NATIONAL AUTONOMOUS UNIVERSITY OF MEXICO FACULTY OF ENGINEERING DIVISION OF ELECTRICAL ENGINEERING COMPUTER ENGINEERING COMPUTER GRAPHICS AND HUMAN-COMPUTER INTERACTION

Final Project 2
Technical Manual
Computer Graphics and Human-Computer Interaction

Student. 319079485

Professor. ING. CARLOS ALDAIR ROMAN BALBUENA

Group 5

Semester 2026-1

Submission Date: November 25, 2025

Index

Introduction

This project consists of developing an interactive 3D simulation built with **C++ and OpenGL 3.3**, recreating a walking steampunk house equipped with a system of hierarchical animations, proximity-interactive objects, an autonomous drone, and a cinematic tour reproducible via keyframes.
The main goal is to demonstrate the mastery of computer graphics techniques: 3D modeling, texturing, lighting, interpolation animation, hierarchical transformations, synthetic camera control, and GPU resource management.

During execution, the user can:

- Freely explore the post-apocalyptic environment..
- Activate house animations via keyboard input.
- Interact with objects that react to proximity.
- Execute a pre-recorded automatic tour.

The scene includes a desert environment skybox , a drone with autonomous circular flight , doors with distance-dependent opening mechanisms , a rotating parabolic antenna (dish) , and four articulated mechanical legs with movement .

The project implementation involves core concepts of three-dimensional computer graphics.

The rendering pipeline begins with the definition of three-dimensional geometry via vertices, normals, and texture coordinates stored in GPU buffers. Programmable shaders process this information: the vertex shader transforms vertex coordinates from local space to screen space using model, view, and projection matrices , while the fragment shader calculates the final color of each pixel by applying Phong lighting models and texture mapping.

The lighting system implements three types of light sources:

Directional Light: Simulates the sun with parallel rays.

Point Lights: Emit radially from specific positions with quadratic attenuation .

Spotlight: A flashlight-type focal light with a defined aperture angle .

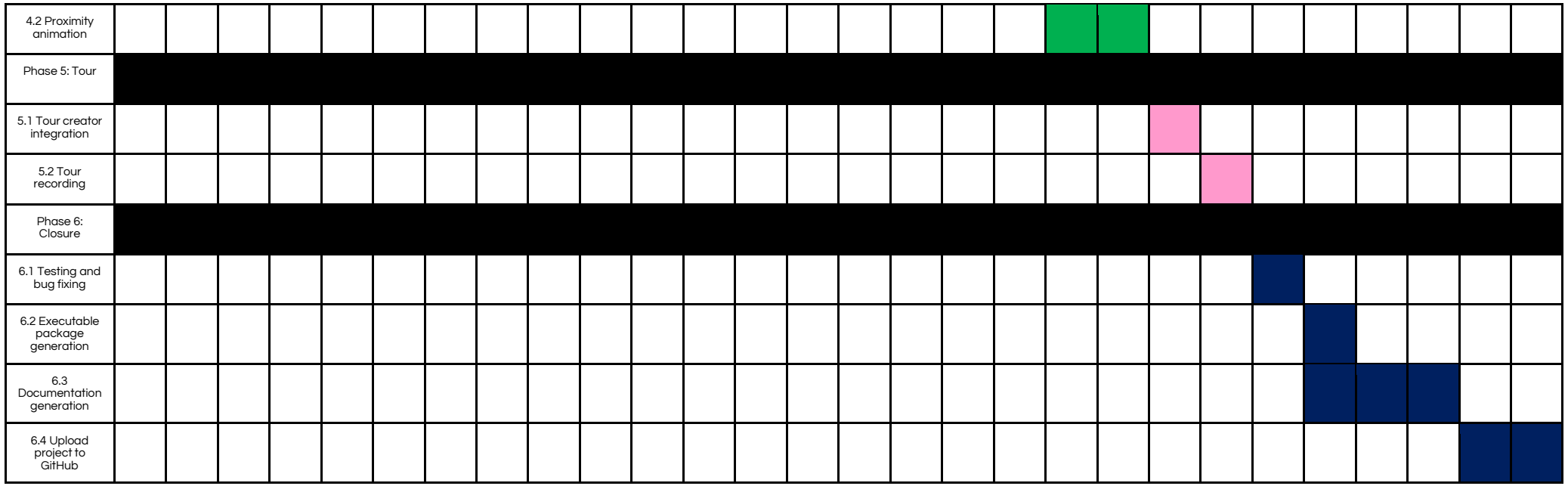
Each light contributes ambient, diffuse, and specular components to the final surface color based on its material and orientation relative to the source.

The project utilizes the GLFW library for window management and input event handling , GLEW to access modern OpenGL extensions , GLM for matrix and vector operations, SOIL2/stb_image for loading images as textures , and a custom parser for .obj files that extracts the geometry of three-dimensional models .

Objectives

The main objective of this project is to develop an interactive 3D scene in real-time using C++ and the OpenGL graphics API.

1.- Gantt Chart	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
Phase 1: Planning and Setup																												
1.1 Selection of references and objects																												
1.2 Environment Configuration (VS, Git)																												
1.3 Library Setup (GLFW, GLEW, GLM)																												
Phase 2: Basic Environment																												
2.1 Shader Loading (Classes)																												
2.2 House facade modeling																												
2.3 House interior modeling																												
2.4 House objects modeling																												
2.5 Texture implementation																												
Phase 3: Scene Creation																												
3.1 Synthetic camera implementation																												
3.2 Loading models and textures																												
3.3 Basic Lighting																												
3.4 Skybox implementation																												
Phase 4: Animation																												
4.1 Hierarchical animation																												


4.2 Proximity animation
Phase 5: Tour
5.1 Tour creator integration
5.2 Tour recording
Phase 6: Closure
6.1 Testing and bug fixing
6.2 Executable package generation
6.3 Documentation generation
6.4 Upload project to GitHub

2.Technical Manual

2.1 Implemented Development Methodology

For this project, I utilized the prototyping methodology, which allowed me to build the steampunk House environment in an iterative and incremental manner. I began with the basic loading of static models and refined each component—such as the drone hierarchy and the leg keyframe system—until achieving the required fluid behavior.

