UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
FACULTAD DE INGENIERÍA
DIVISIÓN DE INGENIERÍA ELÉCTRICA
INGENIERÍA EN COMPUTACIÓN
COMPUTACIÓN GRÁFICA E INTERACCIÓN HUMANO COMPUTADORA

Proyecto 2 Final
Manual Técnico
Computación Gráfica e Interacción Humano Computadora

Alumno. 319079485

Profesor. ING. CARLOS ALDAIR ROMAN BALBUENA

Grupo 5

Semestre 2026-1

Fecha de entrega: 25 de Noviembre 2025

Índice

Introducción

Este proyecto consiste en el desarrollo de una simulación 3D interactiva construida con C++ y OpenGL 3.3, donde se recrea una casa steampunk caminante equipada con un sistema de animaciones jerárquicas, objetos interactivos por proximidad, un dron autónomo y un recorrido cinematográfico reproducible mediante keyframes.

El objetivo principal es demostrar el aprendizaje de técnicas de computación gráfica: modelado 3D, texturizado, iluminación, animación por interpolación, transformaciones jerárquicas, cámara sintética y manejo de recursos en GPU.

Durante la ejecución, el usuario puede:

- Explorar libremente el entorno post-apocalíptico.
- Activar animaciones de la casa mediante teclas.
- Interactuar con objetos que reaccionan a la proximidad.
- Ejecutar un recorrido automático previamente grabado.

La escena incluye un skybox de ambiente desértico, un dron con vuelo autónomo circular, puertas con apertura dependiente de distancia, una antena parabólica giratoria y cuatro patas mecánicas articuladas con movimientos.
La implementación del proyecto involucra conceptos de gráficos por computadora en tres dimensiones.

El pipeline de renderizado comienza con la definición de geometría tridimensional mediante vértices, normales y coordenadas de textura almacenados en buffers de la GPU. Los shaders programables procesan esta información: el vertex shader transforma las coordenadas de los vértices del espacio local al espacio de pantalla mediante matrices de modelo, vista y proyección, mientras que el fragment shader calcula el color final de cada píxel aplicando modelos de iluminación Phong y mapeado de texturas. El sistema de iluminación implementa tres tipos de fuentes luminosas: luz direccional que simula el sol con rayos paralelos, luces puntuales que emiten radialmente desde posiciones específicas con atenuación cuadrática, y una luz focal tipo linterna con ángulo de apertura definido. Cada luz contribuye

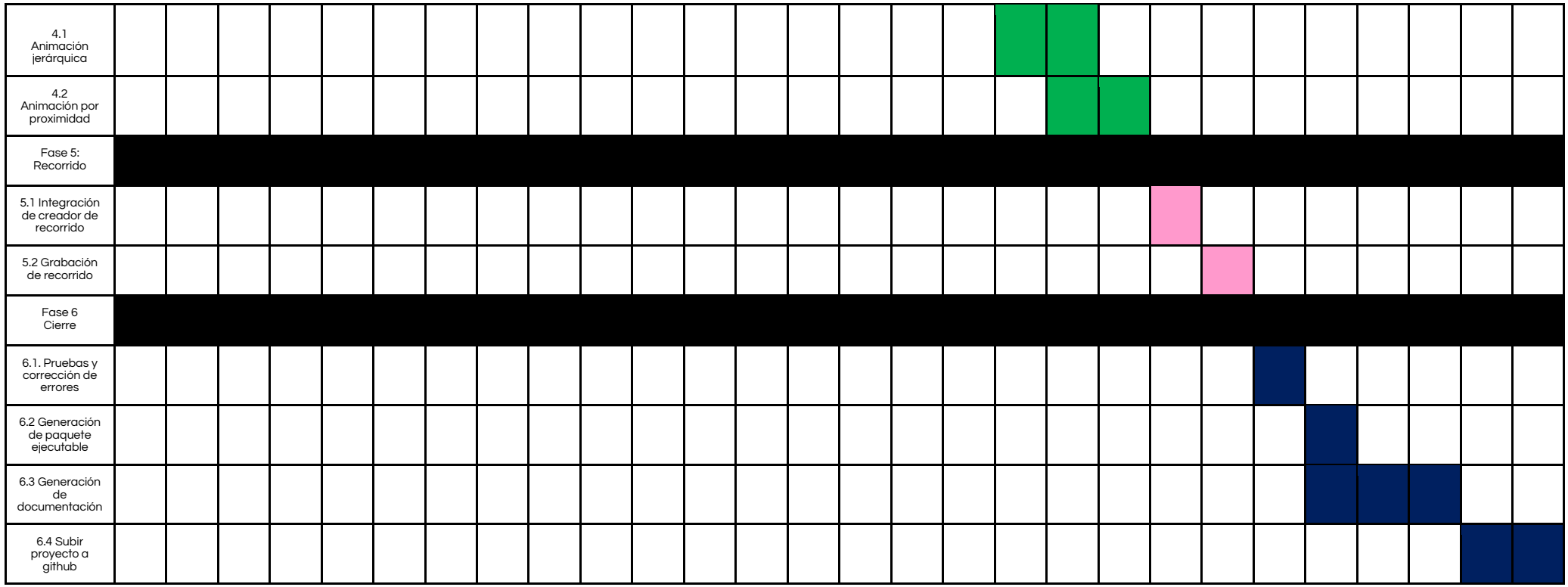
componentes ambientales, difusas y especulares al color final de las superficies según su material y orientación respecto a la fuente.

El proyecto utiliza la biblioteca GLFW para gestión de ventanas y eventos de entrada, GLEW para acceder a extensiones modernas de OpenGL, GLM para operaciones matriciales y vectoriales, SOIL2 para carga de imágenes como texturas, y un parser personalizado para archivos .obj que extrae la geometría de los modelos tridimensionales.

Objetivos

El objetivo principal de este proyecto es desarrollar una escena 3D interactiva en tiempo real utilizando C++ y la API gráfica OpenGL.

1.- Diagrama de Gantt	10/10	15/10	17/10	20/10	21/10	22/10	23/10	24/10	25/10	26/10	27/10	28/10	29/10	30/10	31/10	01/11	02/11	03/11	05/11	06/11	07/11	08/11	09/11	10/11	11/11	12/11	13/11	14/11
Fase 1: Planificación y Setup																												
1.1Eleccion de referencia y objetos																												
1.2. Configuración del Entorno (VS, Git)																												
1.3. Setup de Librerías (GLFW, GLEW, GLM)																												
Fase 2: Entorno Básico																												
2.1. Carga de Shaders (Clases)																												
2.2 Modelado de casa fachada																												
2.3 Modelado interior de la casa																												
2.4 Modelado objetos de la casa																												
2.5 Implantación de texturas																												
Fase 3: Creación de escenario																												
3.1 Implementación cámara sintética																												
3.2 Carga de modelos y texturas																												
3.3 Iluminación Básica																												
3.4 Implementación de skybox																												
Fase 4: Animación																												


4.1 Animación jerárquica
4.2 Animación por proximidad
Fase 5: Recorrido
5.1 Integración de creador de recorrido
5.2 Grabación de recorrido
Fase 6 Cierre
6.1. Pruebas y corrección de errores
6.2 Generación de paquete ejecutable
6.3 Generación de documentación
6.4 Subir proyecto a github

2. Manual Técnico

2.1 Metodología de Desarrollo Implementada

Para este proyecto, utilicé la metodología de prototipos, la cual me permitió construir el entorno de la Casa estilo steampunk de forma iterativa e incremental. Comencé con la carga básica de modelos estáticos y fui refinando cada componente como la jerarquía del dron y el sistema de keyframes de las patas hasta alcanzar el comportamiento fluido que se requería.

