.

5.2.5. Archivos e interacción

El sistema de archivos se derivó de distintos proyectos personales, utilizando una estructura similar a un html, en el cual cada sección esta delimitada e indentada para poder determinar rápida, programática, y visualmente la jerarquía de los datos al igual que su categorización. Para determinar y poder almacenar y cargar las distintas conexiones entre los puertos de los nodos se utilizó un mapeo de punteros, lo cual significa que al momento de guardar un grafo de nodos, las direcciones en memoria de los puertos se convierten a números enteros. Al tratarse de punteros todos los puertos y sus respectivas conexiones van a tener el mismo número identificador, representando que están en el mismo espacio de memoria, y a la hora de cargar el archivo, estos espacios de memoria se traducen de los antiguos a los nuevos, preservando así estas conexiones en un formato muy simple y visualmente coherente.

Como ejemplo, en la Fig. 8 se puede observar la serialización de un nodo para renderizar una esfera, en el cual se delimitan las distintas secciones por su propósito.

Figura 8: Sistema de archivos - nodo

```
Node [ 0 ] Circle
* 1751593207584
Type RENDERING::2D::CIRCLE
( -100 -100 )
In ( 4 )
* 1751593217296
* 1751593147520
* 1751593148480
* 1751593144320
In
Out ( 1 )
* 1751590596368
Out
Data
Data
Node
```

En magenta se determina el nombre de ese nodo.

En lila se determina la categoría y tipo de nodo.

En verde su posición visual

En café los distintos punteros que contiene.

La sección **In** contiene las direcciones de los puertos de entrada y **Out** las direcciones de los puertos de salida.

Estas direcciones al momento de cargar el nodo al sistema se guardan, y utilizando la sección, visible en la Fig. 9, se puede determinar que puertos de salida estaban previamente conectados a que puertos de entrada, y reconstruir esa conexión.

Figura 9: Sistema de archivos - punteros

```
Build
Node-Data
* 1751386435472 - * 1751593144320
Node-Data
Node-Exec
* 1751590592048 - * 1751593217296
Node-Exec
Build
```

5.2.6. Optimización y expansión

Una vez establecida la arquitectura base, se inició un proceso de pulido y optimización del código, acompañado del desarrollo de una amplia librería de nodos predefinidos.

Esto con el propósito de lograr que el sistema fuera tan expresivo y completo como un lenguaje de programación tradicional, reduciendo al mínimo la necesidad de recurrir a *scripts* externos.

5.3. Demostraciones y casos de uso

Para consolidar las capacidades del motor, se elaboraron diversos *scripts* y archivos de prueba, diseñados para poner en evidencia la versatilidad del sistema y dar un punto de inicio para que experimenten los usuarios. Se desarrollaron ejemplos que utilizaban:

5.3.1. Únicamente nodos

Estas demostraciones se plantean para comprobar la solidez del sistema gráfico y su evaluación correcta. Proveen un punto de partida para usuarios nuevos y al ser únicamente nodos visuales, los archivos en esta categoría son auto-explicativos y muy graduales en su complejidad, permitiendo una curva de aprendizaje y comprensión de este sistema.

5.3.2. Únicamente *scripts*

Éstos archivos de demostración incluyen un rango de scripts pre-compilados que permiten al usuario validar la integración con el entorno de ejecución, así como proveen un andamiaje para aprender el funcionamiento del sistema de *scripting* y como interactuar con el motor.

5.3.3. Una combinación de ambos

Los archivos que combinan ambos sistemas permiten destacar el verdadero potencial del motor al permitir que nodos y *scripts* cooperen de manera fluida, integrando múltiples *scripts* que se comunican entre sí a través del sistema de nodos.

El sistema está en un estado satisfactorio. Contiene todos los módulos necesarios y planteados anteriormente para ser no sólo funcional, sino también altamente flexible, customizable, y expandible. La herramienta se encuentra en estado de código abierto en github. [2]

6.1. Sistema de archivos

El sistema de archivos, como se puede observar en la Fig. 10, es legible y editable a bajo nivel por humanos. Su propósito es ser comprensible y modular para poder depurar errores y comprender las relaciones entre los distintos sistemas guardando y desplegando los punteros de los distintos objetos.