This methodology proved useful for this 3D graphics project, as many visual decisions (such as lighting intensity, skybox scale, or interpolation speed) and interaction decisions (such as the door activation distance) could only be evaluated and adjusted correctly by seeing the system rendered in real-time, modifying it through versions until reaching a final one.

Version Control

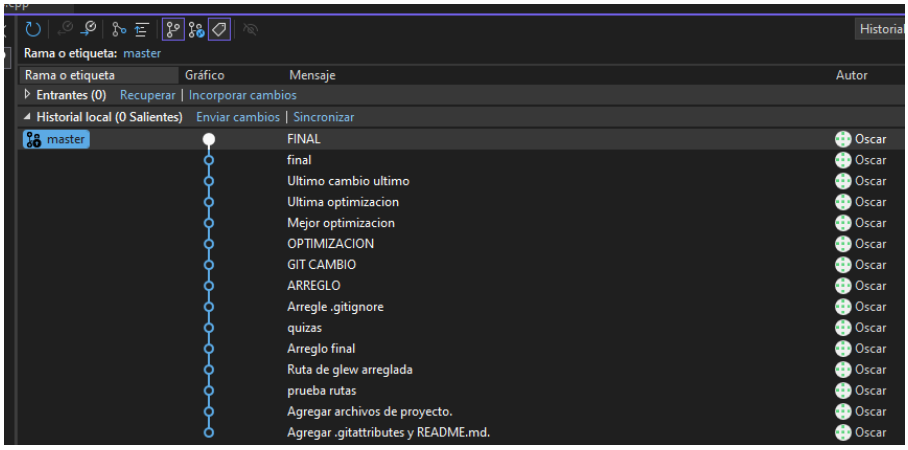
Throughout the entire development process, I used Git and GitHub as a version control system internally until publishing a final version. This was fundamental for organizing the workflow and managing the large number of files and code involved.

I implemented a repository management strategy that allowed me to:

Securely control changes in the source code (.cpp, .h).

Manage external dependencies (libraries and .dlls) and heavy resources (.obj models and textures) through precise .gitignore file configuration, avoiding the upload of temporary or unnecessary compilation files.

Maintain stable versions of the project while experimenting with new features, such as the implementation of the camera tour recorder, ensuring that if any version did not work, I could revert to a previous branch at any time.



2.2 Justification of Technology Use

The technological selection for this project responds to the need for high performance, full control over the graphics pipeline, and industry-standard compatibility.

Blender (3D Modeling and Exporting) Blender was chosen because it is a versatile, open-source 3D creation application with extensive modeling options and a very user-friendly learning curve. Its ability to manage complex meshes, UV mapping, and object hierarchies facilitates the creation of optimized assets for real-time engines.

Furthermore, its native exporter to the Wavefront (.obj) format generates clean and readable files containing geometry (vertices, normals) and texture coordinates, which facilitates parsing by loading libraries like Assimp within the application and coding process.

C++ (Programming Language) C++ is the standard in the computer graphics and video game engine industry due to its efficiency and memory management.

C++ allows low-level control over hardware resources, ensuring that the rendering loop (Game Loop) and complex mathematical calculations (animation interpolations, matrix transformations) execute very quickly, maintaining a stable Frames Per Second (FPS) rate.

OpenGL (Graphics API) OpenGL (Open Graphics Library) was selected as the application programming interface for rendering. Being a cross-platform and hardware-independent specification, it ensures that the project can run on a wide range of devices, as observed when creating the executables.

Unlike commercial game engines (such as Unity or Unreal) that hide complexity, using pure OpenGL allows for the implementation and understanding of the programmable graphics pipeline from scratch.

2.3 State of the Art

Current Context of Computer Graphics In the current landscape of real-time computer graphics (2024-2025), the industry has transitioned towards hybrid rendering techniques that combine traditional rasterization with Ray Tracing in real-time, powered by dedicated hardware (such as NVIDIA's RTX series). Modern graphics engines like Unreal Engine 5 use technologies like Nanite (virtualized geometry) and Lumen (dynamic global illumination) that abstract the complexity of the graphics pipeline.

This project is situated in the domain of Low-Level Graphics Programming using OpenGL 3.3 Core Profile. Although the industry uses more modern APIs like Vulkan or DirectX 12, OpenGL remains the academic and professional standard for understanding the fundamentals of the programmable pipeline.

Unlike using a commercial engine where physics and animation are "black boxes," in this project, I manually implemented the mathematical algorithms that constitute the foundation of such engines:

Hierarchical Animation and Forward Kinematics:

Modern engines: Use Inverse Kinematics (IK) and Skeletal Animation processed on the GPU (Vertex Skinning).

Project Implementation: I implemented a hierarchy of matrix transformations (parent-child) processed on the CPU. This demonstrates the fundamental principle of how model matrices propagate through a Scene Graph, just as Unity or Godot manages internally before sending it to the GPU.

Interpolation System (Keyframing):

Modern engines: Use Bézier curves or Cubic Splines to smooth complex movements, and Animation Blueprints to blend states.

Project Implementation: I developed a proprietary linear interpolation engine that reads raw data from text files (.txt). This simulates the basic functioning of standard file formats like .anim or .fbx.

While the state of the art seeks photorealism through AI (DLSS) and Ray Tracing, this project focuses on the efficiency of the Rasterization Pipeline, demonstrating how to manage multiple draw calls, shader uniforms, and geometric transformations explicitly.

Current Landscape of Virtual Tours Currently, virtual tours have evolved from simple collections of static 360° images (panoramas) to become fully immersive, real-time rendered three-dimensional experiences.