Esta metodología resultó útil para este proyecto de gráficos 3D, ya que muchas decisiones visuales (como la intensidad de la iluminación, la escala del skybox o la velocidad de las interpolaciones) y de interacción (como la distancia de activación de la puerta) solo podían evaluarse y ajustarse correctamente al ver el sistema renderizado en tiempo real, e ir modificando por versiones hasta alcanzar una final.

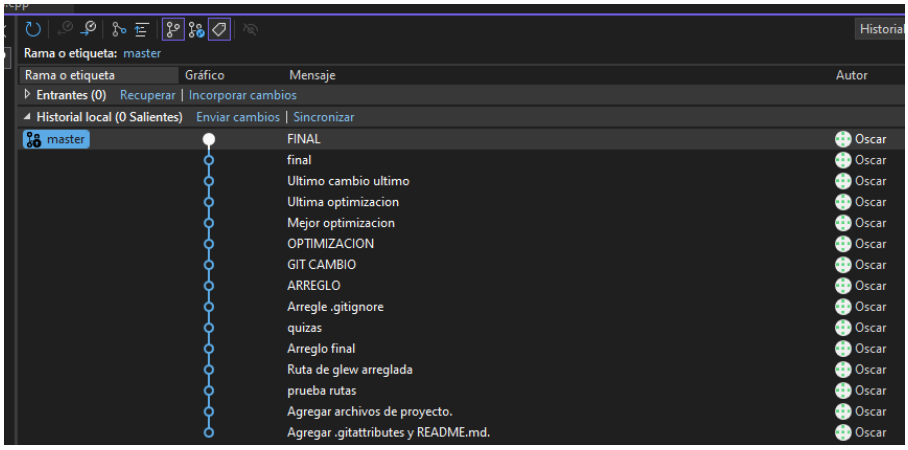
Control de Versiones
Durante todo el proceso de desarrollo utilicé Git y GitHub como sistema de control de versiones dentro de mi interno hasta publicar una versión final. Esto fue fundamental para organizar el flujo de trabajo y gestionar la gran cantidad de archivos y de código involucrados.

Implementé una estrategia de gestión de repositorios que me permitió:

- Controlar los cambios en el código fuente (.cpp, .h) de manera segura.

- Gestionar las dependencias externas (librerías y .dlls) y los recursos pesados (modelos .obj y texturas) mediante una configuración precisa del archivo. gitignore, evitando subir archivos temporales o de compilación innecesarios.

- Mantener versiones estables del proyecto mientras experimentaba con nuevas características, como la implementación del grabador de recorridos de cámara, y si alguna versión no funcionaba, podía regresar en cualquier momento a una rama anterior.



2.2 Justificación de uso de tecnologías

La selección tecnológica para este proyecto responde a la necesidad de un alto rendimiento, control total sobre el pipeline gráfico y compatibilidad estándar en la industria.

Blender (Modelado y Exportación 3D)
Se eligió Blender por ser una aplicacion de creación 3D de código abierto con muchas opciones de modelado y versátil, ya que tiene una curva de aprendizaje demasiado amigable con el usuario. Su capacidad para gestionar mallas complejas, mapeado UV y jerarquías de objetos facilita la creación de activos optimizados para motores en tiempo real.

También, su exportador nativo al formato Wavefront (.obj) genera archivos limpios y legibles que contienen tanto la geometría (vértices, normales) como las coordenadas de textura, lo cual facilita para hacer el parseado por librerías de carga como Assimp dentro de la aplicación y codificación.

C++ (Lenguaje de Programación)
C++ es el estándar en la industria de los gráficos por computadora y el desarrollo de motores de videojuegos debido a su eficiencia y gestión de la memoria.

C++ permite un control de bajo nivel sobre los recursos del hardware, lo que garantiza que el bucle de renderizado (Game Loop) y los cálculos matemáticos complejos (interpolaciones de animación, transformaciones matriciales) se ejecuten muy rapiodo, manteniendo una tasa de cuadros por segundo (FPS) estable.

OpenGL (API Gráfica)
Se seleccionó OpenGL (Open Graphics Library) como la interfaz de programación de aplicaciones para el renderizado. Al ser una especificación multiplataforma e

independiente del hardware, asegura que el proyecto pueda ejecutarse en una amplia gama de dispositivos, como se observo al realizar los ejecutables.

A diferencia de motores de juego comerciales (como Unity o Unreal) que ocultan la complejidad, usar OpenGL puro hace que se puda implementar y comprender desde cero el funcionamiento del pipeline gráfico programable.

2.3 Estado del Arte

Contexto Actual de la Computación Gráfica

En el panorama actual de la computación gráfica en tiempo real (2024-2025), la industria ha transicionado hacia técnicas de renderizado híbrido que combinan la rasterización tradicional con el trazado de rayos (Ray Tracing) en tiempo real, impulsado por hardware dedicado (como las series RTX de NVIDIA). Motores gráficos modernos como Unreal Engine 5 utilizan tecnologías como Nanite (geometría virtualizada) y Lumen (iluminación global dinámica) que abstraen la complejidad del pipeline gráfico.

Este proyecto se sitúa en el dominio de la Programación Gráfica de Bajo Nivel utilizando OpenGL 3.3 Core Profile. Aunque la industria utiliza APIs más modernas como Vulkan o DirectX 12, OpenGL sigue siendo el estándar académico y profesional para comprender los fundamentos del pipeline programable.

A diferencia de usar un motor comercial donde la física y la animación son cajas negras, en este proyecto implementé manualmente los algoritmos matemáticos que constituyen la base de dichos motores:

Animación Jerárquica y Cinemática Directa (Forward Kinematics):

Los motores modernos utilizan Inverse Kinematics (IK) y Skeletal Animation procesada en GPU (Vertex Skinning).

Implementación del Proyecto: Implementé una jerarquía de transformaciones matriciales (padre-hijo) procesada en CPU. Esto muestra el principio fundamental de cómo las matrices de modelo se propagan a través de un grafo de escena (Scene Graph), tal como lo gestiona internamente Unity o Godot antes de enviarlo a la GPU.

Sistema de Interpolación (Keyframing):

Se utilizan curvas de Bézier o Splines Cúbicos para suavizar movimientos complejos, y Animation Blueprints para mezclar estados.