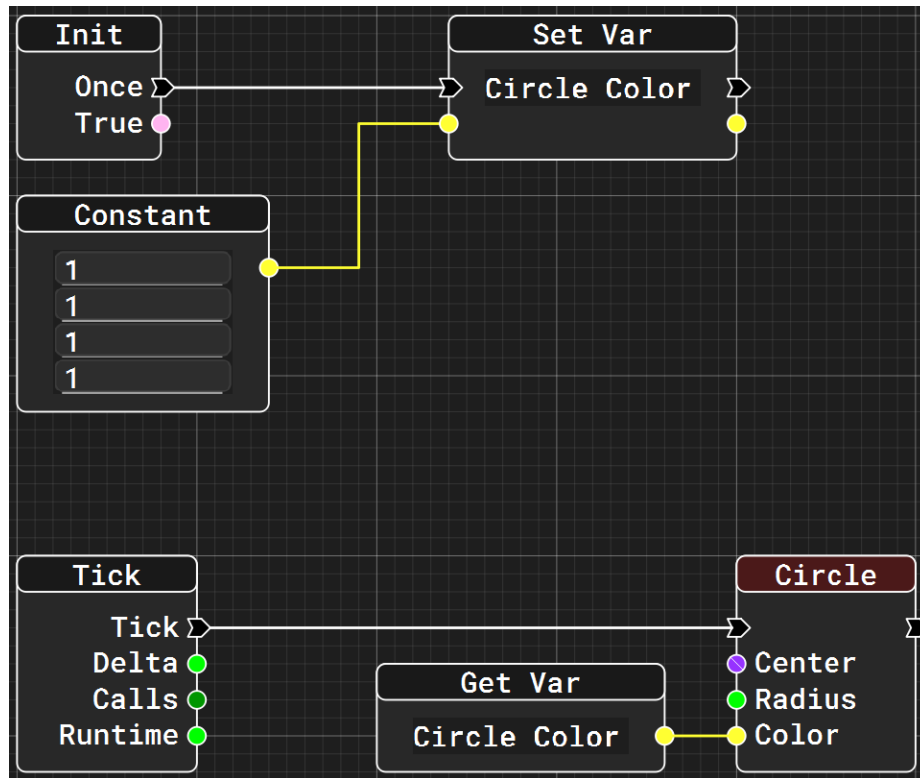
Figura 10: Demostración: sistema de archivos

```
1  Header
2  Version 0.0.1
3  Header
4  Scripts( 1 )
5  Script [ 0 ]
6  ...* 1784299603408
7  Script
8  Scripts
9  Variables( 0 )
10 Variables
11 Node-Groups( 0 )
12 Node-Groups
13 Node-Tree( 1 )
14 Node [ 0 ] GUI
15 ...* 1784299603408
16 Type SCRIPT
17 ...(-400 -100)
18 In( 0 )
19 In
20 Out( 0 )
21 Out
22 Data
23 D:/Coding/Simulator/x64/$(CONFIGURATION)/Script-GUI-Fps.dll
24 Data
25 Node
26 Node-Tree
27 Build
28 Node-Data
29 Node-Data
30 Node-Exec
31 Node-Exec
32 Build
```

6.2. Sistema de nodos

El sistema de nodos actualmente, como se puede observar en la Fig. 11, contiene varias operaciones aritméticas, utilidades para ejecución en bucles, variables, listas, renderizado y muchos más. Para complementar este sistema también se desarrolló un sistema de *undo* y *redo*, para poder revertir cambios ya sea de conexiones, mover los nodos, cambiar valores, etc.

Figura 11: Demostración: sistema de nodos



6.3. Sistema de *scripting*

El sistema de scripting contiene funciones y variables con comentarios explicando su uso y permitiendo hacer referencia a los *scripts* existentes de ejemplo. Existe un *script* básico como se puede ver en la Fig. 12, el cual contiene la base para iniciar con la primera extensión por parte del usuario.

Figura 12: Demostración: sistema de *scripting*

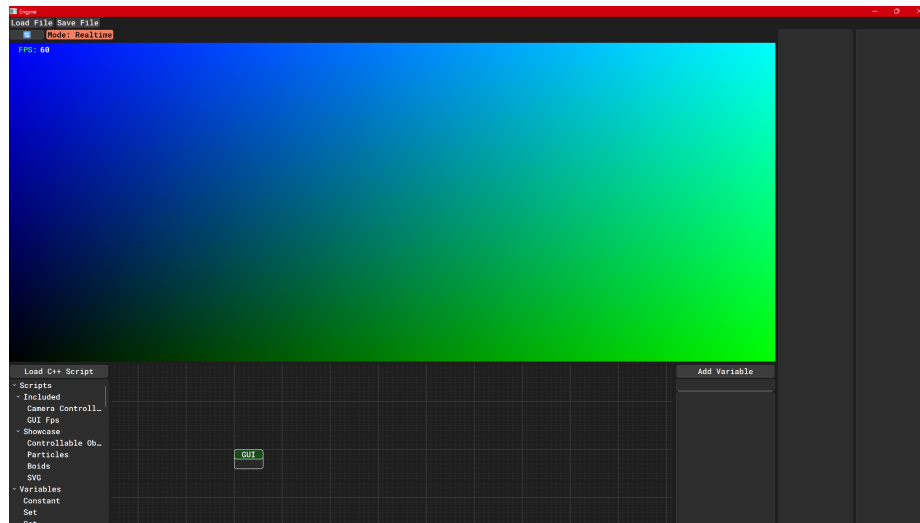
```
1      #include "Script.hpp"
2
3      Script::Script(Session* session) : SCRIPT("Base Script", session) {
4          SCRIPT_INIT;
5      }
6
7      void Script::onLoad() {
8          LOGL(<< "Loaded Base Script");
9      }
10
11     void Script::onUnload() {
12         LOGL(<< "Unloaded Base Script");
13     }
14
15     void Script::exec(const Exec_I* port) {
16         node_error = false; // To clear Highlight
17         if (missingInputs()) {
18             node_error = true; // Will Highlight Node
19             return;
20         }
21
22         execAllDownstream();
23     }
24
25     Ptr_S<Variable> Script::getData(const Data_O* port) {
26         node_error = false; // To clear Highlight
27         if (missingInputs()) {
28             node_error = true; // Will Highlight Node
29             return make_shared<Variable>();
30         }
31
32         return make_shared<Variable>();
33     }
```

6.4. Demostraciones

Utilización de Shaders GLSL en OpenGL.