Dominant Trends:

Real-Time Rendering: Engines like Unreal Engine 5 and Unity have standardized architectural (ArchViz) and tourism visualization, allowing the user to move freely with 6 degrees of freedom (6DOF), instead of jumping between fixed points.

Virtual Cinematography: The use of virtual cameras programmed to follow smooth paths (Splines) is a standard in the film and video game industry to create cutscenes without needing to render pre-recorded video, saving memory and allowing dynamic changes in the scene.

Reactive Environments: The current trend is not just to see, but to interact. Modern environments ("Smart Environments") react to user proximity, activating lights, opening doors, or triggering mechanical animations automatically.

Industry Application The navigation and animation systems simulated in this project are currently used in:

Architecture and Real Estate: For pre-sale tours where the client can walk through a house that does not yet exist, seeing how doors open or how lighting changes.

Industrial Simulation: Training operators of drones or heavy machinery through simulators that replicate the physics and movement hierarchies of real robots.

Project Foundation in the Current Context This project implements the fundamental principles governing these commercial technologies, but built from scratch in C++ and OpenGL:

Automated Camera System (Virtual Cinematography):

In the industry: Tools like Unreal Engine's "Sequencer" are used, allowing keyframes to be placed on a timeline.

In this project: The ReproductorCamara and CameraRecorder modules were developed, functionally emulating a Sequencer. By recording position (XYZ) and orientation (Yaw/Pitch) and then linearly interpolating them, we recreate the mathematical basis of how a modern graphics engine stores and plays back a cinematic.

Procedural and Hierarchical Animation:

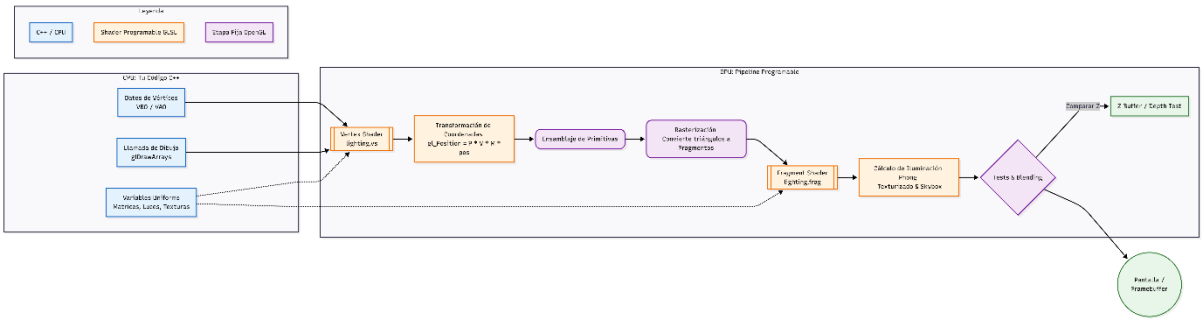
In the industry: Complex skeletons (rigs) are used for machinery and robots.

In this project: The implementation of the Drone and the Walking House using hierarchical matrix transformations (Parent -> Child) demonstrates how complex articulated objects are built without relying on pre-rendered animations, allowing the code to control movement in real-time (algorithmic animation).

Resource Optimization:

While current trends demand powerful hardware (RTX), this project aligns with the trend of efficient computer graphics. By utilizing classic techniques like the Phong lighting model and manual memory management, a fluid virtual tour is achieved that is accessible on mid-range hardware, ideal for mobile devices.

2.4 Graphics Pipeline Diagram



Description of the Implemented Graphics Pipeline
The scene rendering follows the standard OpenGL 3.3 workflow, processing geometry in the following stages:

Vertex Specification (CPU): From the C++ code, vertex attributes (position, normals, texture coordinates) stored in VBOs and organized by the VAOs of each model (House, Drone, Legs) are sent to the GPU.

Vertex Shader (lighting.vs):

This programmable stage receives the raw vertices.

Matrix transformations are applied: Model (object position), View (camera), and Projection (perspective), calculating the final vertex position on the screen (gl_Position).

Rasterization (Fixed Stage): The graphics hardware takes the processed triangles and interpolates the data to generate "fragments" (potential pixels) that will cover the screen.

Fragment Shader (lighting.frag):

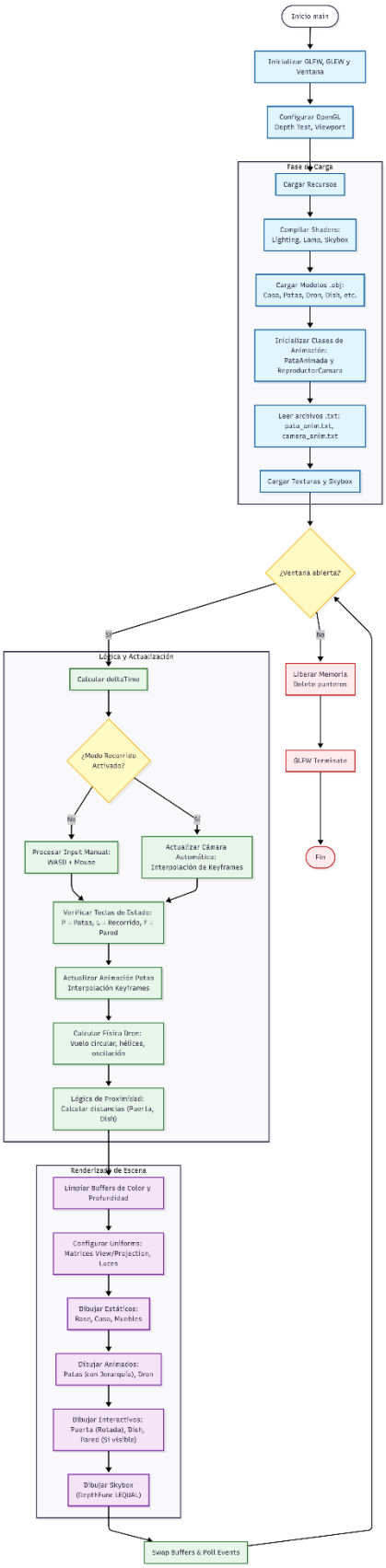
This stage determines the final color of each pixel.