Implementación del Proyecto: Desarrollé la interpolación lineal propio que lee datos crudos desde archivos de texto (.txt). Esto simula el funcionamiento básico de formatos de archivo estándar como .anim o .fbx.

Mientras que el estado del arte busca el fotorrealismo mediante IA (DLSS) y Ray Tracing, este proyecto se enfoca en la eficiencia del Pipeline de Rasterización, demostrando cómo gestionar múltiples draw calls, uniformes de shaders y transformaciones geométricas de manera explícita.

Panorama Actual de los Recorridos Virtuales
En la actualidad, los recorridos virtuales han evolucionado de ser simples colecciones de imágenes estáticas en 360° (panoramas) a convertirse en experiencias tridimensionales completamente inmersivas y renderizadas en tiempo real.

Tendencias Dominantes:

- **Renderizado en Tiempo Real (Real-Time Rendering):** Motores como Unreal Engine 5 y Unity han estandarizado la visualización arquitectónica (ArchViz) y turística, permitiendo al usuario moverse libremente con 6 grados de libertad (6DOF), en lugar de saltar entre puntos fijos.

- **Cinematografía Virtual:** El uso de cámaras virtuales programadas para seguir trayectorias suaves (Splines) es un estándar en la industria del cine y los videojuegos para crear cutscenes (cinemáticas) sin necesidad de renderizar video pre-grabado, ahorrando memoria y permitiendo cambios dinámicos en la escena.

- **Entornos Reactivos:** La tendencia actual no es solo ver, sino interactuar. Los entornos modernos ("Smart Environments") reaccionan a la proximidad del usuario, activando luces, abriendo puertas o desencadenando animaciones mecánicas automáticamente.

Aplicación en la Industria

Los sistemas de navegación y animación que simula este proyecto se utilizan actualmente en:

- **Arquitectura y Bienes Raíces:** Para recorridos de preventa donde el cliente puede caminar por una casa que aún no existe, ver cómo se abren las puertas o cómo cambia la iluminación.

- **Simulación Industrial:** Entrenamiento de operadores de drones o maquinaria pesada mediante simuladores que replican la física y las jerarquías de movimiento de los robots reales.

Fundamentación del Proyecto en el Contexto Actual

Este proyecto, implementa los principios fundamentales que rigen estas tecnologías comerciales, pero construidos desde cero en C++ y OpenGL:

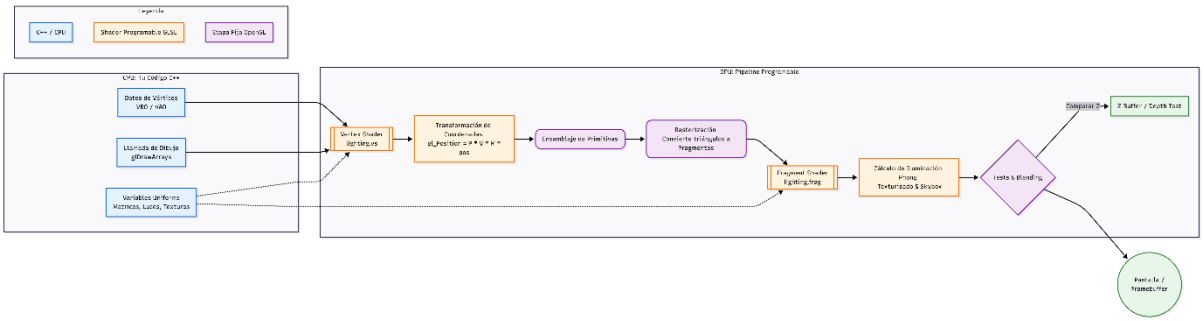
1. **Sistema de Cámara Automatizada (Virtual Cinematography):**
    - En la industria: Se utilizan herramientas como el "Sequencer" de Unreal Engine, que permite colocar keyframes en una línea de tiempo.

    - En este proyecto: Se desarrolló el módulo ReproductorCamara y CameraRecorder, que emula funcionalmente un Sequencer. Al grabar la posición (XYZ) y orientación (Yaw/Pitch) y luego interpolarlas linealmente, recreamos la base matemática de cómo un motor gráfico moderno almacena y reproduce una cinemática.

2. **Animación Procedural y Jerárquica:**
    - En la industria: Se utilizan esqueletos (rigs) complejos para maquinaria y robots.
    - En este proyecto: La implementación del Dron y la Casa Caminante mediante transformaciones matriciales jerárquicas (Padre ----- Hijo) demuestra cómo se construyen objetos complejos articulados sin depender de animaciones pre-renderizadas, permitiendo que el código controle el movimiento en tiempo real (animación algorítmica).

3. **Optimización de Recursos:**
    - Mientras que las tendencias actuales exigen hardware potente (RTX), este proyecto se alinea con la tendencia de computación gráfica eficiente. Al utilizar técnicas clásicas como el modelo de iluminación Phong y gestión manual de memoria, se logra un recorrido virtual fluido que es accesible en hardware de gama media, ideal para moviles

2.4 Diagrama del Pipeline Gráfico (OpenGL)



Descripción del Pipeline Gráfico Implementado

El renderizado de la escena sigue el flujo de trabajo estándar de OpenGL 3.3, procesando la geometría en las siguientes etapas:

1. **Vertex Specification (CPU):** Desde el código C++, se envían los atributos de los vértices (posición, normales, coordenadas de textura) almacenados en los VBOs y organizados por los VAOs de cada modelo (Casa, Dron, Patas).
2. **Vertex Shader (lighting.vs):**

    - Esta etapa programable recibe los vértices crudos.

    - Se aplican las transformaciones matriciales: Modelo (posición del objeto), Vista (cámara) y Proyección (perspectiva), calculando la posición final del vértice en pantalla (gl_Position).

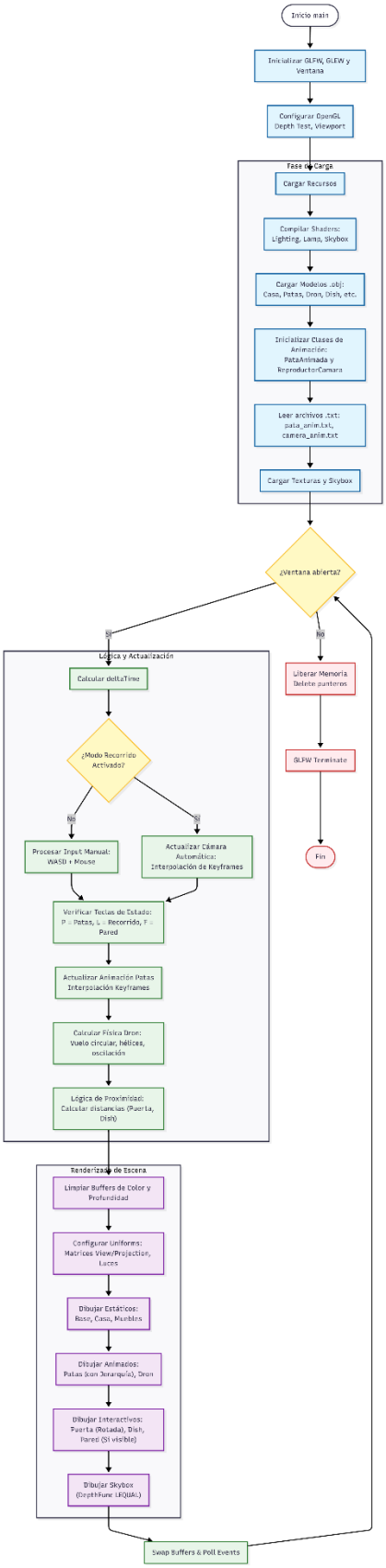
3. **Rasterización (Etapa Fija):** El hardware gráfico toma los triángulos procesados e interpola los datos para generar "fragmentos" (potenciales píxeles) que cubrirán la pantalla.

