311158894

Grupo Teoría: 05

Grupo Laboratorio: 13

Proyecto Final

Casa con 2 habitaciones

Objetivo

Desarrollar un recorrido virtual interactivo utilizando OpenGL que simule de manera realista la exploración de una estructura arquitectónica compuesta por dos cuartos y una fachada, basada en referencias visuales reales o ficticias previamente seleccionadas (excluyendo temáticas específicas indicadas en los lineamientos). El proyecto integrará una cámara sintética para la navegación, y un diseño artístico coherente con las imágenes de referencia seleccionadas. Además, se generará una documentación completa que incluya un diagrama de Gantt, un manual de usuario bilingüe (español o inglés) con los objetivos y la descripción de la interacción, un manual técnico, y un análisis de costos con justificación del precio estimado de venta. El resultado final incluirá un archivo ejecutable funcional, un repositorio en GitHub bien estructurado y sin archivos comprimidos ni superiores a 100 MB, cumpliendo con criterios de realismo, presentación y organización.

Alcance

Este proyecto está fundamentado en el conocimiento adquirido a lo largo del semestre y utiliza tecnologías como GLEW, GLFW y OpenGL. La implementación se basará en las imágenes de referencia (incluidas más abajo en este documento), las cuales proporcionan una base clara para replicar los espacios de la manera más precisa posible, considerando las limitaciones propias del nivel de conocimiento alcanzado hasta ahora. Este trabajo se ha concebido como una comparación directa con sistemas avanzados como Lumion, un motor de renderizado de última generación ampliamente utilizado en firmas de arquitectura.

El proyecto cumplirá con todas las características establecidas en el objetivo general y es importante subrayar que será completamente desarrollado mediante programación directa en OpenGL. Se llevará a cabo en el entorno de Visual Studio, donde se implementarán los diferentes *shaders* y la lógica necesaria para la interacción y renderizado del recorrido virtual.

Un aspecto adicional relevante es el uso de SketchUp como herramienta de apoyo para el modelado, dado que está orientado principalmente al diseño arquitectónico, lo cual facilitará la creación precisa de los espacios requeridos.

Durante el desarrollo, todo el avance del proyecto se documentará y subirá al repositorio general del semestre, donde también estarán disponibles las prácticas de laboratorio. No obstante, el producto final que se entregará incluirá únicamente los archivos necesarios, debidamente organizados y documentados, tal como se solicita.

Finalmente, dado que las imágenes de referencia contienen una cantidad considerable de objetos, se incluirán algunos elementos adicionales no requeridos explícitamente por el profesor. Esto con el objetivo de alcanzar un mayor grado de fidelidad visual, calidad técnica y presentación general en el producto final.

Cronograma de actividades

El primer paso de este proyecto consistió en seleccionar un modelo base adecuado para trabajar. Para ello, se exploraron diversas opciones y se eligió uno que presentó un grado de complejidad suficientemente alto, en concordancia con los objetivos establecidos.

A continuación, se verificó la viabilidad de integrar SketchUp con OpenGL de manera fluida. Este paso fue crucial, ya que fue necesario comprobar que los objetos tridimensionales pudieran exportarse correctamente desde SketchUp y ser leídos sin errores en el entorno de desarrollo. En caso de que esta integración no funcionara adecuadamente, el resto del proyecto no habría sido viable con las herramientas propuestas.

Superada esta etapa, se procedió al modelado del entorno en SketchUp. Durante esta fase, se realizaron pruebas constantes para asegurar que los modelos se cargaban correctamente en OpenGL mediante Visual Studio. Una vez finalizado el modelo tridimensional, se aplicaron texturas basadas en las imágenes de referencia que se habían proporcionado en la etapa de planeación.

Con el entorno modelado y texturizado, se incorporaron los objetos detallados en la lista presentada al profesor. Cada uno fue verificado individualmente para garantizar su correcta carga y visualización en el entorno de desarrollo.

Finalmente, se desarrolló la documentación técnica del proyecto, la cual proporcionó el sustento profesional requerido. Esta incluyó los objetivos del proyecto, su alcance, cronograma (diagrama de Gantt), imágenes de referencia, análisis de costos, manual técnico, manual de usuario y los demás elementos solicitados.