The Phong Lighting Model is implemented, calculating the interaction between surface normals and light sources (directional, point, and spotlight).

Texture sampling is performed (Diffuse and Specular maps), which were previously loaded using SOIL2.

Per-Sample Operations: The Depth Test is executed by comparing against the Z-Buffer to ensure that near objects correctly occlude distant ones (e.g., ensuring the legs are not drawn over the house if they are physically behind it).

Application Flow (Game Loop)



Here is the English translation for the Application Flow (Game Loop) section of your Technical Manual.

Application Flow (Game Loop)
Loading Phase (Blue): Occurs only once. Uses Assimp to load models and the PataAnimada class to read .txt files.

Input Decision (Yellow): Shows the code logic. If the user activates the tour (L), the program ignores mouse/keyboard input and uses data from CameraRecorder.

Update (Green): Shows how new positions are calculated.

Rendering (Purple): The order in which OpenGL draws elements. The Skybox is drawn last for optimization.

The software architecture follows the standard design pattern for real-time graphics applications, known as the Game Loop. This cycle is infinite and executes thousands of times per second until the user decides to close the application. The flow is divided into three stages:

1. Initialization Phase (Setup)
Before entering the infinite loop, the program prepares the environment and loads all necessary resources into RAM and VRAM (Video RAM). This stage executes only once at the beginning.

OpenGL Context and Window: The GLFW library is initialized to create an operating system window and a valid OpenGL context. GLEW is configured to link modern graphics card functions.

Global Configuration: Critical engine capabilities are enabled, such as the Z-Buffer (glEnable(GL_DEPTH_TEST)), so that 3D objects are correctly occluded based on their depth.

Shader Compilation: GLSL files (.vs and .frag) are read, compiled, and linked to create shader programs (Shader lightingShader, Shader skyboxShader).

Resource Loading (Assets):

Models: The Model class uses the Assimp library to read .obj files. Vertices, normals, and texture coordinates are processed and stored in VAOs (Vertex Array Objects) and VBOs (Vertex Buffer Objects) on the GPU.

Textures: Diffuse and specular images are loaded, as well as the 6 textures of the cubic map (Cubemap) for the Skybox.

Animations: The PataAnimada and ReproductorCamara classes open text files (.txt), parse the keyframe information, and store it in dynamic memory data structures.

2. The Main Loop
(while (!glfwWindowShouldClose(window))). In each iteration (frame), the following operations are performed sequentially:

A. Time Calculation (DeltaTime): The time elapsed between the current and previous frame is calculated (currentFrame - lastFrame). This deltaTime variable is important for multiplying all movements and transformations, ensuring that animation speed is constant and independent of the computer's FPS (Frames Per Second).

B. Input Processing: Peripheral events are detected.

The DoMovement() function checks the keyboard state.

Control State Machine: If the play variable of ReproductorCamara is true (Tour Mode), manual movement input (WASD) is ignored. If it is false, the user is allowed to control the camera freely.

C. Logic and Update: Here, the new positions and states of objects are calculated before drawing anything.

Keyframe Interpolation: Instances of PataAnimada calculate the intermediate position between two keyframes based on elapsed time.

Drone Kinematics: Parametric equations for circular flight (sin/cos) are resolved, and the transformation matrices of its child components (propellers and legs) are updated to follow the main body.

Proximity Logic: The vector distance between the camera and interactive objects (Door, Dish) is calculated. If the distance is less than the threshold, their rotation variables are updated toward the active state.

D. Rendering (Draw): Instructions are sent to the GPU to paint the screen.

Clearing: Color and depth buffers are cleared (glClear) to remove the trace of the previous frame.

Shader Configuration: Updated matrices (View, Projection) and light positions are sent to the GPU as uniform variables.

Model Drawing: The Draw() method is invoked for each object.

Hierarchical Transformations: For articulated objects (like legs), a base model matrix is applied, followed by local rotation matrices for each joint before drawing the mesh.

Skybox Rendering: The depth function is changed to GL_LEQUAL, and the environment cube is drawn last to optimize performance.

E. Buffer Swapping: OpenGL uses a Double Buffer system. While one image is displayed on the screen (Front Buffer), the program draws the next one in the background (Back Buffer). at the end of the cycle, they are swapped to show the new image instantly, preventing flickering.

3. Finalization (Cleanup)
When the user closes the window, the loop ends. The program frees dynamically allocated memory (new pointers) and terminates GLFW processes to return control to the operating system cleanly.

2.5 General Software Architecture

The project is contained within a single main source file (Maquina de estados.cpp) that manages all aspects of the program. The architecture can be understood through these layers:

1.-Initialization and Resource Management: Window creation, OpenGL context setup, and loading of shaders and models (in main() and loading sections) .

2.-Rendering Subsystem: Shaders, sending uniforms, VAO/VBO configuration, skybox, and draw calls via Model->Draw() .

3.-Input and Camera Subsystem: Callbacks (KeyCallback, MouseCallback) , DoMovement() , and the Camera class which maintains Position, Yaw, Pitch, and updateCameraVectors().

4.- Animation Subsystem:

- Keyframe animations: (PataAnimada , ReproductorCamara)
- Algorithmic animations: (Drone, propellers, oscillations) .
- Proximity animation: (Door , Dish).

