NATIONAL AUTONOMOUS UNIVERSITY OF MEXICO
FACULTY OF ENGINEERING

COMPUTER ENGINEERING

Computer Graphics and Human-Computer Interaction

Professor: Eng. Carlos Aldair Román Balbuena

Final Project:

Technical Manual

Student: 319252903

Group: 05

Submission Date: November 24, 2025

INDEX

TECHNICAL MANUAL - FINAL PROJECT.

PROJECT DESCRIPTION.

This project consists of the three-dimensional recreation of a fictional facade inspired by the Hello Kitty universe, designed to represent a colorful and friendly environment characteristic of its aesthetics. The development was carried out using OpenGL together with the C++ programming language, employing models created in Blender, which were exported and integrated using the Assimp library.
The scene is composed of four main areas, each with a set of modeled, textured and in some cases animated objects, with the purpose of generating interactivity, dynamism and visual coherence within the 3D environment. However, this technical manual will only describe and explain two of these areas: the main hallway and the laundry room, in order to delve into their structure, components and operation within the project.
This project has educational and demonstrative purposes, and seeks to show the integration between modeling in Blender and real-time rendering through OpenGL.

STATE OF THE ART.

The development of interactive three-dimensional environments has experienced significant growth in recent decades, driven by advances in graphics hardware, real-time rendering techniques and increasingly accessible modeling tools. This project is framed within the field of applied computer graphics, where disciplines such as 3D modeling, shader programming, dynamic lighting management and particle system creation for visual effects converge.

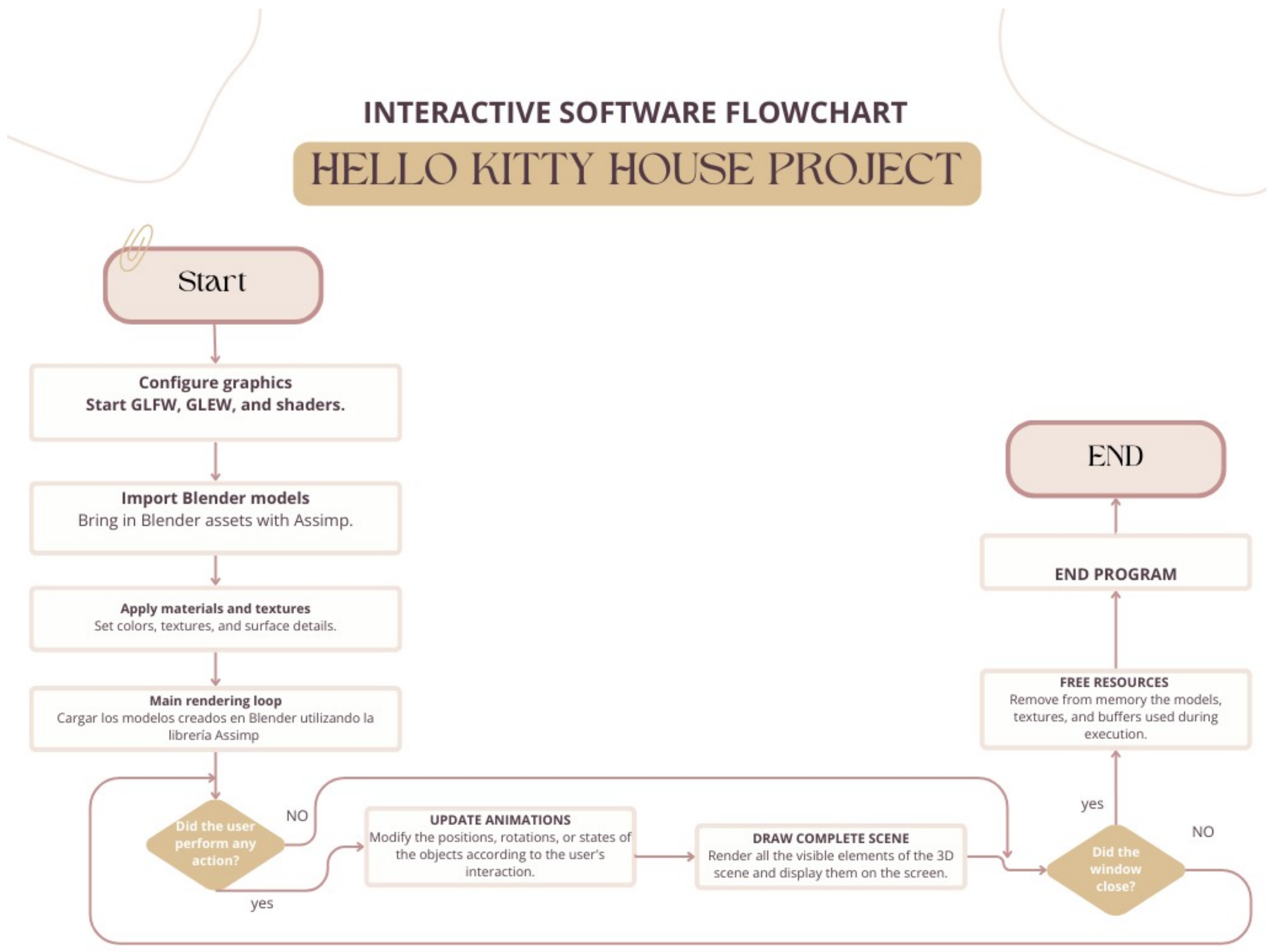
OBJECTIVES.

Develop a complete and functional three-dimensional scene inspired by the Hello Kitty universe, modeled in Blender and implemented in OpenGL using C++, integrating modeling, texturing, animation and interaction processes, and practically applying the knowledge acquired in the theoretical and laboratory sessions of the Computer Graphics course, in order to achieve a coherent, dynamic and properly documented visual environment at a technical level.

The main objectives of this project are as follows:

➔ Model and develop a complete and functional three-dimensional scene inspired by the Hello Kitty universe, recreating an attractive and thematic visual environment.
➔ Apply professional 3D modeling techniques and tools using Blender, ensuring correct topology, texturing and model export.
➔ Implement animations and interactions through OpenGL and C++, in order to provide the scene with dynamism and coherent visual behavior.
➔ Prepare the technical documentation corresponding to the development, structure and operation of the software, ensuring its correct understanding and future maintenance.