En esta demostración se integra código GLSL simple dentro de la sección designada **Background** en un archivo. Dibuja una gradiente dependiendo de las coordenadas en pantalla, como se puede observar en la Fig. 13.

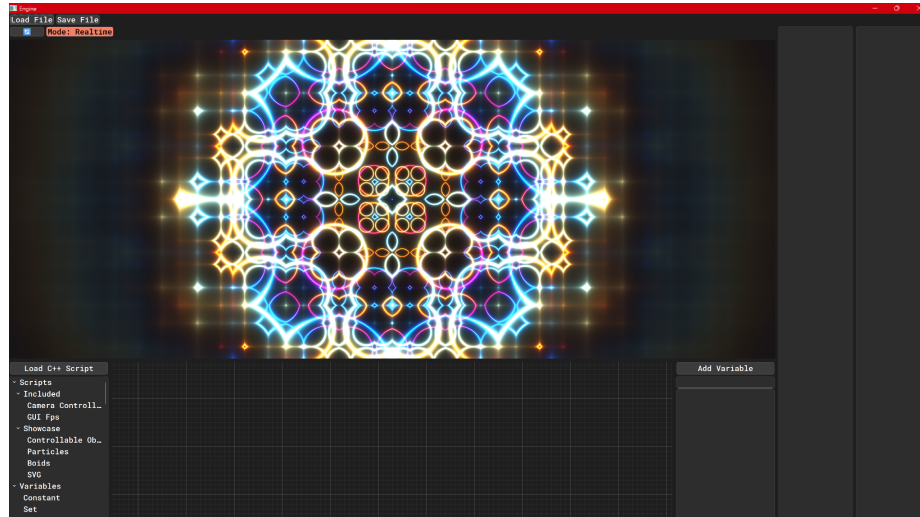
Figura 13: Demostración 1



Utilización de shaders GLSL en OpenGL.

Extendiendo la funcionalidad de la demostración previa se puede implementar el siguiente shader: <https://www.shadertoy.com/view/mtYGWY> [31]. Visible en la Fig. 14

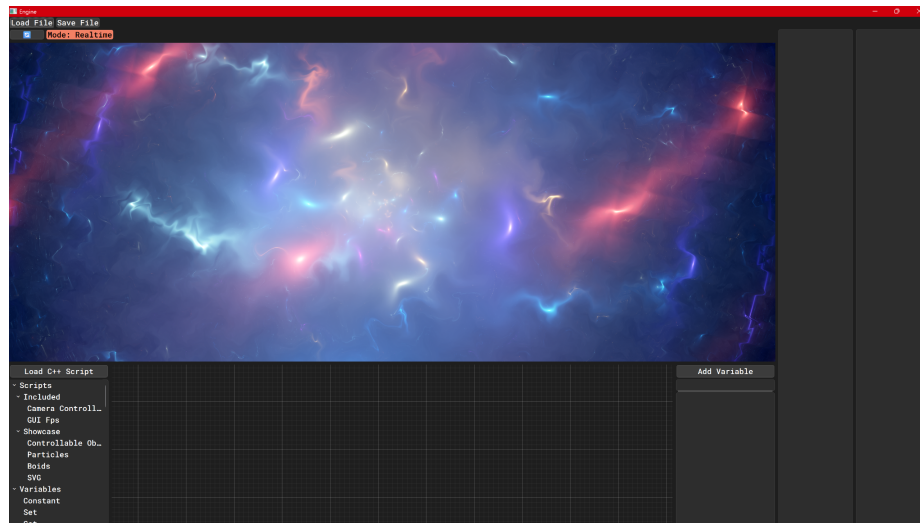
Figura 14: Demostración 2



Utilización de shaders GLSL en OpenGL.

La Fig. 15, demostrando otra implementación de un shader un poco más complejo que usa múltiples funciones matemáticas y una evolución temporal más compleja en GLSL.

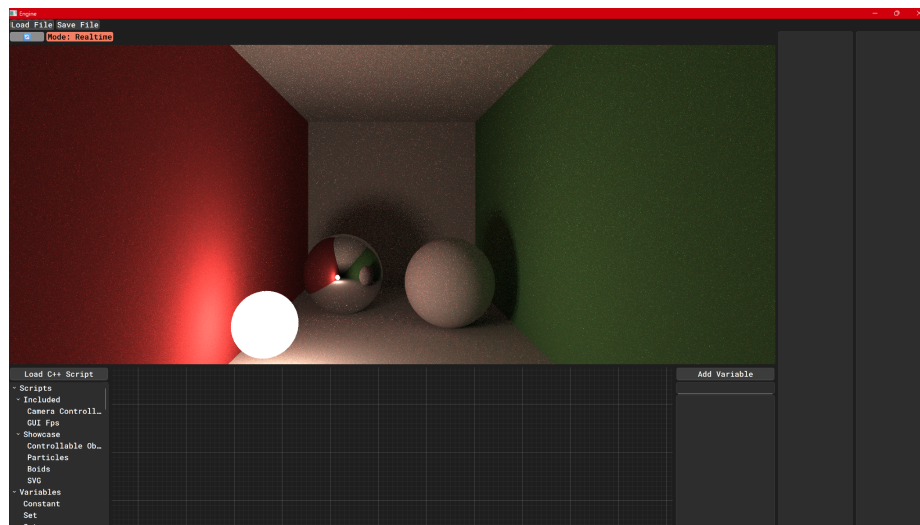
Figura 15: Demostración 3



Utilización de shaders GLSL en OpenGL.