5.- Game Loop: Synchronizes time (deltaTime), processes inputs, updates animations, configures lights, and performs rendering per frame

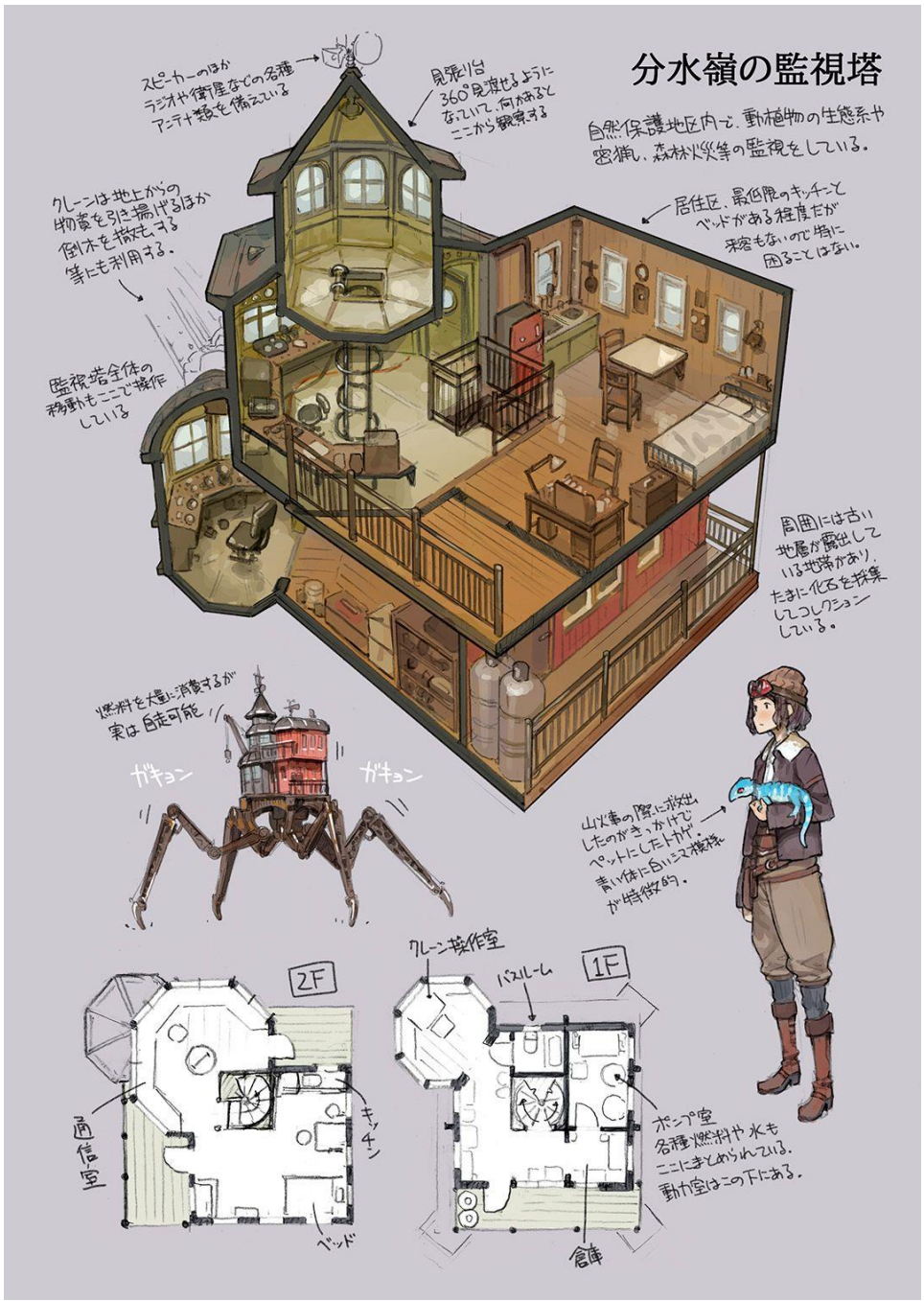
2.6 Initialization and Resource Loading

Initialization (in main()): Libraries (GLFW, GLEW) are initialized, the window is created, and OpenGL properties are configured (such as glEnable(GL_DEPTH_TEST)) .

Shaders:

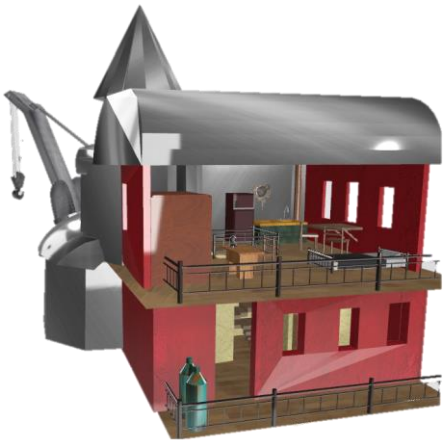
Shader objects are created (lightingShader, lampShader, skyboxShader) . The shaders encapsulate compilation and linking (attribute and uniform binding). The documentation lists paths for .vs and .frag files and the main uniforms (model/view/projection, lights, material) .

Resource Loading (in main()):
Reference image



- Shader classes are instantiated to compile the GLSL shaders (.vs and .frag).
- Multiple Model objects are instantiated to load all the meshes (.obj):

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

Models/12casacompleta.obj	
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	

List of Loaded 3D Objects

2.7 Game Loop Design and Time Control

Structure: Each iteration calculates deltaTime = currentFrame - lastFrame;, processes events (glfwPollEvents()), calls DoMovement() to update input-driven movement, updates animations, and then renders. This makes animations using deltaTime consistent regardless of the framerate.

The PataAnimada and ReproductorCamara classes are instantiated, which call their loadAnimationFromFile() methods to load the keyframe .txt files into memory .

- Calculates deltaTime (time elapsed since the last frame).
- Processes inputs (DoMovement()).
- Updates the state of all animations (Drone, Dish, Door, Legs, and Camera).
- Clears the screen (glClear).
- Configures Shaders and lights (uniforms).
- Draws all 3D objects (model->Draw()).
- Draws the Skybox last (with glDepthFunc(GL_LEQUAL)).
- Swaps the drawing buffers (glfwSwapBuffers).