SOFTWARE FLOWCHART.


INTERACTIVE SOFTWARE FLOWCHART
HELLO KITTY HOUSE PROJECT
Start
Configure graphics
Start GLFW, GLEW, and shaders.
Import Blender models
Bring in Blender assets with Assimp.
Apply materials and textures
Set colors, textures, and surface details.
Main rendering loop
Cargar los modelos creados en Blender utilizando la librería Assimp
Did the user perform any action?
NO
yes
UPDATE ANIMATIONS
Modify the positions, rotations, or states of the objects according to the user's interaction.
DRAW COMPLETE SCENE
Render all the visible elements of the 3D scene and display them on the screen.
Did the window close?
NO
yes
FREE RESOURCES
Remove from memory the models, textures, and buffers used during execution.
END PROGRAM
END

The diagram shows the execution flow of the 3D interactive software "Hello Kitty House", from its start to the program's closure. The process begins with the graphics environment configuration, where the main rendering components are initialized, such as GLFW, GLEW and shaders, necessary to create the display window and manage real-time graphics.

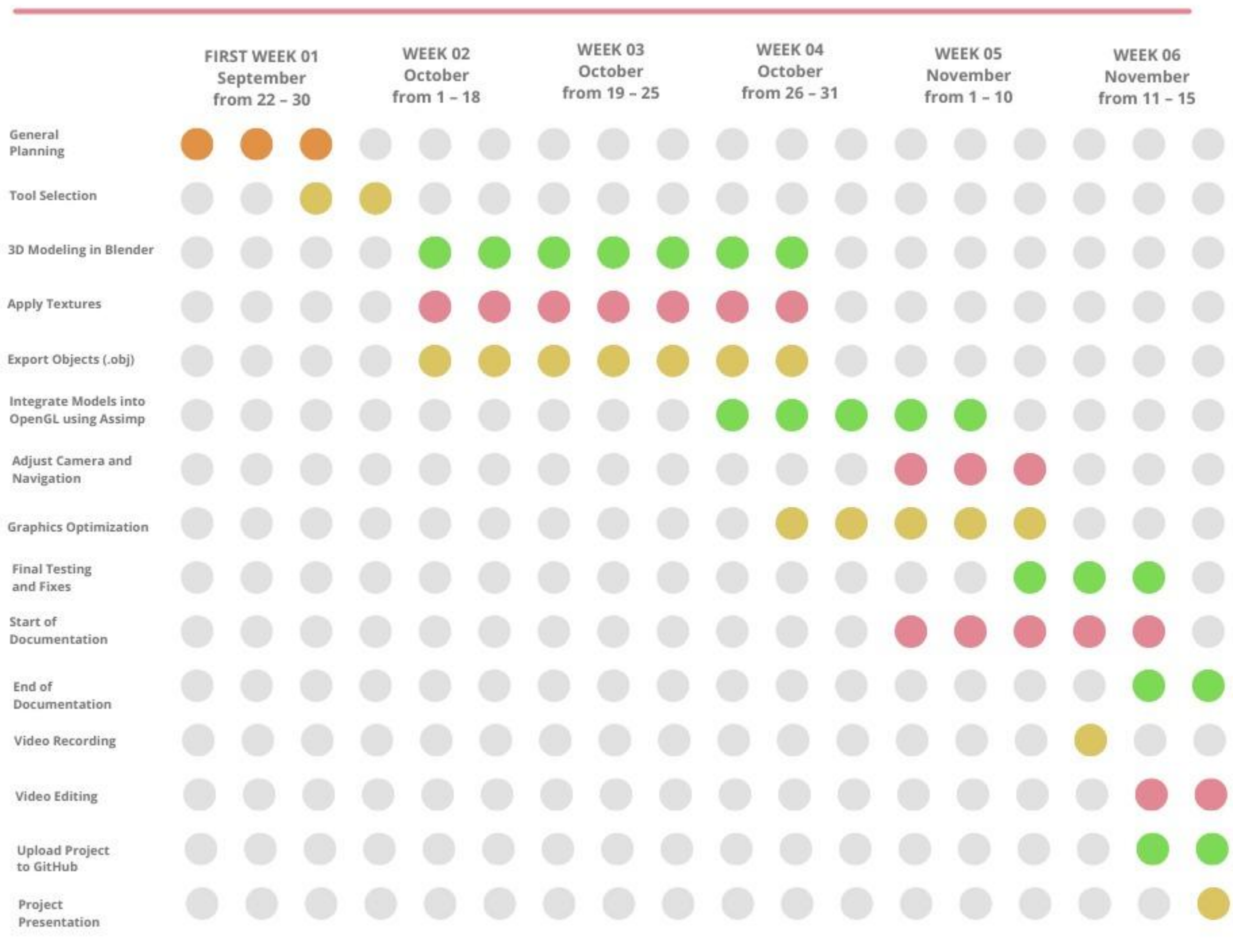
Subsequently, the system imports the 3D models created in Blender, integrating them into the environment through the Assimp library. Next, materials and textures are applied to the objects, defining their colors, surfaces and visual properties to give them a realistic appearance.

Once this preparation phase is completed, the program enters the main rendering loop, where events, animations and scene visualization are continuously managed. Within this loop, the system evaluates whether the user performs any action (such as pressing keys or interacting with the camera). If this occurs, the animations or dynamic elements of the scene are updated.

Afterwards, the software proceeds to draw the complete scene, rendering all visible models and objects with their respective textures and lighting. At the end of each cycle, the system verifies if the window has been closed by the user. If not, the loop continues; if it has been closed, the program releases the used resources (such as models, textures and buffers) and correctly terminates execution.

This flow guarantees a fluid and optimized interactive experience, combining the modeling performed in Blender with the power of OpenGL and C++ to represent a three-dimensional environment inspired by the Hello Kitty universe.

GANTT CHART.



This timeline shows the planning of key tasks for the development of a 3D interactive software project, distributed over a period of six weeks, from September 22 to November 15. Reflecting a progressive workflow, ranging from initial planning to the delivery and final presentation of the project.