La siguiente demostración, en la Fig. 16 es de una simulación 3D de trazado de rayos y luz para visualizar una escena simple de forma muy rápida y eficiente, demostrando la capacidad y eficiencia del renderizado del fondo. <https://www.shadertoy.com/view/4s2cWK> [32]

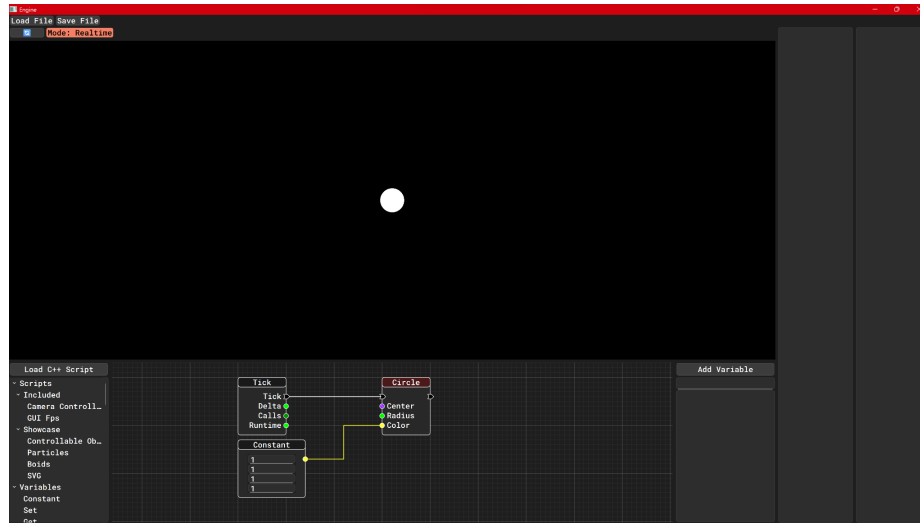
Figura 16: Demostración 4



Utilización de nodos 2D

La demostración visible en la Fig. 17 es la forma más simple del sistema de nodos, en donde únicamente se especifica que se desea renderizar un círculo en el centro de la pantalla con su posición (0,0) y su radio (3.0) por defecto, y en color blanco que obtiene del nodo constante `rgba(1,1,1,1)`

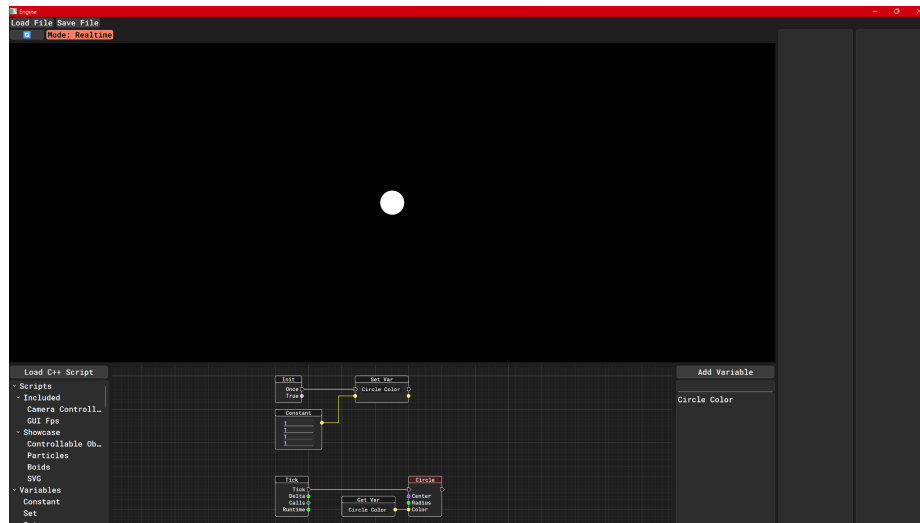
Figura 17: Demostración 5



Utilización de nodos 2D y variables

El mismo resultado que la demostración anterior, pero se demuestra la creación, asignación y utilización de una variable en la Fig. 18. Esta se asigna solo una vez al cargar el archivo o al resetear la simulación, y se lee cada vez que se quiere renderizar el círculo.