Callback Management: Global functions (KeyCallback, MouseCallback) receive key presses and mouse movement, updating global state variables (such as the keys array or activating animations).

2.8 Keyframe Animation Logic (Legs and Camera)

The animation system for the legs and the camera is identical and is based on linear interpolation between key points.

Data Structure Two structs are defined, FRAME and CAM_FRAME , which act as containers for pose values (e.g., pataPosX, rotSuperior for the leg; posX, yaw, pitch for the camera). Both animation classes (PataAnimada, ReproductorCamara) contain a static array of these structures (KeyFrame[MAX_FRAMES]).

Playback Process

- **Loading:** loadAnimationFromFile() reads a .txt file. The first line is an integer N (the FrameIndex). The following loop reads N lines and fills the KeyFrame[] array.
- **Activation:** startStopAnimation() sets the play flag to true and resets the counters.
- **Interpolation Calculation:** The interpolation() method is called at the beginning of each segment (between two keyframes). It calculates the delta between keyframe playIndex and playIndex + 1, and divides it by MAX_PASOS_INTERPOLACION (190). The result is stored in the Inc variables (e.g., pataPosXInc).
- **Animation Cycle:** The Animation() method is called in every frame of the while loop.
    - If play is false, it does nothing.
    - If play is true, it adds the ...Inc variables to the current pose variables (e.g., pataPosX += pataPosXInc;).
    - Increments the i_curr_steps counter.
    - When i_curr_steps reaches MAX_PASOS_INTERPOLACION, it means the next keyframe has been reached. At that point, it increments playIndex and calls interpolation() again to calculate the next segment.

2.9. Proximity and Algorithmic Animation Logic

Other animations do not use keyframes but are calculated on the fly (algorithmically) in each frame.

Drone (Algorithmic Animation)

The Drone's movement is a composition of several time-dependent mathematical functions:

- **Drone:** Position in XZ calculated by a parametric circle using sin/cos with dronAnguloVuelo += speed * deltaTime and altitude with oscillation sin(glfwGetTime() * 2.0f) * amplitude. Propellers rotate with dronAnguloHelice += 1000.0f * deltaTime.

- **Circular Flight:** The position (X, Z) is calculated using a parametric circle. A dronAnguloVuelo angle is constantly incremented with deltaTime. The position is calculated as dronX = sin(dronAnguloVuelo) * radioVuelo and dronZ = cos(dronAnguloVuelo) * radioVuelo.

- **Vertical Wobble:** The dronY height is modified by adding the result of sin(glfwGetTime() * 2.0f) * 0.2f, creating a small oscillation.

- **Propellers and Legs:** The propellers rotate at a constant speed (dronAnguloHelice += 1000.0f * deltaTime). The legs use an oscillation similar to the vertical wobble (sin(glfwGetTime() * 2.5f)).

Door and Dish (Proximity Animation)

In each cycle of the while loop, the distance between the camera and these objects is calculated**.**

- **Distance Calculation: float dist = glm::length(camera.GetPosition() - objectPosition);.**
- **Dish: If dist < dishActivateDistance, its rotation angle is simply incremented (dishRotation += 90.0f * deltaTime).**
- **Door: A puertaRotObjetivo is established (0.0f if far, 95.0f if near). Then, instead of jumping to that value, puertaRotActual is interpolated toward the target using deltaTime, creating an opening and closing effect.**

2.10 Rendering and Matrix Usage (Transformation Hierarchy)

In PataAnimada::Draw, there is a model stacking sequence:

1. Start with model = translate(posOffset), offset rotations.
2. translate(pataPosX, pataPosY, pataPosZ), then rotate(rotSuperior) and modeloS_Sup->Draw().
3. modelTemp = model is saved, apply translate(ensambleSup), rotate(rotEnmedio), draw modeloS_Enm.
4. Repeat for the lower leg with ensambleEnm.

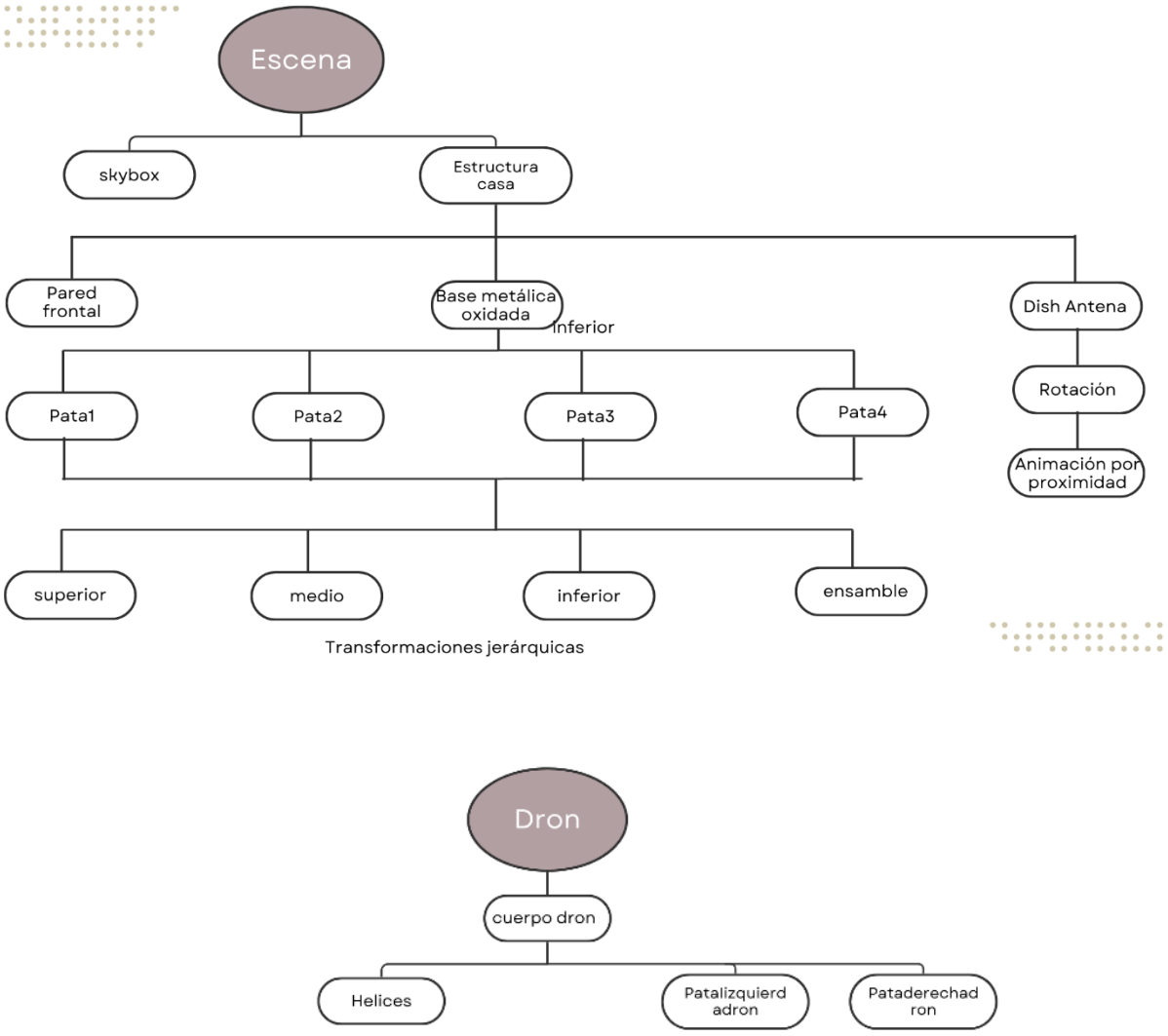
This implements the hierarchy; each segment inherits the transformation of the predecessor (parent → child).
modelTemp is used as it is a copy of the current state of the parent matrix, just before drawing a new segment.