4. **Fragment Shader (lighting.frag):**

- Esta etapa determina el color final de cada píxel.
- Se implementa el Modelo de Iluminación de Phong, calculando la interacción entre las normales de la superficie y las fuentes de luz (direccional, puntual y spotlight).
- Se realiza el muestreo de texturas (Diffuse y Specular maps) cargadas previamente con SOIL2.

5. **Per-Sample Operations:** Se ejecuta el Depth Test (Prueba de Profundidad) comparando con el Z-Buffer para asegurar que los objetos cercanos oculten correctamente a los lejanos (ej. que las patas no se dibujen encima de la casa si están detrás).

Flujo de la Aplicación (Game Loop)



Fase de Carga (Azul): Ocurre solo una vez. Uso de Assimp para cargar los modelos y clase PataAnimada para leer los .txt.

Decisión de Input (Amarillo): Aquí se muestra la lógica del codigo. Si el usuario activó el recorrido (L), el programa ignora el mouse/teclado y usa los datos del CameraRecorder.

Actualización (Verde): Muestra cómo se calculan las posiciones nuevas.

Renderizado (Morado): Es el orden en que OpenGL dibuja las cosas. Skybox se dibuja al final para optimización.

La arquitectura del software sigue el patrón de diseño estándar de aplicaciones gráficas en tiempo real, conocido como Game Loop (Bucle de Juego). Este ciclo es infinito y se ejecuta miles de veces por segundo hasta que el usuario decide cerrar la aplicación. El flujo se divide en tres etapas:

1. Fase de Inicialización (Setup)
Antes de entrar al bucle infinito, el programa prepara el entorno y carga todos los recursos necesarios en la memoria RAM y VRAM (Video RAM). Esta etapa se ejecuta una sola vez al inicio.

Contexto OpenGL y Ventana: Se inicializa la librería GLFW para crear una ventana del sistema operativo y un contexto de OpenGL válido. Se configura GLEW para enlazar las funciones modernas de la tarjeta gráfica.

Configuración Global: Se habilitan capacidades críticas del motor, como el Z-Buffer

(glEnable(GL_DEPTH_TEST)) para que los objetos 3D se ocluyan correctamente según su profundidad.

Compilación de Shaders: Se leen, compilan y enlazan los archivos GLSL (.vs y .frag) para crear los programas de sombreado (Shader lightingShader, Shader skyboxShader).

Carga de Recursos (Assets):

Modelos: La clase Model utiliza la librería Assimp para leer archivos .obj. Se procesan los vértices, normales y coordenadas de textura, almacenándolos en VAOs (Vertex Array Objects) y VBOs (Vertex Buffer Objects) en la GPU.

Texturas: Se cargan las imágenes difusas y especulares, así como las 6 texturas del mapa cúbico (Cubemap) para el Skybox.

Animaciones: Las clases PataAnimada y ReproductorCamara abren los archivos de texto (.txt), pasan la información de los keyframes y la almacenan en estructuras de datos en memoria dinámica.

2. El Bucle Principal (Main Loop)
(while (!glfwWindowShouldClose(window))). En cada iteración (frame), se realizan secuencialmente las siguientes operaciones:

A. Cálculo de Tiempo (DeltaTime): Se calcula el tiempo transcurrido entre el fotograma actual y el anterior (currentFrame - lastFrame). Esta variable deltaTime es importante para multiplicar todos los movimientos y transformaciones, garantizando que la velocidad de las animaciones sea constante e independiente de los FPS (cuadros por segundo) de la computadora.

B. Procesamiento de Entrada (Input): Se detectan los eventos de periféricos.

La función DoMovement() verifica el estado del teclado.

Máquina de Estados de Control: Si la variable play del ReproductorCamara es true (Modo Recorrido), se ignora el input de movimiento manual (WASD). Si es false, se permite al usuario controlar la cámara libremente.

C. Lógica y Actualización (Update): Aquí se calculan las nuevas posiciones y estados de los objetos antes de dibujar nada.

Interpolación de Keyframes: Las instancias de PataAnimada calculan la posición intermedia entre dos fotogramas clave basándose en el tiempo transcurrido.

Cinemática del Dron: Se resuelven las ecuaciones paramétricas para el vuelo circular (sin/cos) y se actualizan las matrices de transformación de sus componentes hijos (hélices y patas) para que sigan al cuerpo principal.

Lógica de Proximidad: Se calcula la distancia vectorial entre la cámara y objetos interactivos (Puerta, Dish). Si la distancia es menor al umbral (Threshold), se actualizan sus variables de rotación hacia el estado activo.

D. Renderizado (Draw): Se envían las instrucciones a la GPU para pintar la pantalla.

Limpieza: Se borran los buffers de color y profundidad (glClear) para eliminar el rastro del frame anterior.

Configuración de Shaders: Se envían a la GPU las matrices actualizadas (View, Projection) y las posiciones de las luces como variables uniform.

Dibujado de Modelos: Se invoca el método Draw() para cada objeto.

Transformaciones Jerárquicas: Para objetos articulados (como las patas), se aplica una matriz de modelo base y luego matrices de rotación locales para cada articulación antes de dibujar la malla.

Renderizado del Skybox: Se cambia la función de profundidad a GL_LEQUAL y se dibuja el cubo del entorno al final para optimizar el rendimiento

E. Intercambio de Buffers (Swap Buffers): OpenGL utiliza un sistema de Doble Buffer. Mientras se muestra una imagen en pantalla (Front Buffer), el programa dibuja la siguiente en segundo plano (Back Buffer). Al final del ciclo, se intercambian para mostrar la nueva imagen instantáneamente, evitando parpadeos (flickering).

3. Finalización (Cleanup)
Cuando el usuario cierra la ventana, el bucle termina. El programa libera la memoria asignada dinámicamente (punteros new) y termina los procesos de GLFW para devolver el control al sistema operativo limpiamente.

2.5 Arquitectura General del Software

El proyecto está contenido en un único archivo fuente principal (Maquina de estados.cpp) que gestiona todos los aspectos del programa. La arquitectura puede entenderse en estas capas.

1.-Inicialización y gestión de recursos: creación de ventana, contexto OpenGL, carga de shaders y modelos (main() y secciones de carga).