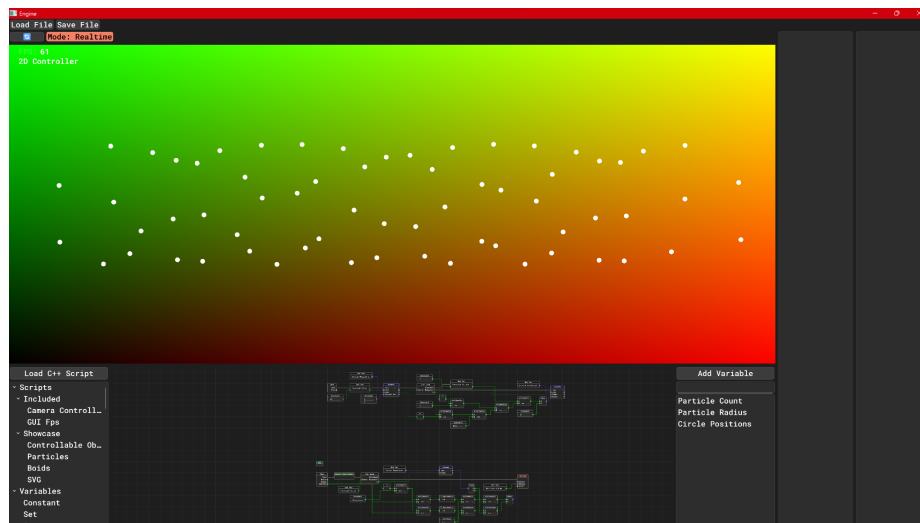
Figura 18: Demostración 6



Utilización de nodos 2D y variables complejas

En la Fig. 19 se puede observar una mezcla complicada de variables y nodos que permite renderizar círculos en un patrón designada, con variables que pueden definir la cantidad de círculos, la frecuencia en la que se mueven, y una variable de tipo lista que contiene las posiciones de todos estos círculos.

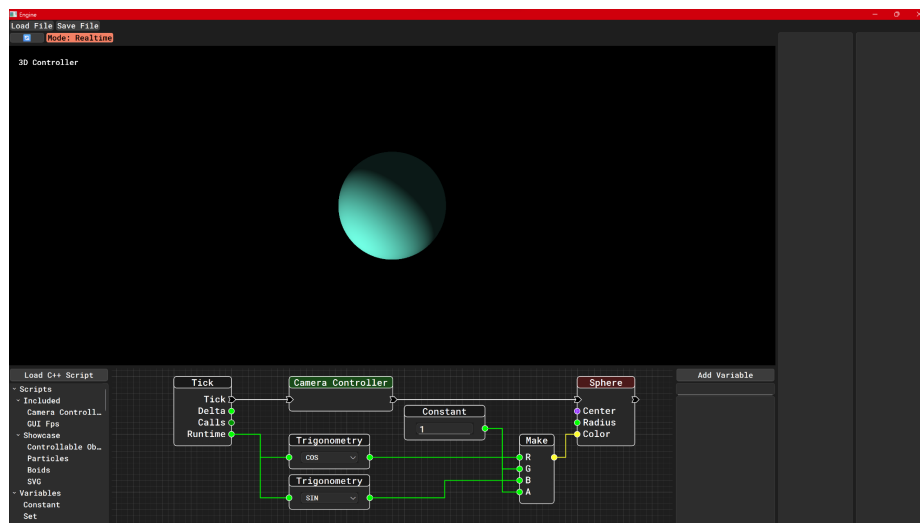
Figura 19: Demostración 7



Utilización de nodos 3D

Similar a la Demostración #5 pero utilizando una esfera en vez del círculo, y con su color fluctuando a lo largo del tiempo, esta utilización se puede ver en la Fig. 20.

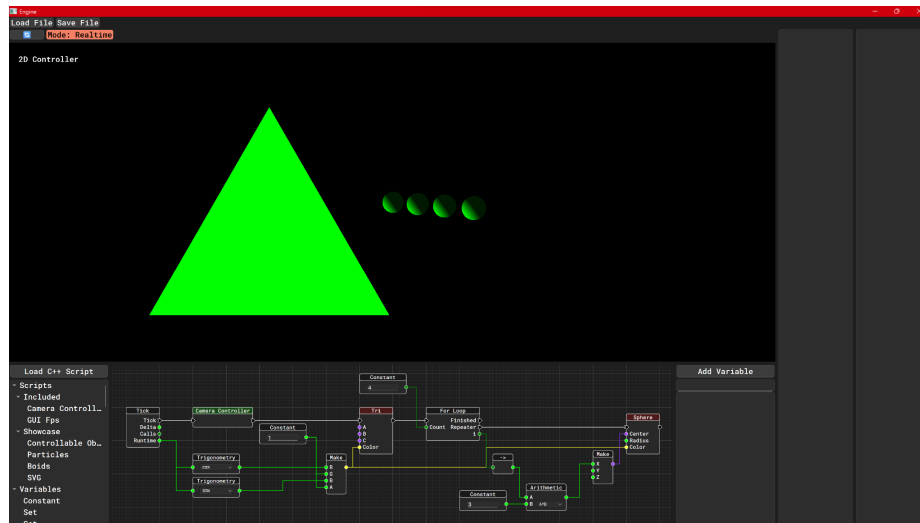
Figura 20: Demostración 8



Utilización de nodos 2D y 3D

La Fig. 21 es de un archivo que simplemente demuestra la mezcla de las funciones 2D y 3D al mismo tiempo