Diagrama de Gantt

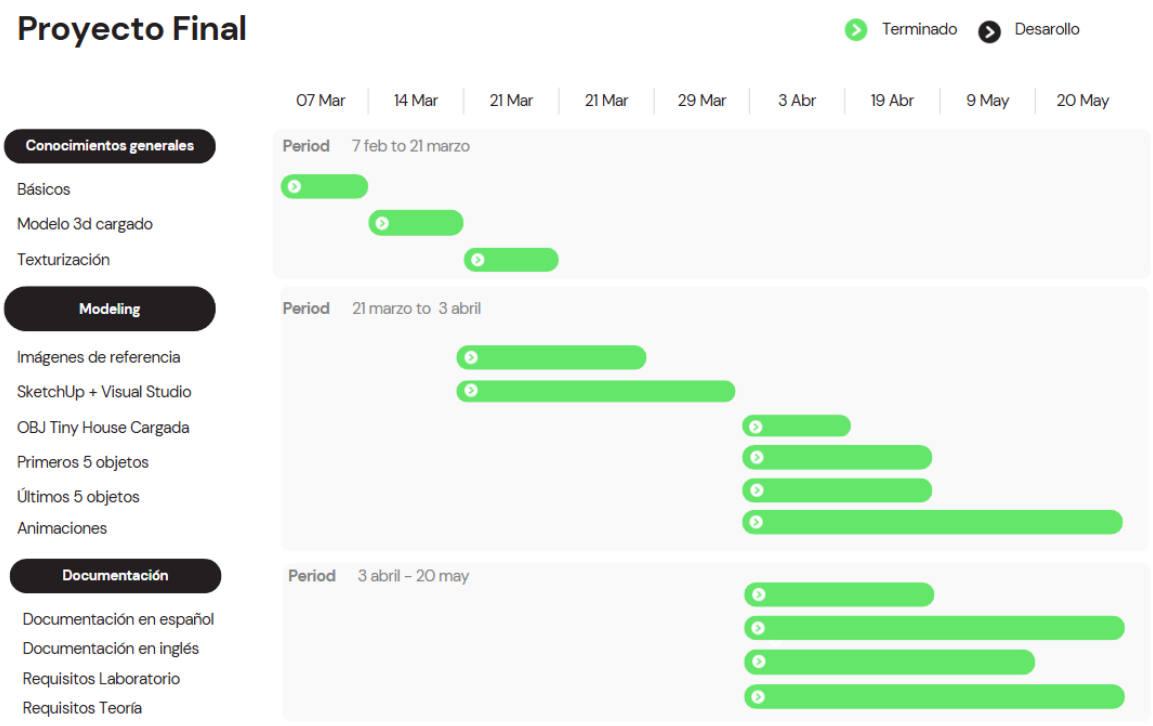

Diagrama de Gantt.

Limitantes

En este proyecto se presentaron diversos retos que superaron el conocimiento previamente adquirid. Uno de los principales desafíos fue el dominio de técnicas avanzadas de renderizado, como iluminación. Se enfrentaron varias dificultades al intentar implementar los distintos elementos de iluminación, ya que esta atravesaba los objetos en lugar de proyectarse y reflejarse correctamente sobre sus superficies.

Por otro lado, el texturizado no logró ofrecer la fidelidad visual alcanzada por programas como Lumión, los cuales integran mapas normales para simular con mayor realismo el comportamiento de la luz. Como resultado, se generó un efecto visual de calidad considerablemente menor.

Además, los objetos utilizados contaban con una resolución geométrica muy baja, es decir, una cantidad reducida de vértices, aristas y triángulos. Por ello, se optó por alternativas más viables que permitieran representar, en la medida de lo posible, los elementos propios de una escena arquitectónica con los recursos disponibles.

Finalmente, otra limitante importante fue la ambientación del entorno. Resultó complejo simular adecuadamente elementos como árboles, cielo y pasto con volumen, lo que redujo el realismo general del escenario.

Manual Técnico

Este manual técnico documenta el desarrollo de un proyecto de computación gráfica cuyo objetivo es renderizar una escena tridimensional interactiva utilizando OpenGL. El programa carga modelos 3D en formato OBJ, implementa una cámara en primera persona, y simula distintos tipos de iluminación, incluyendo luces direccionales, puntuales y tipo linterna (spotlight).
La aplicación ha sido desarrollada en C++ con el uso de bibliotecas especializadas como GLEW para el manejo de extensiones de OpenGL, GLFW para la creación de ventanas y captura de eventos, GLM para operaciones matemáticas, y stb_image o SOIL2 para el manejo de texturas. Además, se han desarrollado clases personalizadas como Shader, Camera y Model para facilitar la arquitectura modular del sistema.
Este documento describe la estructura del código, la configuración de luces y texturas, el proceso de carga y renderizado de modelos, así como los procedimientos para compilar y ejecutar el proyecto correctamente.