2.-Subsistema de renderizado: shaders, envío de uniforms, VAO/VBO, skybox, draw calls por Model->Draw().

3.-Subsistema de entrada y cámara: callbacks (KeyCallback, MouseCallback), DoMovement() y la clase Camera que mantiene Position, Yaw, Pitch y updateCameraVectors().

4.-Subsistema de animación:

- animaciones por keyframes (PataAnimada, ReproductorCamara)
- animaciones algorítmicas (dron, hélices, oscilaciones)
- animación por proximidad (puerta, dish).

5.-Game loop: sincroniza tiempo (deltaTime), procesa entradas, actualiza animaciones, configura luces y realiza render por frame.

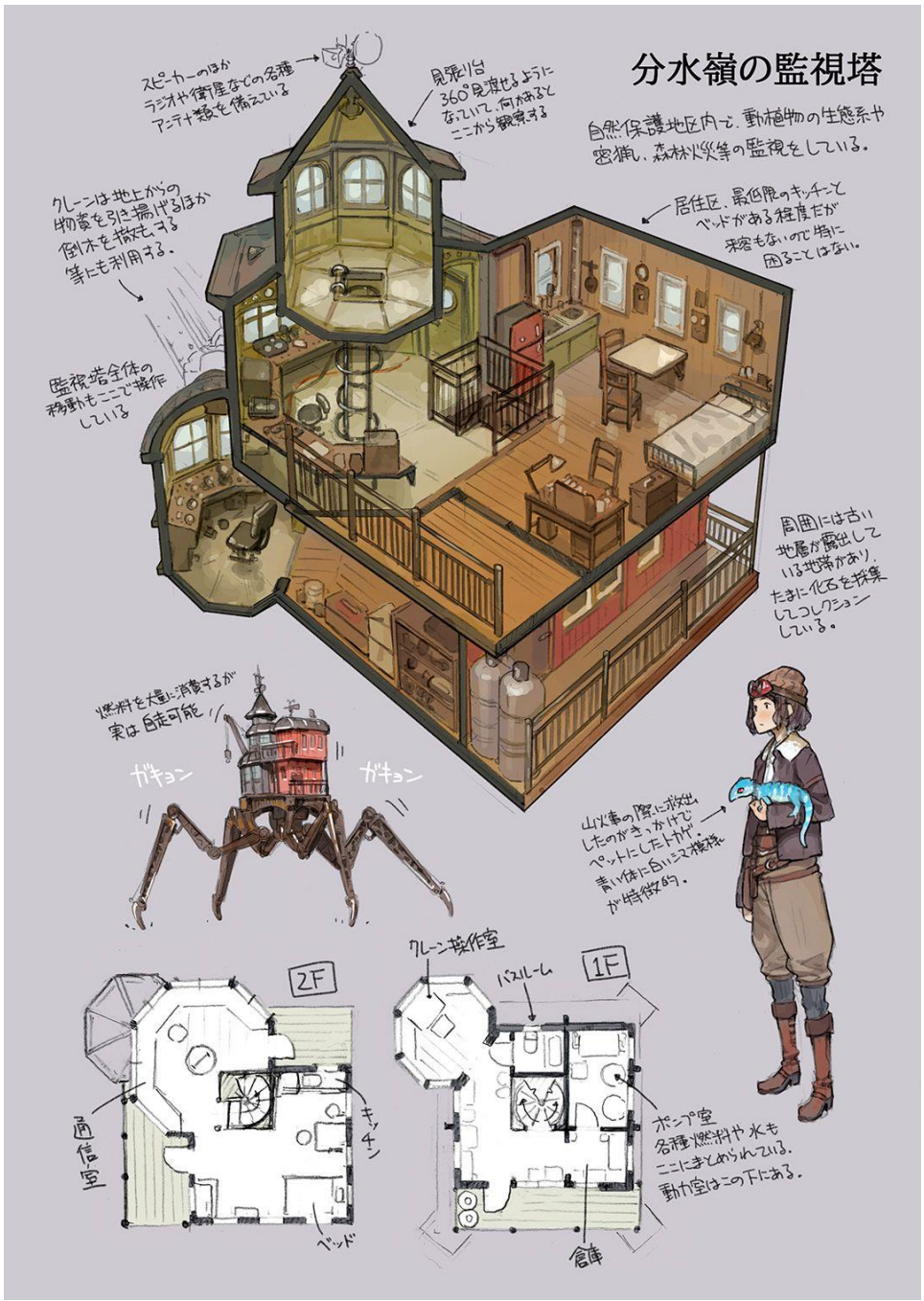
2.6 Inicialización y carga de recursos

Inicialización (en main()): Se inicializan las librerías (GLFW, GLEW), se crea la ventana y se configuran las propiedades de OpenGL (como glEnable(GL_DEPTH_TEST)).

Shaders: se crean objetos Shader (lightingShader, lampShader, skyboxShader). Los shaders encapsulan compilación y enlaces (binding de atributos y uniforms). La documentación alista paths de archivos .vs y .frag y los uniforms principales (model/view/projection, luces, material).

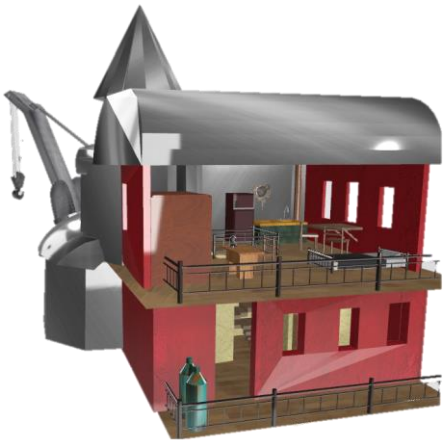
Carga de Recursos (en main()):

Imagen de referencia



- Se instancian las clases Shader para compilar los shaders GLSL (.vs y .frag).
- Se instancian múltiples objetos Model para cargar todas las mallas (.obj):

Models/pata superior.obj	
Models/pata enmedio.obj	
Models/pata inferior.obj	
Models/pata completa.obj	
Models/base.obj	

Models/12casacompleta.obj **La casa viene integrada con los objetos recreados de las referencias**	
Models/dish.obj	
Models/droncuerpo.obj	
Models/helicederecha.obj	
Models/heliceizquierda.obj	
Models/pataizquiredadron.obj	

Models/pataderechadron.obj	
Models/paredfrentesegundopiso.obj	
Models/marco.obj	
Models/puerta.obj	

Tabla listado de objetos

2.7 Diseño del Game Loop y control de tiempo

Estructura: cada iteración calcula deltaTime = currentFrame - lastFrame;, procesa eventos (glfwPollEvents()), llama DoMovement() para actualizar input-driven movement, actualiza animaciones y luego renderiza. Esto hace que las animaciones que usan deltaTime sean consistentes independientemente del framerate.
Se instancian las clases PataAnimada y ReproductorCamara, las cuales llaman a sus métodos loadAnimationFromFile() para cargar los .txt de keyframes en memoria .