- model → parent matrix
- modelTemp → child matrix (current segment)

Then only the child is modified, without affecting the parent.
This achieves that:

- The middle segment is based on the result of the upper one.
- The lower segment is based on the middle one.
- No segment affects another upwards.

2.11 Scene Tree Diagram (Scene Graph)



Represents the hierarchical structure

Each node inherits transformations from its parent (translation, rotation, and scale), and OpenGL calculates the final position by applying matrix multiplication in order.

2.12 Camera Player

Structure: The CAM_FRAME struct contains posX, posY, posZ, yaw, pitch, and their respective increment values. The logic is very similar to that of PataAnimada (it pre-calculates increments and uses i_curr_steps to advance through the animation).

Interaction with Camera: The resetElements(cam) method writes directly to cam.Position, cam.Yaw, and cam.Pitch, and subsequently calls cam.updateCameraVectors() to ensure the camera's internal vectors match the new orientation.

Playback Mode vs. Free Mode: When the player is active, hardware inputs (keyboard/mouse) are disabled to prevent conflict. Additionally, when the tour is triggered, the leg animations are also activated (calling pataX->startStopAnimation() simultaneously when activating recorridoCamara).

2.13 Resource Management (Memory and VRAM)

Single Loading: Each model is loaded only once, and Model* pointers are shared between the legs and other instances. This ensures that the Model class internally stores its VAOs/VBOs efficiently and releases them correctly in the destructor.

Cleanup Process:

Loop through Model* pointers: delete m;

glDeleteVertexArrays;

glDeleteBuffers;

glfwTerminate();

4. Project Cost Analysis

In Mexico, for a university engineering student (undergraduate level) undertaking freelance programming or computer graphics projects in 2024-2025, the hourly rate can vary depending on factors such as experience, technical level, and project complexity.

General Salary References: The minimum hourly wage in Mexico in 2025 is around $34.85 MXN for the rest of the country and $52.48 MXN in the Northern Border Free Zone. These rates are only a reference for non-specialized and formal work. The national average monthly income is around $8,364 MXN, which would be equivalent to approximately $48 MXN/hour for intensive workdays.

Project Summary
Type: Academic Computer Graphics (CG) Demo.

Art: House modeled in Blender, UV mapping, and basic textures.

Engine/Viewer: C/C++ with OpenGL in Visual Studio (Libraries: GLFW, GLEW/GLAD, GLM, Assimp).

Platform: Windows x86.

Deliverables: Executable, assets, demo video, user manual, and technical report.

Resource Breakdown
Team: 1 Student (CG Developer, Artist/Integrator).

Total Hours (H): 100 hours.

Student Rate: $50 MXN/h.

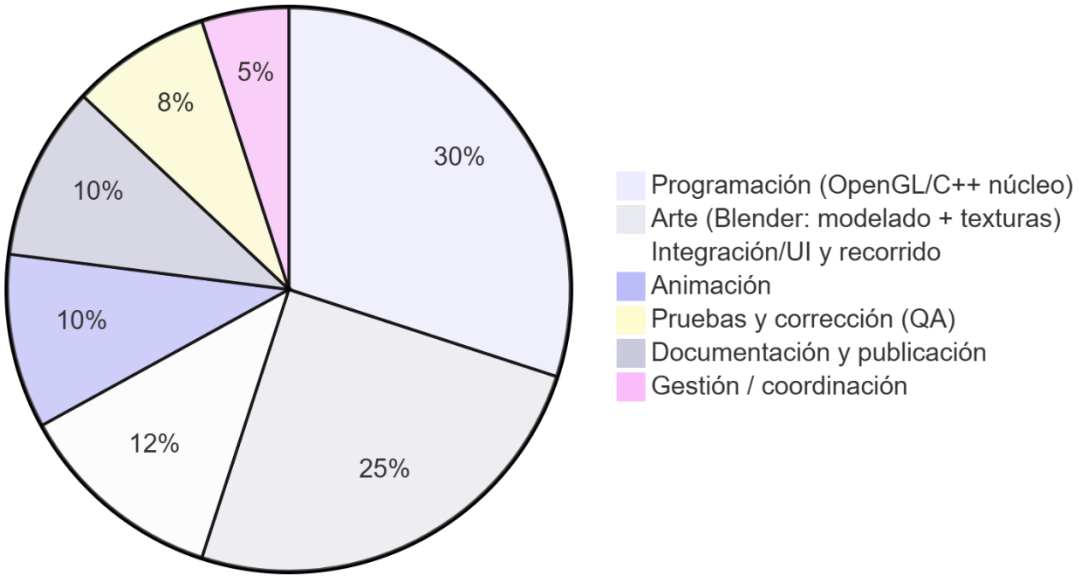
Licenses: All developed with free software (Blender, VS Community, OSS libraries).