Metodología de Desarrollo Aplicada

Para el desarrollo de este proyecto se adoptó un enfoque estructurado de programación modular, orientado a objetos. Aunque no se implementó una metodología formal de ingeniería de software como Scrum o Cascada, sí se siguieron principios fundamentales de buenas prácticas en programación gráfica:

- Modularidad: Separación clara del código en clases específicas: Shader para gestión de shaders, Camera para navegación en la escena, y Model para carga y renderizado de modelos.
- Iteración incremental: El proyecto se desarrolló en etapas funcionales: configuración básica de OpenGL, implementación de cámara, carga de modelos, y finalmente iluminación.
- Reutilización de código: Se utilizaron librerías reutilizables de terceros (GLFW, GLEW, stb_image, GLM, SOIL2) que permitieron enfocar el desarrollo en la lógica del proyecto.
- Pruebas constantes: A lo largo del desarrollo se hicieron ejecuciones frecuentes para verificar la correcta visualización de modelos, luces y movimiento de cámara.

Este enfoque permitió mantener un código legible, ordenado y fácil de depurar durante todas las fases del desarrollo.

Requisitos del Sistema y Configuración en Visual Studio

Para compilar y ejecutar correctamente este proyecto en Visual Studio, se deben cumplir los siguientes requisitos:

Requisitos del sistema:

- Sistema operativo: Windows 10 o superior
- Memoria RAM: mínimo 4 GB (recomendado 8 GB o más)
- Tarjeta gráfica compatible con OpenGL 3.3 o superior
- Espacio libre en disco: al menos 200 MB para recursos, librerías y modelos

Configuración en Visual Studio:

1. Versión recomendada: Visual Studio 2019 o 2022 con el paquete de desarrollo de escritorio con C++ instalado.
2. Crear un proyecto nuevo:
    - Proyecto de tipo: "Aplicación de consola en C++".
3. Agregar los archivos del proyecto: main.cpp, Shader.h/.cpp, Camera.h/.cpp, Model.h/.cpp, y las carpetas de modelos y shaders.
4. Configurar rutas adicionales:
    - En propiedades del proyecto → C/C++ → General → Directorios de inclusión adicionales:
        - Ruta a GLEW (incluye carpeta /include)
        - Ruta a GLFW, GLM, stb_image, y SOIL2 (si aplica)
    - En propiedades del proyecto → Vinculador → General → Directorios de bibliotecas adicionales:
        - Ruta a las bibliotecas (/lib) de GLEW, GLFW, etc.
5. Agregar dependencias al vinculador:
    - En propiedades del proyecto → Vinculador → Entrada → Dependencias adicionales:
    - opengl32.lib
    - glew32.lib
    - glfw3.lib
    - SOIL2.lib
    - assimp

6. Asegúrate de tener las DLL necesarias:
    - Copia archivos como glew32.dll, glfw3.dll, SOIL2.dll. assimp-vc-140mt.dll conforme la estructura que se tiene en el repositorio.

Una vez configurado correctamente, el proyecto debería compilar y ejecutar mostrando una escena 3D interactiva con cámara y modelos cargados.

Estructura del Código y Explicación del Funcionamiento

El proyecto está organizado de forma modular y estructurada para facilitar su comprensión y mantenimiento. A continuación se describen los componentes principales:

Archivos principales:

- main.cpp: Contiene el flujo principal del programa, la configuración de la ventana, el bucle de renderizado y la lógica de entrada del usuario.
- Shader.h/.cpp: Implementa una clase para compilar, enlazar y usar shaders (GLSL).
- Camera.h/.cpp: Controla el movimiento y orientación de la cámara mediante teclado y ratón.
- Model.h/.cpp: Maneja la carga de modelos 3D en formato .obj y su renderizado.

Esta organización asegura un funcionamiento claro y modular del motor gráfico desarrollado, facilitando la ampliación con nuevas funcionalidades como animaciones o físicas.

Documentación del Código

A continuación, se presenta la documentación detallada de las secciones clave del código fuente del proyecto:

main.cpp

- Bibliotecas utilizadas: Incluye librerías estándar (iostream, cmath), gráficas (GLEW, GLFW), de utilidades de imagen (stb_image, SOIL2), matemáticas (GLM) y clases personalizadas (Shader, Camera, Model).