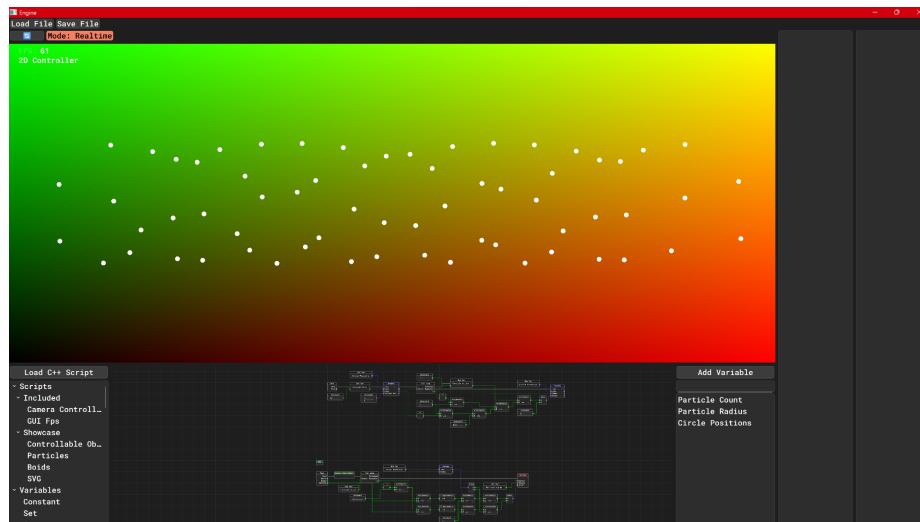
Figura 21: Demostración 9



Utilización de *scripting* y librerías externas

En la Fig. 22 se demuestra la implementación de una librería externa para renderizado de imágenes de vector, y su interacción con el sistema de nodos para definir un parámetro de resolución.

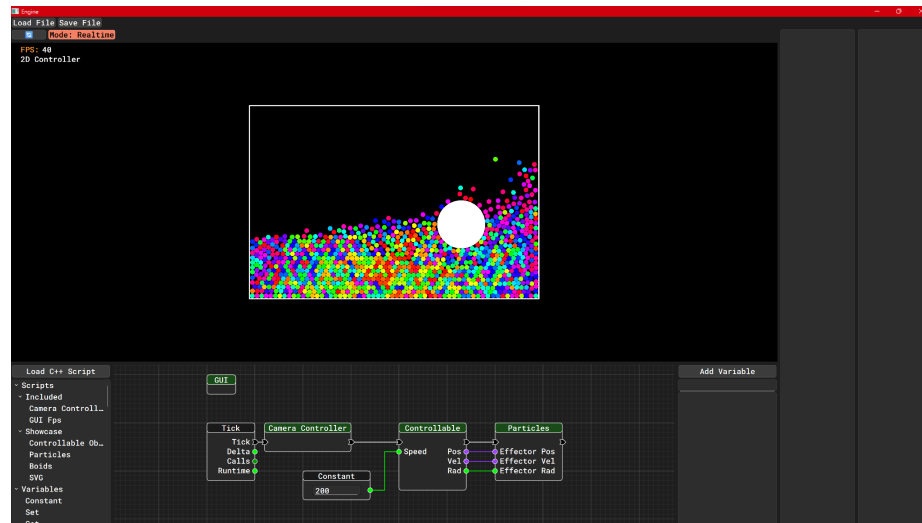
Figura 22: Demostración 10



Utilización de *scripting* e interacción con nodos

Como se observa en la Fig. 23, en este caso se demuestra la interacción entre dos *scripts* diferentes a través del sistema de nodos, y la utilidad de un *script* al poder implementar una simulación de partículas de forma mucho más rápida, eficiente y comprensible que con el sistema de nodos

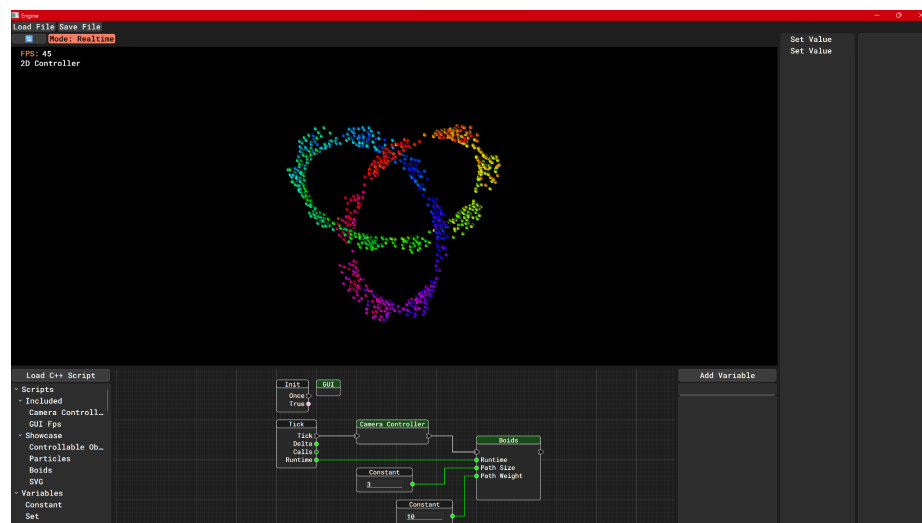
Figura 23: Demostración 11



Utilización de *scripting* para comportamiento complejo

El *script* más complejo en el cual las partículas 3D siguen un esquema de Objeto Pájaro-ide o *BOID* por sus siglas en inglés, el cual es un concepto en el cual cada partícula conoce el estado de las partículas a su alrededor y en base a ellas sigue reglas de separación, agrupación, alineación, colisión, y de seguir un camino predeterminado, similar al comportamiento de las aves. Estas fuerzas se pueden controlar con variables, y toda la simulación rota para poder lentamente observar desde todos los ángulos. Esto, visible en la Fig. 24