Hardware: Personal PC.

Level: Academic with basic polish.

Effort Breakdown by Activity

ibución de horas por macro-actividad (100 h)



Refined Distribution by Phases

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) Cost Estimation (MXN))

Parameters

- **Hours (H) = 100**
- **Rate (R) = $50 MXN/h**

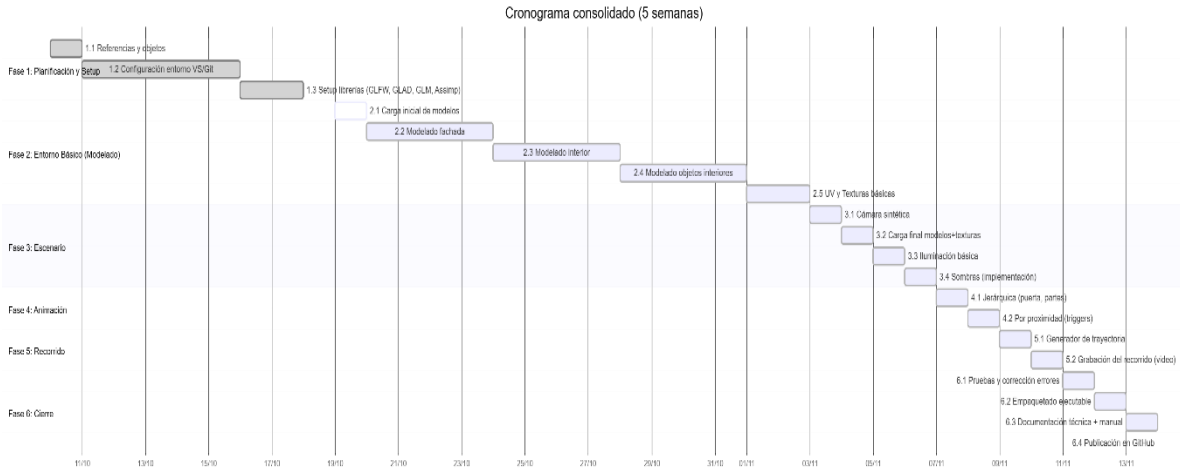
Calculation

- **Base Cost = H × R = 100 × 50 = $5,000 MXN**
- **Total = $5,000 MXN**

Total Estimated Hours: 100 h

Activity	Hours	Cost (MXN)
Programming (OpenGL/C++)	36	1,800
Art (Blender/OpenGL)	27	1,350
Integration / UI	16	800
Testing (QA)	8	400
Documentation	8	400
Management / Delivery	5	250
Total	100	5,000

Justificación de cronograma



Schedule Justification

Phase 1 (12 h) Planning and Setup

- **Objectives:** Define scope, establish a reproducible environment, and ensure the base graphics pipeline.
- **Activities:**
  - Selection of architectural references, ensuring metric scale is as real as possible.
  - Git repo initialization (folder structure, .gitignore, preliminary README).
  - Visual Studio Configuration (Debug/Release targets, x64), library integration.

Phase 2 (35 h) Basic Environment / Modeling and Texturing

- **Objectives:** Complete geometric representation of the house (facade, interior, objects) with UVs and basic textures.
- **Art Methodology:** Blocking → Topological refinement → UV unwrap → Baking / Texture assignment.
- **Activities:**
  - Sequential modeling (facade → interior → objects), pivot and scale normalization, creation of simple materials (albedo, minimal specular).
- **Deliverables:** /models folder with .obj versions.

Phase 3 (18 h) Scene Creation

- **Objectives:** Incorporate the model into the viewer with camera, lighting, and shadows adapted to the architecture.
- **Activities:** Orbital and free camera; directional + point + ambient lighting; shadow implementation (basic shadow map).

Phase 4 (10 h) Animation

- **Objectives:** Demonstrate principles of hierarchical and reactive (proximity) animation.
- **Activities:** Hierarchy setup (doors, joints), proximity triggers (camera distance sensor → action).

Phase 5 (8 h) Tour

- **Objectives:** Generate and record a narrative tour showcasing the environment.
- **Activities:** Camera keyframe interpolation (spline/lerp), speed control and easing (smooth, non-abrupt movement), video capture (external tool or frames) at capture quality ≥ 1080p, sustained 30 FPS.

Phase 6 (17 h) Closing

- **Objectives:** Ensure quality, reproducibility, documentation, and final publication.
- **Activities:** Functional testing, bug fixing, packaging (binaries + assets), writing the technical report and user manual, GitHub publication (release v1.0.0).

Deliverables

- Windows x86 Executable + Assets folder.
- Repository with code, shaders, and build scripts.
- User Manual and Technical Manual (PDF).
- Demo Video (2–4 min) showcasing modeling, lighting, and animation.

The complete source code of the project, including all header files, shaders, 3D models, and textures, is available in the GitHub repository. The project structure is clearly organized, with separate folders for each type of resource and comments within the code to facilitate understanding:

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

To visualize the complete operation of the house and all its implemented features, a demonstration video has been prepared. This video showcases the navigation through the environment and the activation of the animations for each part.

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

References

- 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