- Variables globales: Se declaran constantes para el tamaño de la ventana, estados de teclado, configuración de la cámara y control del tiempo entre cuadros (deltaTime).
- Luz y color: Se definen estructuras para manejar la posición de luces puntuales y efectos visuales como cambios de color dinámico con funciones trigonométricas.

Funciones clave

- KeyCallback(GLFWwindow*, int, int, int, int): Maneja entradas de teclado y permite mover luces o activar efectos.
- MouseCallback(GLFWwindow*, double, double): Controla la rotación de la cámara con el ratón.
- DoMovement(): Actualiza la posición de la cámara según teclas presionadas.
- Animación(): Actualiza las variables que controlan las animaciones basándose en el tiempo transcurrido.

Bucle principal (while)

- Limpia buffers (glClear).
- Calcula tiempo entre frames.
- Aplica transformaciones (view, projection, model).
- Actualiza las propiedades de cada tipo de luz y pasa los datos a los shaders.
- Dibuja modelos cargados en la escena: Ground, Tiny, Crystal.
- Renderiza un cubo que representa la luz con lampShader.

Shaders y Texturas

- Se utiliza la clase Shader para compilar y vincular shaders personalizados.
- Se asignan las texturas difusas y especulares a través de unidades de textura (GL_TEXTURE0, GL_TEXTURE1).

Modelos

- Se emplea la clase Model para importar objetos .obj y renderizarlos mediante Draw(shader).

Camera.h

Bibliotecas utilizadas

Este archivo incluye bibliotecas estándar de C++ (<vector>), así como librerías específicas de OpenGL y matemáticas gráficas como GLEW y GLM (glm/glm.hpp, glm/gtc/matrix_transform.hpp).

Variables y constantes

- Enumeración Camera_Movement: Define los movimientos posibles de la cámara (adelante, atrás, izquierda, derecha).

- Constantes predeterminadas: Define valores por defecto para el ángulo de giro (YAW), inclinación (PITCH), velocidad de movimiento (SPEED), sensibilidad del mouse (SENSITIVITY) y nivel de zoom (ZOOM).

Clase Camera

Una clase que abstrae la lógica de una cámara en un entorno tridimensional, encargada de gestionar su posición, orientación y proyección. Permite recibir entradas del teclado y del mouse para navegar libremente por la escena.

Atributos principales:

- position, front, up, right, worldUp: Vectores que definen la posición y orientación espacial de la cámara.
- yaw, pitch: Ángulos de rotación de la cámara.
- movementSpeed, mouseSensitivity, zoom: Parámetros de configuración dinámica de la cámara.

Métodos principales:

- Constructores: Permiten inicializar la cámara con vectores o con valores escalares.
- GetViewMatrix(): Devuelve la matriz de vista utilizando la función glm::lookAt.
- ProcessKeyboard(Camera_Movement, GLfloat): Procesa la entrada del teclado para mover la cámara según la dirección y el tiempo.
- ProcessMouseMovement(GLfloat, GLfloat, GLboolean): Actualiza los ángulos yaw y pitch para rotar la cámara con el mouse.
- ProcessMouseScroll(GLfloat): Método preparado para aplicar zoom con la rueda del mouse (actualmente vacío).
- Getters: Métodos como GetZoom(), GetPosition() y GetFront() permiten obtener el estado actual de la cámara.
- updateCameraVectors(): Método privado que recalcula los vectores front, right y up a partir de los ángulos actuales. Es crucial para mantener la orientación visual coherente al rotar la cámara.

Mesh.h

Propósito general

Esta clase encapsula la estructura y renderizado de una malla 3D compuesta por vértices, índices y texturas. Está diseñada para ser usada en conjunto con modelos importados mediante Assimp.

Estructuras principales

- Vertex: Contiene posición (vec3), normal y coordenadas de textura.
- Texture: Guarda el ID de textura, tipo (difusa o especular) y ruta original.

Funcionalidad clave

- Constructor Mesh(): Recibe los datos de vértices, índices y texturas; inicializa buffers mediante setupMesh().
- Draw(Shader): Renderiza la malla vinculando texturas y enviando datos al shader.
- setupMesh() (privado): Configura VAO, VBO y EBO, y define los punteros de atributos del vértice.