Calcula deltaTime (tiempo transcurrido desde el último fotograma).

Procesa entradas (DoMovement()).

Actualiza el estado de todas las animaciones (Dron, Plato, Puerta, Patas y Cámara).

- **Limpia la pantalla (glClear).**
- **Configura los Shaders y las luces (uniforms).**
- **Dibuja todos los objetos 3D (modelo->Draw()).**
- **Dibuja el Skybox al final (con glDepthFunc(GL_LEQUAL)).**
- **Intercambia los búferes de dibujo (glfwSwapBuffers).**

Gestión de Callbacks: Funciones globales (KeyCallback, MouseCallback) que reciben presionadas y movimiento del mouse y actualizan variables globales de estado (como el array keys o activando las animaciones) .

2.8 Lógica de Animación por Keyframes (Patas y Cámara)

El sistema de animación de las patas y de la cámara es idéntico y se basa en la interpolación lineal entre puntos clave.

Estructura de Datos
Se definen dos structs, FRAME y CAM_FRAME , que actúan como contenedores para los valores de pose (ej. pataPosX, rotSuperior para la pata; posX, yaw, pitch para la cámara). Ambas clases de animación (PataAnimada, ReproductorCamara) contienen un array estático de estas estructuras (KeyFrame[MAX_FRAMES]).

Proceso de Reproducción

Carga: loadAnimationFromFile() lee un archivo .txt . La primera línea es un entero N (el FrameIndex). El bucle siguiente lee N líneas y llena el array KeyFrame[].

Activación: startStopAnimation() pone la bandera play en true y reinicia los contadores.

Cálculo de Interpolación: El método interpolation() se llama al inicio de cada tramo (entre dos keyframes). Calcula el delta entre el keyframe playIndex y playIndex + 1, y lo divide por MAX_PASOS_INTERPOLACION (190). El resultado se almacena en las variables Inc (ej. pataPosXInc).

Ciclo de Animación: El método Animation() es llamado en cada fotograma del while loop.

Si play es false, no hace nada.

Si play es true, suma las variables ...Inc a las variables de pose actuales (ej. pataPosX += pataPosXInc;).

Incrementa el contador i_curr_steps.

Cuando i_curr_steps alcanza MAX_PASOS_INTERPOLACION, significa que ha llegado al siguiente keyframe. En ese punto, incrementa playIndex y vuelve a llamar a interpolation() para calcular el siguiente tramo.

2.9. Lógica de Animación Algorítmica y por proximidad

Otras animaciones no usan keyframes, sino que se calculan sobre la marcha (algorítmicamente) en cada fotograma.

Dron (Animación Algorítmica)

El movimiento del Dron es una composición de varias funciones matemáticas que dependen del tiempo:

Dron: posición en XZ calculada por círculo paramétrico usando sin/cos con dronAnguloVuelo +=velocidad*deltaTime y altitud con oscilación sin(glfwGetTime()*2.0f)*amplitud. Hélices giran con dronAnguloHelice += 1000.0f * deltaTime.

Vuelo Circular: La posición (X, Z) se calcula usando un círculo paramétrico. Un ángulo dronAnguloVuelo se incrementa constantemente con deltaTime. La posición se calcula como dronX = sin(dronAnguloVuelo) * radioVuelo y dronZ = cos(dronAnguloVuelo) * radioVuelo.

Tambaleo Vertical: La altura dronY se modifica sumándole el resultado de sin(glfwGetTime() * 2.0f) * 0.2f, creando una pequeña oscilación.

Hélices y Patas: Las hélices giran a una velocidad constante (dronAnguloHelice += 1000.0f * deltaTime). Las patas usan una oscilación similar a la del tambaleo vertical (sin(glfwGetTime() * 2.5f)).

Puerta y Plato (Animación por Proximidad)

En cada ciclo del while loop, se calcula la distancia entre la cámara y estos objetos.

Cálculo de Distancia: float dist = glm::length(camera.GetPosition() - objectPosition);.

Plato (Dish): Si dist < dishActivateDistance, simplemente se incrementa su ángulo de rotación (dishRotation += 90.0f * deltaTime) .

Puerta: Se establece un puertaRotObjetivo (0.0f si está lejos, 95.0f si está cerca). Luego, en lugar de saltar a ese valor, se interpola el puertaRotActual hacia el objetivo usando deltaTime, creando una apertura y cierre.

2.10 Renderizado y uso de matrices (jerarquía de transformaciones)

En PataAnimada::Draw hay una secuencia de model stacking:

Start con model = translate(posOffset), rotaciones de offset.

translate(pataPosX,pataPosY,pataPosZ), luego rotate(rotSuperior) y modeloS_Sup->Draw().

Se guarda modelTemp=model, aplicar translate(ensambleSup), rotate(rotEnmedio), dibujar modeloS_Enm.

Repetir para la pata inferior con ensambleEnm.

Esto implementa la jerarquía, cada segmento hereda la transformación del antecesor (padre → hijo).

Se hace uso de modelTemp ya que es una copia del estado actual de la matriz padre, justo antes de dibujar un nuevo segmento.

model → matriz del padre
modelTemp → matriz hijo (segmento actual)