Figura 24: Demostración 12



Discusión de resultados

La herramienta desarrollada cumple satisfactoriamente con las funciones de simulación y visualización en entornos 2D y 3D, permitiendo la manipulación de elementos primitivos y la correcta gestión de escenas mediante las funciones de guardado y carga. Estas capacidades garantizan la reproducibilidad de las simulaciones y la posibilidad de continuar el trabajo en distintos momentos sin pérdida de información, lo cual resulta muy importante tanto para la experimentación como para la enseñanza.

En cuanto a su arquitectura, el sistema es bastante robusto dada su flexibilidad y capacidad de extensión, lo cual resultado de su diseño modular y de la implementación de scripting nativo en C++. Estas características permiten que usuarios con conocimientos técnicos puedan ampliar las funcionalidades del motor, crear nuevos comportamientos o integrar sistemas personalizados, adaptando la herramienta a una amplia variedad de necesidades.

No obstante, se identificó que el sistema de nodos se encuentra en un estado funcional pero limitado, presentando una estructura básica que aún no explota todo el potencial de la edición visual y programación de flujo. Si bien esto representa un punto de mejora, su impacto se ve considerablemente mitigado por la robustez del sistema de scripting, el cual ofrece una alternativa completa para la definición de comportamientos avanzados y el control detallado de las simulaciones.

Los archivos de demostración desarrollados cumplieron con su propósito. Estos ejemplos proveen un punto de partida accesible para los usuarios nuevos, ilustrando las funciones principales del motor, así como escenarios más complejos que evidencian la versatilidad y potencia del entorno de simulación. Estos casos de uso sirven no solo como documentación práctica, sino también como base para futuras ampliaciones o validaciones experimentales.

Respecto a la distribución y despliegue, actualmente la herramienta cuenta con una versión ejecutable estable para Windows 11. Las librerías empleadas ofrecen compatibilidad multiplataforma si un usuario deseara adaptarlo a otro ambiente, pero este desarrollo se concentró en este sistema operativo por razones de alcance. Dada la naturaleza del proyecto y los objetivos planteados, esta decisión es técnicamente adecuada y permite mejorar la

estabilidad y consistencia de la versión presentada.

Adicionalmente, el código abierto del proyecto representa un componente de alto valor académico y técnico. Este enfoque promueve la colaboración, la transparencia y la posibilidad de evolución comunitaria del motor, permitiendo que otros desarrolladores o investigadores puedan contribuir con mejoras, nuevas funciones o integraciones.

En resumen, los resultados obtenidos demuestran que la herramienta alcanza un nivel funcional sólido dentro del alcance definido, combinando flexibilidad, rendimiento y claridad conceptual. Aunque existen áreas de mejora, principalmente en el sistema de nodos y en la distribución multiplataforma, la base implementada es un punto de partida robusto y extensible para el desarrollo futuro de un motor de simulación modular, educativo y de propósito general.

Se desarrolló un motor de simulación Euler basado en nodos y scripts que permite la creación modular de simulaciones físicas con primitivos 2D y 3D. La estructura del sistema facilita la conexión e interacción entre distintos componentes, lo que proporciona un entorno versátil y extensible para la construcción de escenarios dinámicos y el análisis de fenómenos físicos. Esta modularidad, junto con la integración de un lenguaje de *scripting*, ofrece un equilibrio adecuado entre accesibilidad y flexibilidad, permitiendo tanto el uso didáctico como la experimentación avanzada.

La herramienta permite visualizar, guardar y cargar escenas de forma intuitiva, garantizando la reproducibilidad de las simulaciones y facilitando su adaptación a distintos propósitos. Su diseño se orienta a fomentar la comprensión de conceptos fundamentales en física computacional, integrando la parte teórica y práctica mediante la representación visual de los procesos numéricos. Esto la convierte en una plataforma útil para la enseñanza y el aprendizaje, así como para la exploración de nuevos métodos o configuraciones experimentales.

El sistema de nodos y el sistema de *scripting* están integrados y entrelazados de tal forma que son complementarios y permiten la comunicación entre sí, lo cual resulta en una herramienta poderosa que el usuario puede aprender y utilizar para sus propios proyectos y estudio.

Durante el desarrollo, se comprobó la estabilidad y el correcto funcionamiento del sistema mediante una serie de pruebas y scripts de ejemplo, los cuales evidenciaron la capacidad del motor para adaptarse a diversas situaciones y configuraciones. Los resultados obtenidos muestran que la herramienta responde adecuadamente a las exigencias de flexibilidad, usabilidad y coherencia numérica, ofreciendo una base sólida para futuras ampliaciones.