Model.h

Propósito general

La clase Model permite importar, procesar y renderizar modelos 3D completos desde archivos .obj mediante la biblioteca Assimp. Automatiza el manejo de múltiples mallas (Mesh), texturas y estructuras jerárquicas.

Funcionalidad clave

- Constructor Model(path): Carga el modelo desde archivo y procesa todas sus mallas recursivamente.
- Draw(Shader): Dibuja cada malla cargada usando el shader proporcionado.

Procesos internos

- loadModel(): Utiliza Assimp para leer el archivo, extraer nodos y recorrerlos recursivamente.
- processNode(): Recorre los nodos del modelo, extrae cada malla (aiMesh) y la transforma en una instancia de Mesh.
- processMesh(): Traduce vértices, normales, coordenadas UV e índices desde el formato de Assimp al formato usado por OpenGL.
- loadMaterialTextures(): Evita cargar texturas duplicadas, asigna texturas difusas y especulares por malla.

Carga de texturas externas

- TextureFromFile(): Función auxiliar que carga imágenes desde disco usando SOIL2 y genera los GLuint de textura en OpenGL.

Shader.h

Propósito general

La clase Shader encapsula la lógica de carga, compilación, depuración y uso de shaders de vértices y fragmentos en OpenGL. Proporciona una interfaz sencilla para trabajar con programas de shaders durante el renderizado.

Funcionalidad clave

- Constructor Shader(vertexPath, fragmentPath): Carga archivos GLSL desde disco, compila los shaders, crea el programa de shader y enlaza ambos módulos. También gestiona y muestra errores de compilación y enlace si ocurren.

- Use():
  Activa el shader actual para ser usado en las operaciones de dibujo (glUseProgram).
- getColorLocation():
  Retorna la ubicación de la variable uniforme "color" en el shader, útil para manipular colores desde C++.

Variables internas

- Program: Identificador del programa de shader creado y enlazado.
- uniformColor: Ubicación de la variable uniforme color dentro del shader fragmento (si se usa).

---

core.frag

Propósito general

Este shader de fragmento define el color final de cada píxel utilizando una entrada de color interpolada (ourColor) proveniente del vertex shader o de una variable uniforme.

Funcionalidad

- Entrada (in vec3 ourColor): Recibe el color del vértice interpolado por la rasterización.
- Salida (out vec4 color): Define el color final del fragmento que se mostrará en pantalla, incluyendo un canal alfa fijo de 1.0 (opacidad total).
- main(): Asigna directamente el valor de ourColor con opacidad total al fragmento.

---

core.vs (Vertex Shader)

Propósito general

Este shader de vértices transforma la posición de cada vértice de modelo a coordenadas de clip utilizando las matrices model, view y projection, y transmite un color uniforme hacia el fragment shader.

Funcionalidad

- Entrada (layout(location = 0) in vec3 position): Recibe la posición de cada vértice desde el VBO.
- Uniformes:
  - mat4 model: Transformación del objeto local al mundo.
  - mat4 view: Transformación de mundo a vista (cámara).
  - mat4 projection: Proyección perspectiva u ortográfica.
  - vec3 color: Color uniforme que se usará por vértice.
- Salida (out vec3 ourColor): Pasa el color al fragment shader.

Función main():

- Calcula gl_Position como el producto de las matrices de transformación aplicadas a la posición del vértice.
- Asigna el color uniforme a la variable ourColor, que se interpolará por fragmento.

---

lighting.frag

Este shader aplica iluminación Phong combinando luz direccional, cuatro luces puntuales y una luz tipo spotlight. Usa texturas difusas y especulares, incluye atenuación por distancia y permite descartar fragmentos transparentes si se activa la opción transparency. La iluminación se calcula mediante funciones auxiliares según el tipo de luz y se combina en el color final del fragmento.

---

lighting.vs

Este shader transforma la posición de cada vértice al espacio de clip (gl_Position) usando las matrices model, view y projection. Calcula la posición del fragmento en el mundo (FragPos), la normal transformada correctamente (Normal) y transmite las coordenadas de textura (TexCoords) al fragment shader para iluminación y texturizado.