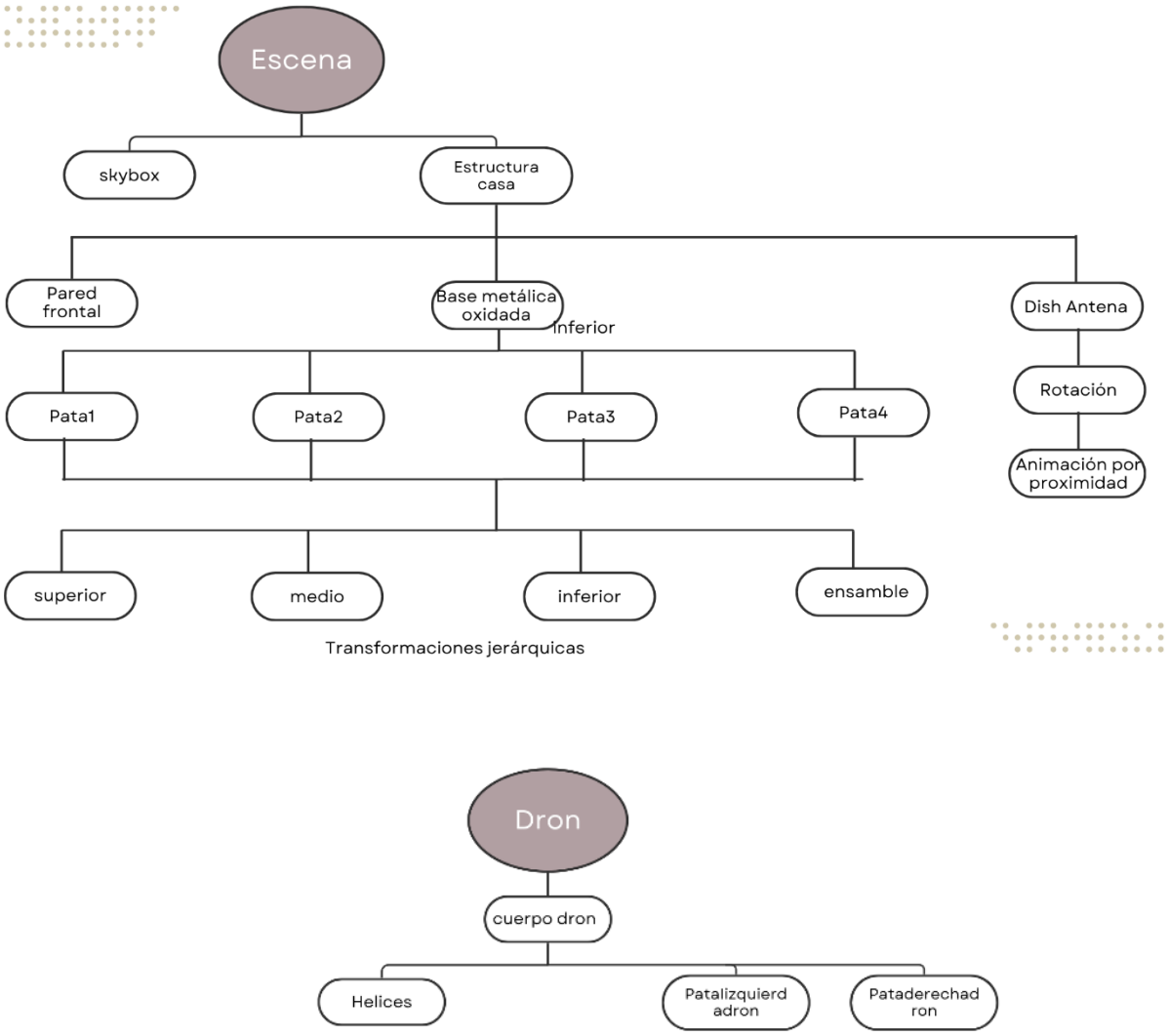
Luego solo se modifica el hijo, sin afectar al padre.

Así se logra que:

- El segmento medio se base en el resultado del superior.
- El inferior se base en el medio.

Ningún segmento afecta a otro hacia arriba.

2.11 Diagrama de Árbol de Escena (Scene Graph)



Representa la estructura jerárquica

Cada nodo hereda transformaciones de su padre (traslación, rotación y escala), y OpenGL calcula la posición final aplicando multiplicación de matrices en orden.

2.12 Reproductor de cámara

Estructura CAM_FRAME contiene posX,posY,posZ,yaw,pitch e incrementos. La lógica es muy similar a la de PataAnimada (precalcula incrementos y usa i_curr_steps para avanzar).

Interacción con Camera: resetElements(cam) escribe directamente en cam.Position, cam.Yaw, cam.Pitch y luego llama a cam.updateCameraVectors().

Modo reproductor vs modo libre: cuando el reproductor está activo, las entradas d hardware se desactivan. Además, cuando se el recorrido las patas también se activan (llamar pataX->startStopAnimation() al activar recorridoCamara)

2.13 Gestión de recursos (memoria y VRAM)

Carga única: cargar cada modelo solo una vez y compartir punteros Model* entre patas y otras instancias. Esto segura que Model internamente almacene sus VAOs/VBOs y libere en destructor.

for each Model* m: delete m;
glDeleteVertexArrays;
glDeleteBuffers;
glfwTerminate();

4.- Análisis de costos del proyecto

En México, para un estudiante universitario de ingeniería (nivel licenciatura) que realiza proyectos freelance de programación o computación gráfica en 2024-2025, la tarifa por hora puede variar dependiendo de factores como experiencia, nivel técnico, complejidad del proyecto

Referencias salariales generales:

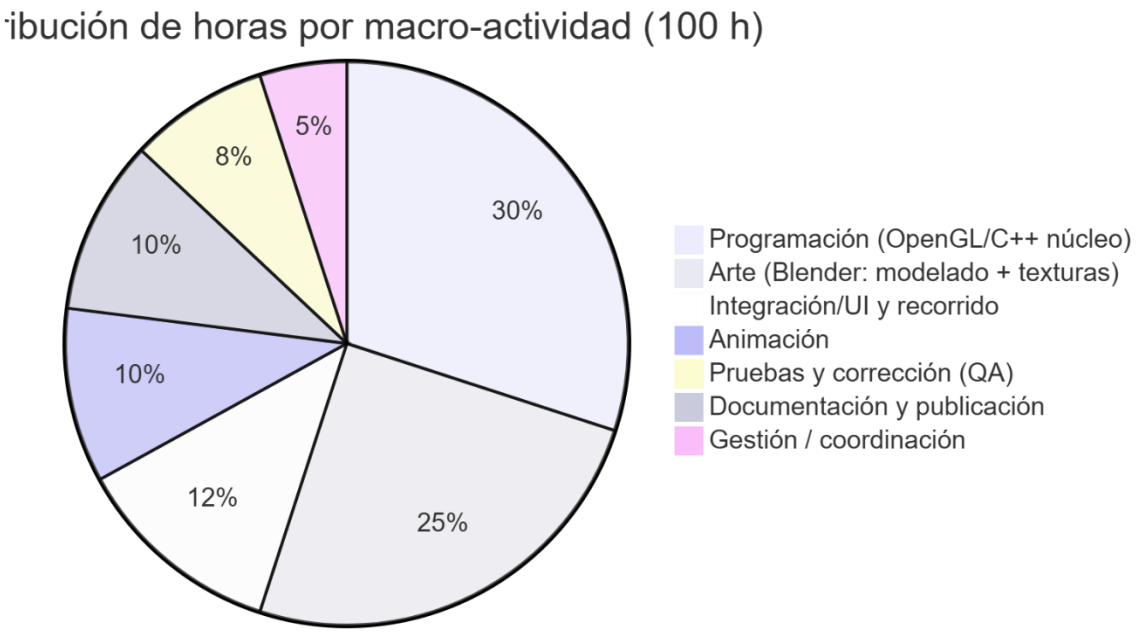
El salario mínimo por hora en México en 2025 se sitúa en torno a $34.85 MXN para el resto del país y $52.48 MXN en la Zona Libre de la Frontera Norte. Estas tarifas son solo referencia para trabajos no especializados y formales.
El ingreso mensual promedio nacional ronda los $8,364 MXN, lo que equivaldría a unos $48 MXN/hora para jornadas intensivas

Resumen del proyecto

Tipo: demo académica de CG.
Arte: casa modelada en Blender, UV y texturas básicas.
Motor/Viewer: C/C++ con OpenGL en Visual Studio (GLFW/GLAD/GLM/Assimp, opcional ImGui).
Plataforma: Windows x86.
Entregables: ejecutable, assets, video demo, manual de usuario y reporte técnico.