1. **Project start.**
   - During Week 1 (September 22-30), general planning is carried out, defining the objectives, scope, resources, team and methodology of the project, establishing the foundation for subsequent development.
2. **Initial preparation.**
   - Between Weeks 1 and 2 (September 22 to October 18), the selection of tools to be used throughout the project is made. This includes choosing the modeling software (Blender), graphics libraries (OpenGL, Assimp), and auxiliary technologies for texturing and rendering. With the tools defined, work environments are configured and the first base files are prepared.
   - In parallel, 3D modeling in Blender begins, starting with simple structures and three-dimensional sketches of the main elements of the environment and the project objects. 3D Modeling in Blender
3. **3D production and texturing.**
   - From Week 2 to Week 4 (October 1-31), a 3D modeling stage is carried out in Blender, where the shapes and characteristics of the objects are detailed.
   - Simultaneously, tasks of applying textures and materials are developed, assigning colors, reflections and surface properties that give realism to the models. Once finished, the models

are exported in .obj format to facilitate their compatibility with OpenGL and the Assimp library, responsible for importing the files within the graphics engine.

4. **Integration and technical adjustments.**
    - During Weeks 4 and 5 (October 26 to November 10), the integration of 3D models in OpenGL is performed using the Assimp library. In this process, loading paths are configured, texture coordinates, lighting and shaders necessary for real-time visualization are adjusted.
    - Subsequently, camera and navigation adjustments are made, defining viewing angles, user movements and interactive controls.
    - Graphic optimization tasks are also executed, aimed at improving program performance, reducing loading times and ensuring a smooth experience.
5. **Testing and Documentation.**
    - Between Week 5 and Week 6 (November 1-15), final testing and project corrections are performed, verifying proper operation and stability of the 3D environment.
    - In parallel, technical documentation is prepared and an explanatory video is created, showing the final result of the development.
6. **Project Closure.**
    - In Week 6 (November 11-15), all closing activities are completed. The complete project is uploaded to the GitHub platform, where source code files, models and documentation are hosted.
    - Finally, the project presentation is prepared, in which the results will be presented, the development process will be explained, marking the official closure of the project.

PROJECT SCOPE.

➔ Complete scene of a Hello Kitty themed house with multiple rooms and different settings.

➔ Integration of models created in Blender and exported in .obj format.

➔ Implementation of interactive animations: doors, curtains, drawers, washing machine drum, clock hands, etc.

➔ Dynamic lighting system with directional light and five point lights.

➔ Advanced graphic effects through particle systems (water and steam).

➔ Native geometry generated using OpenGL (chair, armchair, clock and light switch).

➔ Real-time rendering with Phong lighting techniques, blending, depth testing and custom shaders.

➔ Full keyboard and mouse interaction through a first-person (FPS) camera.

➔ Loading of textures, shaders and models from organized folder structure (Images, Models, Shader).

IMPLEMENTED ROOMS AND THEIR ELEMENTS.

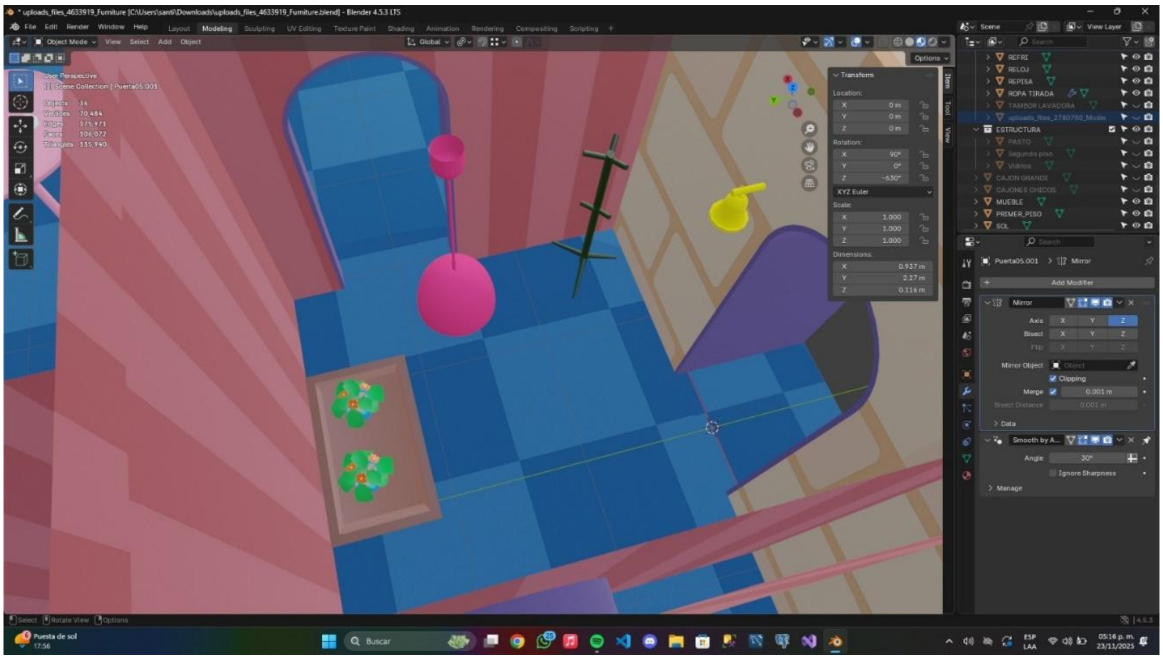
This section describes the rooms developed within the 3D scene, as well as the objects modeled, textured and integrated in each space. It also includes the animations implemented and interactive elements defined during the project development.

Room 1: Entrance Hall.

In this area, the main elements that make up the house entrance hall were integrated, all modeled and textured in Blender, and subsequently exported for implementation in OpenGL. This space functions as the main entrance and contains decorative and functional objects that provide visual identity to the environment.