En síntesis, el motor de simulación desarrollado constituye una solución integral para la creación de entornos físicos virtuales, combinando simplicidad en su uso con la posibilidad de profundizar en la modelación y el análisis computacional. Su estructura modular y extensible sienta las bases para futuras mejoras, como la incorporación de nuevos métodos

de integración, sistemas de partículas más complejos o la interacción en tiempo real con motores gráficos externos.

- [1] W. E. Boyce y R. C. DiPrima, *Elementary Differential Equations and Boundary Value Problems*, 10th. John Wiley & Sons, 2012.
- [2] A. Martínez. “Simulator,” visitado 16 de oct. de 2025. dirección: <https://github.com/Raylight-Developer/Simulator>
- [3] R. L. Burden y J. D. Faires, *Numerical Analysis*, 9th. Brooks/Cole, 2010.
- [4] “NVIDIA Omniverse Brings Pixar USD and AI to Scientific Computing,” visitado 16 de oct. de 2025. dirección: <https://nvidianews.nvidia.com/news/nvidia-omniverse-scientific-computing>
- [5] Visitado 16 de oct. de 2025. dirección: <https://x-engineer.org/euler-integration/>
- [6] T. H. Cormen, C. E. Leiserson, R. L. Rivest y C. Stein, *Introduction to Algorithms*, 3rd. Cambridge, MA: MIT Press, 2009, cap. 3, págs. 47-91.
- [7] B. García-Archilla, J. M. Sanz-Serna y R. D. Skeel, “Long-time-step methods for oscillatory differential equations,” *SIAM Journal on Scientific Computing*, vol. 20, n.º 3, págs. 930-963, 1999.
- [8] R. J. LeVeque, *Finite Difference Methods for Ordinary and Partial Differential Equations*. SIAM, 2007.
- [9] H. Samet, *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.
- [10] J. A. Sokolowski y C. M. Banks, “Real-time versus non-real-time modeling and simulation: Implications for analysis and training,” en *Proceedings of the Winter Simulation Conference*, 2010, págs. 2687-2698.
- [11] P. J. Schneider y D. H. Eberly, *Geometric Tools for Computer Graphics*. Morgan Kaufmann, 2002.
- [12] Visitado 16 de oct. de 2025. dirección: <https://jaced.com/wp/2007/01/04/triangle-square-circle-a-psychological-test/>
- [13] J. E. Bresenham, “Algorithm for Computer Control of a Digital Plotter,” *IBM Systems Journal*, vol. 4, n.º 1, págs. 25-30, 1965. DOI: 10.1147/sj.41.0025

- [14] J. D. Foley, A. van Dam, S. K. Feiner y J. F. Hughes, *Computer Graphics: Principles and Practice*, 2nd. Addison-Wesley, 1996.
- [15] D. H. Eberly, *Triangulation by Ear Clipping*. Geometric Tools, 2008. visitado 16 de oct. de 2025. dirección: <https://www.geometrictools.com/Documentation/TriangulationByEarClipping.pdf>
- [16] E. Angel y D. Shreiner, *Interactive Computer Graphics: A Top-Down Approach with Shader-Based OpenGL*, 6th. Addison-Wesley, 2012.
- [17] Visitado 16 de. oct. de 2025. dirección: https://help.altair.com/inspire/en_us/topics/implicit/primitive_t.htm
- [18] P. Shirley, M. Ashikhmin y S. Marschner, *Fundamentals of Computer Graphics*, 3rd. A K Peters, 2009.
- [19] K. Suffern, *Ray Tracing from the Ground Up*. A K Peters, 2007.
- [20] S. Marschner y P. Shirley, *Fundamentals of Computer Graphics*, 4th. CRC Press, 2015.
- [21] T. Akenine-Möller, E. Haines y N. Hoffman, *Real-Time Rendering*, 4th. CRC Press, 2018.
- [22] J. F. Hughes et al., *Computer Graphics: Principles and Practice*, 3rd. Addison-Wesley, 2013.
- [23] M. Botsch, L. Kobbelt, M. Pauly, P. Alliez y B. Lévy, *Polygon Mesh Processing*. CRC Press, 2010.
- [24] Visitado 16 de. oct. de 2025. dirección: <https://ar.inspiredpencil.com/pictures-2023/3d-topology-head>
- [25] T. Akenine-Möller, E. Haines y N. Hoffman, *Real-Time Rendering*, 4th. CRC Press, 2018.
- [26] Visitado 16 de. oct. de 2025. dirección: <https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation/rasterization-stage.html>
- [27] J. P. Morrison, *Flow-Based Programming: A New Approach to Application Development*, 2nd. CreateSpace, 2010.
- [28] B. Stroustrup, *The C++ Programming Language*, 4th. Addison-Wesley, 2013.
- [29] *OpenGL Mathematics (GLM)*, <https://github.com/g-truc/glm>, 2025.
- [30] *Qt 6 Framework*, <https://www.qt.io/product/qt6>, 2025.
- [31] kishimisu, visitado 16 de oct. de 2025. dirección: <https://www.shadertoy.com/view/mtYGWY>
- [32] culdevu, visitado 16 de oct. de 2025. dirección: <https://www.shadertoy.com/view/4s2cWK>