Equipo: 1 estudiante (dev CG, artista/integrado).
Horas totales (H): 100
Tarifa estudiantil: 50 MXN/h
Licencias: todo con software libre (Blender, VS Community, librerías OSS).
Hardware: PCs personal.
Nivel: académico con pulido básico

Desglose de esfuerzo por actividades



Distribución refinada por fases

Fase	Nombre	Sub-tareas principales	Horas
1	Planificación y Setup	Referencias y objetos; Configuración VS/Git; Setup librerías (GLFW, GLAD/GLEW, GLM, Assimp)	12
2	Entorno Básico (Modelado + Texturas)	Carga inicial; Modelado fachada; Modelado interior; Objetos; UV y texturas	35
3	Creación de Escenario	Cámara; Carga final; Iluminación básica; Sombras (shadow map)	18
4	Animación	Jerárquica (puertas/partes); Por proximidad (triggers)	10
5	Recorrido	Generador de trayectoria; Grabación de recorrido (video)	8
6	Cierre	Pruebas y corrección Empaquetado Documentación Publicación en GitHub	17
	Total		100

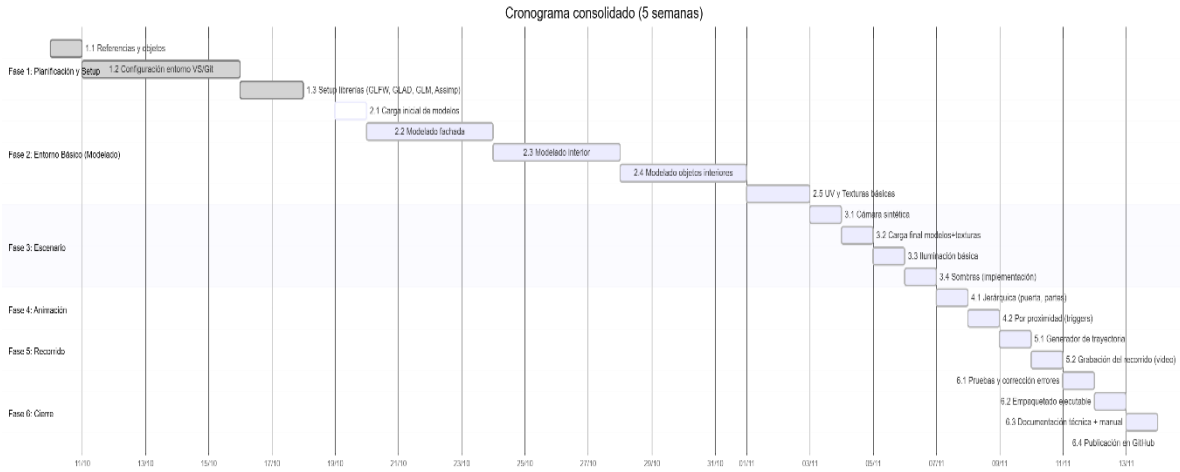
2) Estimación de costos (MXN)

Parámetros
- Horas (H) = 100
- Tarifa (T) = 50 MXN/h

Cálculo
- Costo base = H × T = 100 × 50 = 5,000 MXN
- Total = 5,000 MXN
Total de horas estimadas: 100 h

Actividad	Horas	Costo (MXN)
Programación (OpenGL/C++)	36	1,800
Arte (Blender/OpenGL)	27	1,350
Integración/UI	16	800
Pruebas (QA)	8	400
Documentación	8	400
Gestión/entrega	5	250
Total	100	5,000

Justificación de cronograma



Detalle por fase

Fase 1 (12 h) Planificación y Setup

Objetivos definir alcance, establecer entorno reproducible, asegurar el pipeline gráfico base.

Actividades: selección de referencias arquitectónicas, escala métrica lo más real posible.

- Inicialización repo Git (estructura carpetas, .gitignore, README preliminar).
- Configuración Visual Studio (targets Debug/Release, x64), integración de librerías.

Fase 2 (35 h) Entorno Básico / Modelado y Texturas

Objetivos: representación geométrica completa de la casa (fachada, interior, objetos) con UV y texturas básicas.

Metodología de arte: blocking → refinamiento topológico → UV unwrap → baking / asignación de texturas.

Actividades: modelado secuencial (fachada → interior → objetos), normalización de pivot y escala, creación de materiales simples (albedo, especular mínimo).

- Entregables: carpeta /models con versiones .obj,

Fase 3 (18 h) Creación de Escenario

Objetivos: incorporar el modelo en el visor con cámara, iluminación y sombras que se adaptan a la arquitectura**.**

- Actividades: cámara orbital y libre; iluminación direccional + puntual + ambiental; implementación de sombras (shadow map básico).

Fase 4 (10 h) Animación

Objetivos: demostrar principios de animación jerárquica y reactiva (proximidad).

- Actividades: jerarquía (puertas, articulaciones), triggers por proximidad (sensor distancia cámara → acción).

Fase 5 (8 h) Recorrido

Objetivos: generar y grabar un recorrido narrativo que muestre el ambiente.

- Actividades: interpolación de keyframes de cámara (spline/lerp), control de velocidad y easing, captura video (herramienta externa o frames).

no brusca), calidad de captura ≥ 1080p, 30 FPS sostenidos.

Fase 6 (17 h) Cierre

Objetivos: asegurar calidad, reproducibilidad, documentación y publicación final.

- Actividades: pruebas funcionales, corrección de errores, empaquetado (binarios + assets), redacción de informe técnico y manual de usuario, publicación en GitHub (release v1.0.0).

Entregables

- Ejecutable Windows x86 + carpeta de assets.
- Repositorio con código, shaders y scripts de build.
- Manual de usuario y Manual Técnico (PDF).
- Video demo (2–4 min) mostrando modelado, iluminación y animación.

El código fuente completo del proyecto, incluyendo todos los archivos de cabecera, shaders, modelos 3D y texturas, se encuentra disponible en el repositorio de GitHub. La estructura del proyecto está organizada de forma clara, con carpetas separadas para cada tipo de recurso y comentarios en el código que facilitan su comprensión:

Repositorio GitHub:
https://github.com/soeil1/319079485_ProyectoFinalTeoria_GPO05.git

Para visualizar el funcionamiento completo de la casa y todas sus características implementadas, se preparó un video demostrativo que muestra la navegación el entorno, la activación de las animaciones de cada parte

Video demostrativo:
https://youtu.be/0YKs-a2w2YA

Referencias

- de Vries, J. (2023). *Learn OpenGL: Learn modern OpenGL graphics programming in a step-by-step fashion*. https://learnopengl.com/

- Blender Foundation. (2024). *Blender 4.0 Reference Manual*. https://docs.blender.org/manual/en/latest/

- Khronos Group. (2023). *OpenGL 3.3 Core Profile Specification*. https://www.khronos.org/registry/OpenGL/

- Glassner, A. S. (Ed.). (1989). *An Introduction to Ray Tracing*. Academic Press.
GLFW Documentation. (2023). *GLFW 3.3 Documentation*. https://www.glfw.org/documentation.html

- OpenGL Mathematics (GLM). (2023). *GLM 0.9.9 Documentation*. https://github.com/g-truc/glm