Modeled and integrated elements:

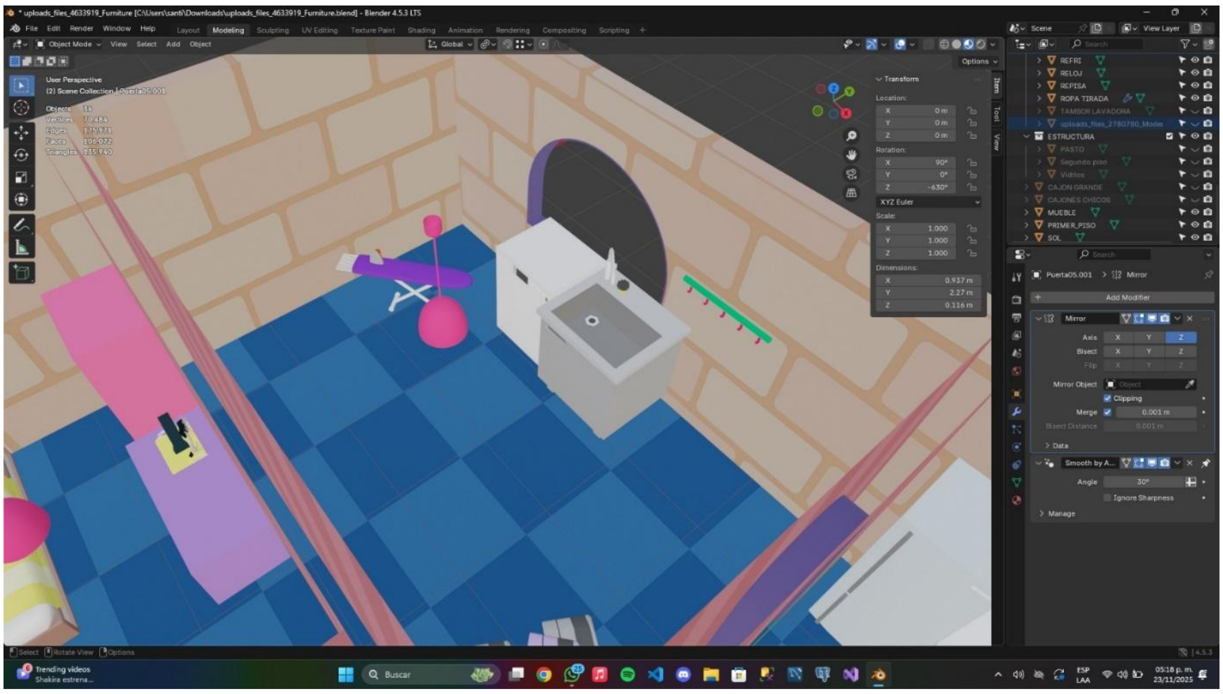
- Bell
- Two-door cabinet
- Coat rack
- Vases
- Flowers


Images. Room 01.

Room 2: Laundry Room.
The laundry room corresponds to the second room developed within the project. This space integrates functional objects and elements typical of a domestic area intended for washing and ironing clothes. All models were created in Blender, optimized and subsequently exported for implementation within the 3D environment in OpenGL.

Modeled and integrated elements:

- Washing machine
- Sink
- Ironing board
- Iron
- Wall-mounted coat rack



Images. Room 02.

IMPLEMENTED ANIMATIONS.

Within the 3D environment, different animations were integrated that allow providing dynamism and interactivity to the objects present in the developed rooms. Each of these animations was programmed in OpenGL through transformations applied in real time, controlled by internal variables and assigned keys within the input system.

DEVELOPED ANIMATIONS:

- Opening and closing of doors
- Rotation of the washing machine drum
- Opening and closing of curtains in the windows
- Iron releasing steam

LIMITATIONS.

➔ No prior knowledge of Blender.

➔ We still had no knowledge of how to texture in Blender before starting this part.

➔ Visual resources limited to academic purposes.

➔ No prior knowledge to use Git and GitHub.

➔ The program was developed and tested on Windows. Its execution on other operating systems such as macOS or Linux might require adjustments.

➔ Recreation of real scenarios is not allowed (UNAM, companies).

➔ Development was performed on Windows-compatible platforms.

COST ANALYSIS OF MODELING AND TECHNICAL PRODUCTION.

1. Costs by technical production

a) Modeling and texturing in Blender Includes the complete creation of the structure, rooms, furniture and animatable objects.

- Complete environment modeling (facade, interiors, interactive objects): ↳ 18 hours approx.

- Texturing, UV organization and materials: ⤷ 8 hours approx.
- Physical simulations with nCloth (sheets, teapot): ⤷ 4 hours approx.
- Total: 30 hours

b) Export and adjustments

- Separation of individual objects
- Pivot adjustment for animations
- Final verification of scales, normals and orientation Total: 6 hours

c) OpenGL integration

- Shader implementation (Phong, particles, materials)
- Lighting system (directional, point, spotlight)
- FPS camera configuration
- Model loading with Assimp
- Animation programming (doors, drawers, curtains, drum, steam)
- Keyboard/mouse handling and events
- Visual effects and performance adjustments Total: 45 hours

d) Documentation and debugging

- Includes structure processes, corrections and technical documentation.
- Folder organization and base parameters
- Correction of visual and integration errors
- Technical documentation preparation Total: 12 hours

2. Total time invested 30 + 6 + 45 + 12 = 93 total hours approximately This figure encompasses all development, testing, errors and learning.

3. Technical knowledge value Includes combined skills of:

- 3D Technical Artist: modeling, UVs, textures, modular structure.
- Graphics Developer: shaders, lighting, 3D transformations, animation, events, particles.

Technologies involved:

- Blender
- OpenGL + C++
- GLFW, GLEW, GLM, Assimp
- GLSL Shaders, particle system

The technical level applied is equivalent to the joint work of two professional profiles.