Bibliotecas y librerías Utilizadas:

```
#include <iostream>         // Entrada/salida en consola
#include <cmath>            // Funciones matemáticas estándar
#include <GL/glew.h>        // Manejo de extensiones modernas de OpenGL
#include <GLFW/glfw3.h>     // Ventanas y entrada de usuario
#include "stb_image.h"      // Carga de texturas
#include <glm/glm.hpp>      // Tipos matemáticos (vec3, mat4, etc.)
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
#include "SOIL2/SOIL2.h"    // Carga alternativa de texturas
#include "Shader.h"         // Clase personalizada para manejar shaders
#include "Camera.h"         // Clase para controlar la cámara
#include "Model.h"          // Clase para cargar y dibujar modelos .obj
```

Manual de Usuario

Para usar este programa es muy sencillo. Únicamente necesita un teclado y un mouse. Las siguientes teclas le permiten desplazarse por el ambiente tridimensional:

W: Moverse hacia adelante

S: Moverse hacia atrás

A: Moverse hacia la izquierda

D: Moverse hacia la derecha

1: Abrir puerta principal

2: Cerrar ventana que está de frente a la entrada

3: Abrir puerta del microondas

4: Mover la casa rodante completa (simulando avanzar)

Además, puede usar el **mouse** para rotar la cámara y observar el entorno en todas direcciones.

La experiencia de navegación está diseñada en primera persona, similar a un videojuego 3D. Se recomienda utilizar una resolución de pantalla mínima de 1920x1080 para una visualización óptima.

Diagrama de Flujo de Codigo

Diagrama de Flujo

Imágenes de Referencia y Resultados

Figura 1. Exterior.

Figura 2. Interior con vistas a la cocina y escaleras.

Figura 3. Vista superior.

Figura 4. Sala

Figura 5. Habitación.

Figura 6. Habitación vista alternativa.

Análisis de Costos

Concepto	**Detalle**	**Costo Aproximado (MXN)**
Computadora	Uso proporcional de equipo propio valuado en $15,000 (20% para este proyecto)	$3,000
Licencia Sketchup	Prueba gratuita sin fines de lucro	$0
Texturas	Descargadas gratuitamente desde la web	$0
Mano de obra total (mes y medio)	100 horas estimadas * $100/hora	$10,000
Aprendizaje de animaciones en OpenGL y uso de Sketchup	5 horas autodidacta * $100/hora	$500
Documentación del proyecto	Elaboración de manual, edición de imágenes y PDF final	$200
Total estimado		**$13,700**

Conclusiones

El proyecto logró cumplir con todos los objetivos establecidos, consolidándose como una demostración integral de los conocimientos adquiridos en el curso de Computación Gráfica. A través del uso de OpenGL y su ecosistema de herramientas, se implementaron sistemas clave que permiten la creación de una experiencia interactiva tridimensional: desde la importación y manipulación de modelos 3D hasta la simulación de diferentes tipos de iluminación y una cámara en primera persona completamente funcional.

La generación exitosa de un archivo ejecutable representa no solo el cierre técnico del proyecto, sino también su transformación en una aplicación autónoma y accesible para el usuario final. Este logro no habría sido posible sin una profunda comprensión de las dependencias del sistema, el manejo de bibliotecas y la configuración cuidadosa del entorno de desarrollo.

Durante el desarrollo surgieron diversos desafíos, especialmente debido a la suspensión académica de un mes por motivos de paro, lo que obligó a replantear tiempos, prioridades y estrategias de trabajo. A pesar de ello, se logró mantener el rumbo gracias a una planificación detallada reflejada en el diagrama de Gantt, que sirvió como brújula organizativa para seguir avanzando en los distintos frentes del proyecto: modelado, texturizado, renderizado y navegación.

Este recorrido virtual no sólo es una simulación técnica: es una puerta de entrada a mundos más complejos. Se cimentaron las bases para comprender tecnologías de siguiente nivel como Vulkan o técnicas de trazado de rayos (raytracing), las cuales requieren una familiaridad profunda con conceptos como buffers, shaders y pipelines de renderizado. De este modo, el proyecto no sólo cumplió su función pedagógica, sino que despertó una vocación más amplia por la exploración técnica y artística del desarrollo tridimensional.

Finalmente, esta experiencia también dejó lecciones sobre el valor del trabajo artesanal en programación gráfica. Optar por una construcción desde cero en lugar de utilizar motores comerciales como Unity o Unreal permitió valorar el esfuerzo real detrás de cada línea de código, así como comprender los costos de producción en tiempo y recursos. Este conocimiento es clave para cualquier futuro emprendimiento profesional o académico en el campo de la tecnología visual.

Anexos

Repositorio:
https://github.com/acasanova009/CompuGrafica

SketchUp Gratuito:
https://www.sketchup.com/en/plans-and-pricing/sketchup-free