4. Hourly rate estimation According to market standards:

- Beginner / Student: $14–20 USD/hour
- Intermediate with self-taught knowledge (your case): $25–40 USD/hour
- Professional Freelance: $50–80 USD/hour

Using a fair value: $25 USD/hour × 93 hours = $2,325 USD

5. Adjustment by deliverable value The project consists not only of models or code, but of a complete deliverable, which includes:

- Comprehensive 3D modeling
- Texturing and UV mapping
- Full OpenGL integration
- Custom animation engines
- Programmed lighting system
- FPS camera

- Visual effects and particles
- Structured technical documentation
- Scalable modular organization

This makes it a functional prototype, comparable to the development of a video game scene. Therefore, a professional range is considered: Recommended final product value: $2,300 – $2,600 USD
Cost analysis conclusion The project represents a solid technical development that combines 3D modeling, graphics programming and documentary production. With an estimated total of 93 hours and a cost updated to 2025, the final project value ranges between $2,300 and $2,600 USD, accurately reflecting the complexity of the work performed and the specialization required to build a fully functional interactive environment.

APPLIED SOFTWARE METHODOLOGY.

For the development of this project, an incremental and collaborative methodology was adopted, aligned with the course schedule and requirements. This approach allowed progress through successive cycles of design, modeling, programming, integration and testing, ensuring that each iteration added new functionalities without compromising system stability.

1. Initial planning.

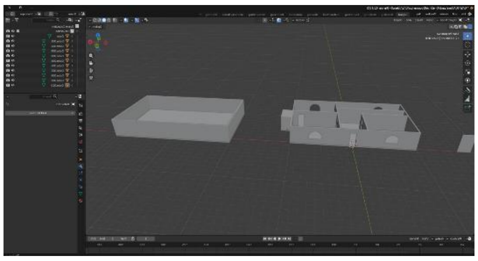
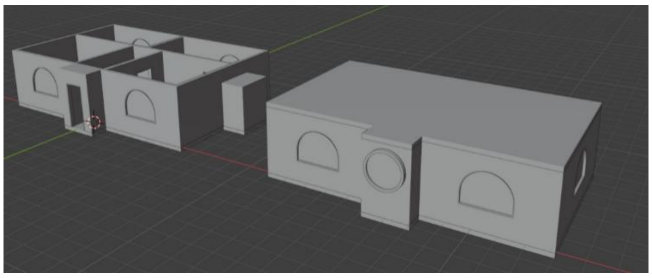
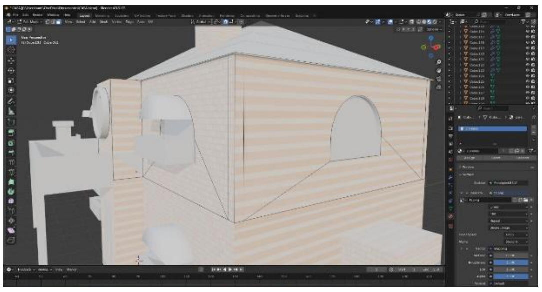
1.1 Theme selection. The theme chosen for the project was a house inspired by the Hello Kitty universe. This decision was made due to the visual characteristics of the concept: a colorful, harmonious, recognizable style with an aesthetic appeal suitable for a 3D environment. Its design allows the integration of decorative and functional elements that naturally lend themselves to three-dimensional modeling, in addition to offering opportunities to implement animations and effects in a coherent manner.
1.2 Definition of essential content. During the first work sessions, it was established that the scene should include:

- Four main rooms.
- Structural elements, furniture and decorative objects.
- Animatable objects for interaction (doors, curtains, drawers, drum, hands).
- Particle system representing steam. These components were selected because they provided visual value to the interior of the house and allowed the implementation of relevant animations within the project.

2. 3D design and modeling.

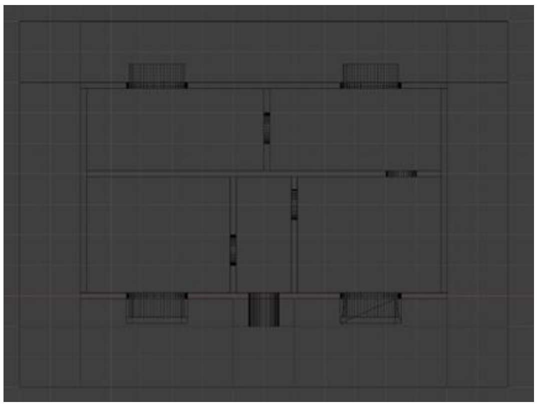
The 3D design and modeling of the project was carried out in Blender, based on a set of visual references and sketches that allowed establishing a style consistent with the colorful and distinctive aesthetics of Hello Kitty. From these references, the general structure of the house was first built and, subsequently, each of the objects that make up the rooms. The process included the creation of optimized models, coherent in scale and suitable for export in .obj format, thus facilitating their correct integration in the environment programmed with OpenGL.
2.1 Initial Facade Construction.
The main structure was modeled from an initial cube, to which extrusions were applied to form the exterior walls and entrance volumes. However, this first version presented problems related to wall thickness: having been generated through multiple extrusions, their normals and UVs became misaligned.

This situation caused textures to deform or appear incorrectly projected on surfaces when attempting to apply them for the first time.

Images. Initial structure of the house.

2.2 Implemented Solution: Structure Remodeling. These defects made evident the need to rebuild the structure from scratch to ensure a clean workflow compatible with the UV mapping system. Then the reconstruction was carried out from a base cube, this to ensure a uniform topology.



Images. Initial assembly of basic house elements.

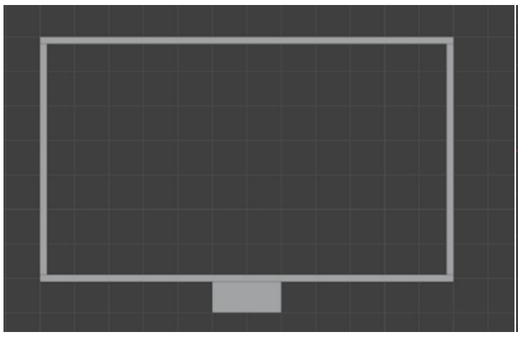
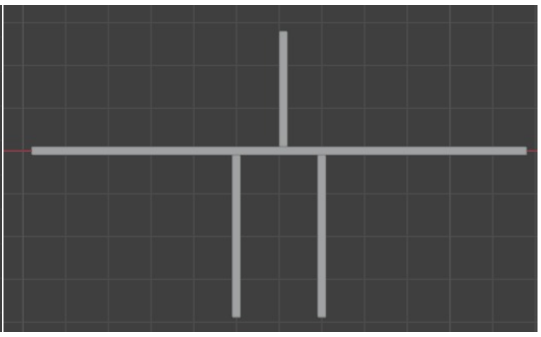
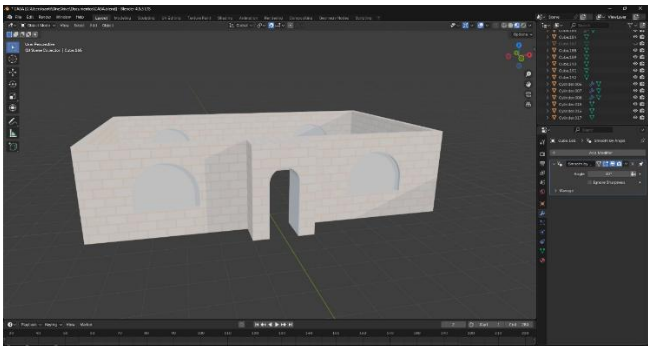
For this, the following process was followed:

1. Creation of the initial cube as the base volume of the house.
2. Entering Edit mode to work directly on the geometry.
3. Activation of the Correct Face Attributes option, which ensured that future extrusions would preserve the orientation and attributes of faces without deformations.
4. Delimitation of walls, openings and levels, using controlled cuts (Loop Cuts) and clean extrusions.
5. Construction of door and window frames using inverse extrusions, vertical cuts and proportion control from orthographic view.
6. Constant topology review, ensuring a simple polygon flow that would facilitate subsequent UV mapping. This procedure allowed building a solid structure, topologically coherent and free of the problems detected in the first versions.

2.3 Facade and Interior Walls Development.
After the reconstruction of the main structure, the fundamental elements of the dwelling were modeled again. The following were generated:

- Front facades, including the main door, from clean extrusions on the base volume.
- Curved windows, created through controlled extrusion and application of smooth bevels to obtain rounded edges and a uniform silhouette.
- Interior walls, initially defined from a general plane in top view.

To establish the room distribution, guide lines were drawn in top view, which were later converted into volumes through vertical extrusions. The main criterion for this distribution was that the four rooms of the house would maintain similar proportions in terms of useful space, although they would not necessarily share the same shape. This resulted in the following internal organization.

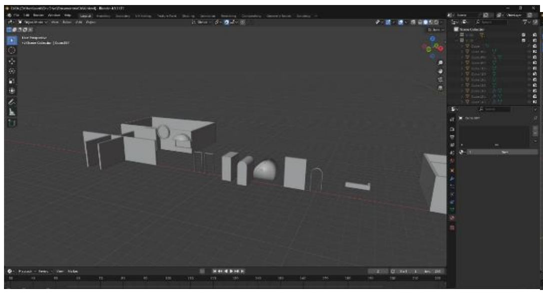
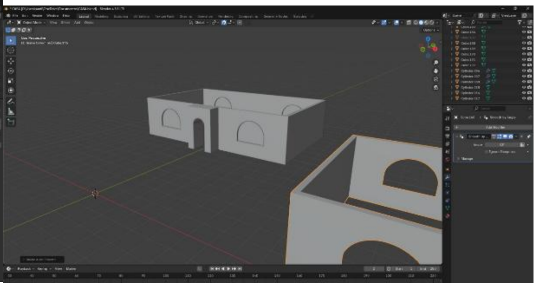



Images. View of the house after reconstruction with base textures and room distribution.

2.4 Windows and Frames Construction.

The modeling of doors and windows was carried out following a modular approach to ensure geometric precision and facilitate their subsequent texturing and animation. First, the corresponding holes were generated in the walls through controlled cuts and clean extrusions, maintaining a simple topology that would avoid distortions.

2.4.1 Windows.

The windows were built from basic cubes, adjusting their shape to the contour of the hole in the wall. For curved windows, additional cuts were added and, when necessary, a bevel to smooth the edges. Each frame was scaled and precisely aligned within its opening.

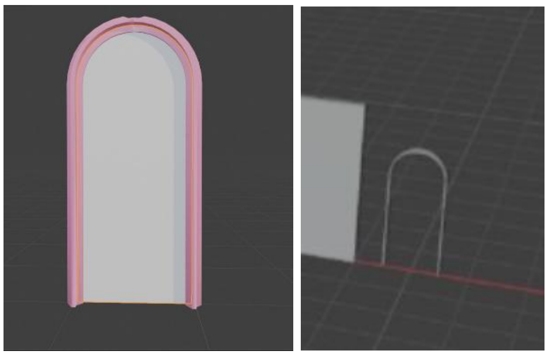


Images. View of the house after reconstruction with base textures and room distribution.

2.4.2 Door Modeling.

The base door was modeled from a cube, adjusting its dimensions until obtaining the final shape that would serve as a template. This door functioned as a master model, since all other doors in the dwelling were generated as derived duplicates (child doors). This ensured that each door maintained consistent geometry and uniform scale throughout the scene.

To integrate the door into the walls, a boolean modifier was applied to the corresponding wall, cutting the exact hole where the door would be inserted. By using a single door as the main model, it was possible to optimize the modeling process and reduce the total weight of the scene.

During preparation for export, the original door was moved to the origin to correctly align its pivot, which was placed at the hinge edge, and ensure proper animation within OpenGL. The duplicates retained this configuration, facilitating their integration in the different interior spaces.



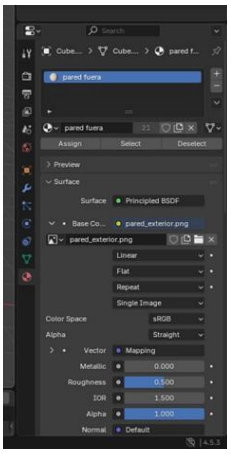
Images View of the house after reconstruction with base textures and room distribution.

2.5. Texture Application

An essential part of the project was texture assignment. The texturing process was carried out entirely in Blender, using materials based on images and UV mapping configurations adapted to the characteristics of each surface. This procedure was applied to both the main structure of the house and the additional objects later exported to OpenGL.

To apply a texture to any element of the model:

1. Selection of the object from the Viewport.
2. In the right panel, within Material Properties, a new material was added.
3. In the Base Color section, Image Texture was selected and subsequently the corresponding image.
4. Once loaded, the texture was adjusted from the UV Editing panel to ensure correct mapping.



Cube... > Cube... > pared f...

pared fuera

pared fuera 21

Assign Select Deselect

Preview

Surface

Surface: Principled BSDF

Base Co...: pared_exterior.png

pared_exterior.png

Linear

Flat

Repeat

Single Image

Color Space: sRGB

Alpha: Straight

Vector: Mapping

Metallic: 0.000

Roughness: 0.500

IOR: 1.500

Alpha: 1.000

Normal: Default

4.5.3

Images. Texture assignment from the Base Color parameter in the materials panel.

2.5.1 Mapping adjustment using nodes. Once the image was loaded, the node editor (Shading Workspace) was accessed. Blender only generates three nodes by default (Texture → Principled BSDF → Output), so the following nodes were added:

- Texture Coordinate
- Mapping

These allow controlling the orientation, repetition (tiling) and scale of the texture on each surface.

.

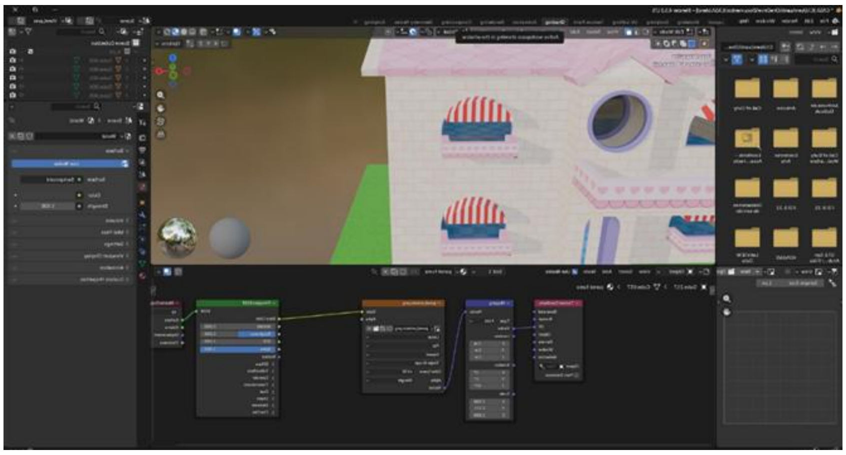


Image of Basic node configuration used to manipulate texture coordinates.

In this project, it was necessary to rotate the wall texture 90° on the Z axis to correctly align the pattern, in addition to adjusting the scale factors to repeat the image the necessary amount.

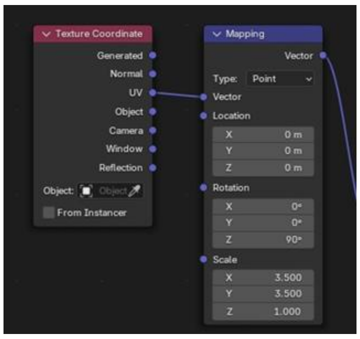

```
Texture Coordinate          Mapping
    Generated                          Vector
    Normal                  Type: Point
    UV                      Vector
    Object                  Location
    Camera                    X        0 m
    Window                    Y        0 m
    Reflection                Z        0 m
Object:  Object             Rotation
From Instancer                X        0°
                              Y        0°
                              Z       90°
                            Scale
                              X     3.500
                              Y     3.500
                              Z     1.000
```

Image of Rotation and texture repetition adjustment through the Mapping node.

2.5.2 Specific texturing of curved surfaces (awnings).
Awnings present semi-curved geometry, so their UV projection required a specific procedure.

First the object was selected, and in edit mode (A to select the entire mesh), a new UV was generated using one of these methods:

- Project from View (recommended option for curved awnings)
- Cylinder Projection, aligning the projection to the object

Subsequently, in the UV editor the UV island was adjusted:

- it was positioned over the stripe texture,
- it was scaled to straighten the lines,
- it was rotated if necessary,
- and proportions were adjusted until obtaining a clean pattern.

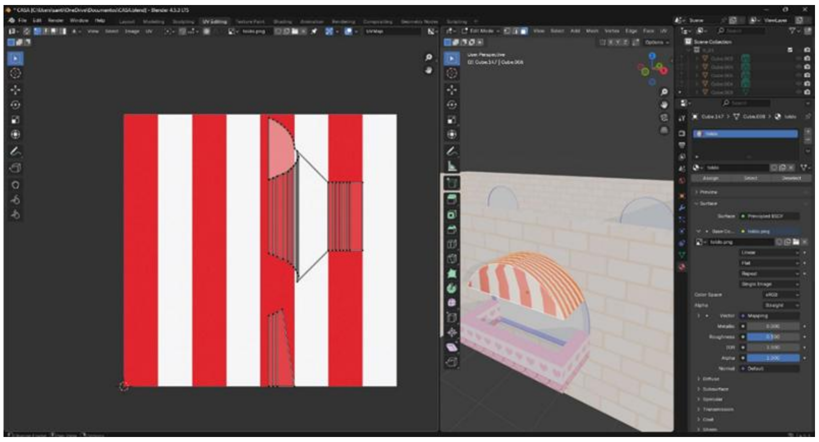


Image of Manual arrangement of the UV island over the stripe texture to correct distortions.

After applying materials, controlling tiling and rearranging UVs on each element, the house acquired a visual style consistent with the selected theme (Hello Kitty). The walls, frames, awnings and decorative elements show clean alignment and correct texture repetition without distortions.

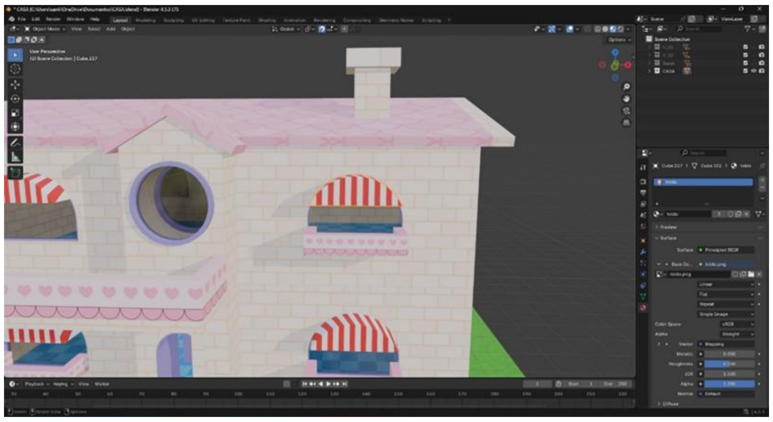
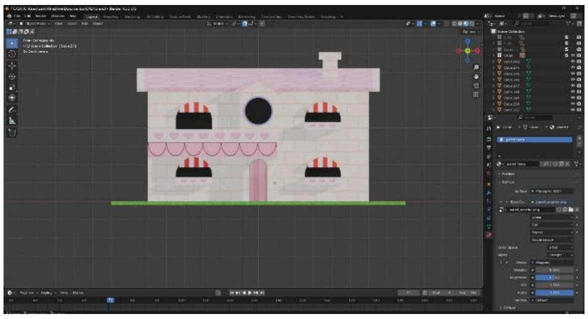
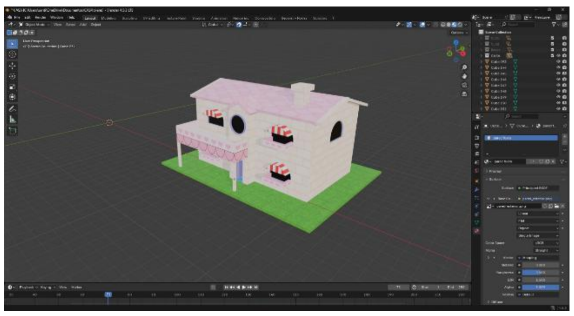


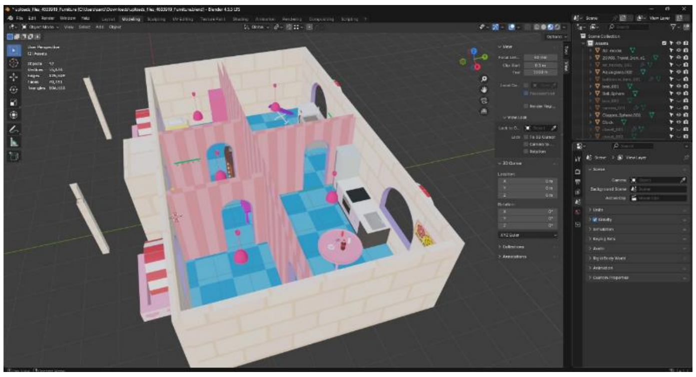
Image of Final view of the facade completely textured in Blender.

2.6 Floor Modeling and Texturing.

The house floor was obtained from the same base structure used to generate the walls. As a starting point, a cube was used, whose geometry was modified to form the walls of the first and second levels. During this process, the bottom face of the cube remained as a continuous flat surface, which allowed using it directly as the dwelling floor without the need to create an additional object.
In the following images, the final version of the house and its interiors can be seen, with all modeled and textured elements integrated into their environment.



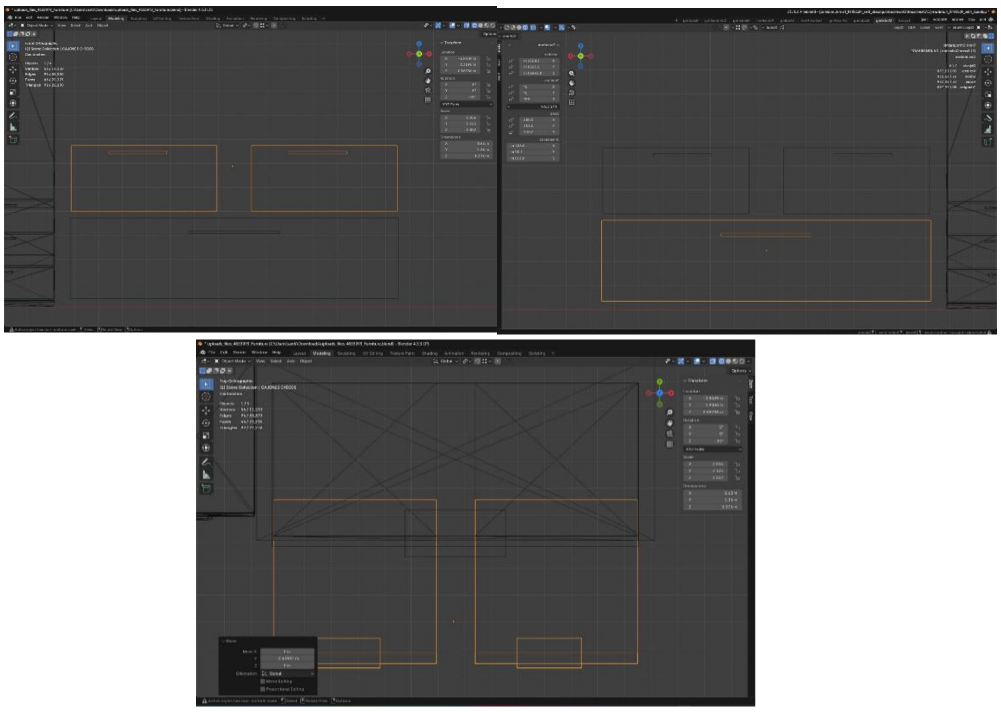



Final images of the house.

The objects that were created to furnish our facade were the following:

The cabinet was built mainly from cubes, adjusting their dimensions through basic transformations to define the main body and upper cover. The drawers were modeled as independent objects in order to allow their subsequent animation in OpenGL; for this they were extracted from the front volume using Inset and Extrude, which provided the necessary depth to differentiate each movable section.

The handles were resolved as simple pieces based on very low height cubes, placed separately to facilitate the movement of the drawers during animation. The structure was carefully organized in collections and pivots, so that each drawer could open correctly forward without deformations.



3. Export of 3D Models to .OBJ Format.

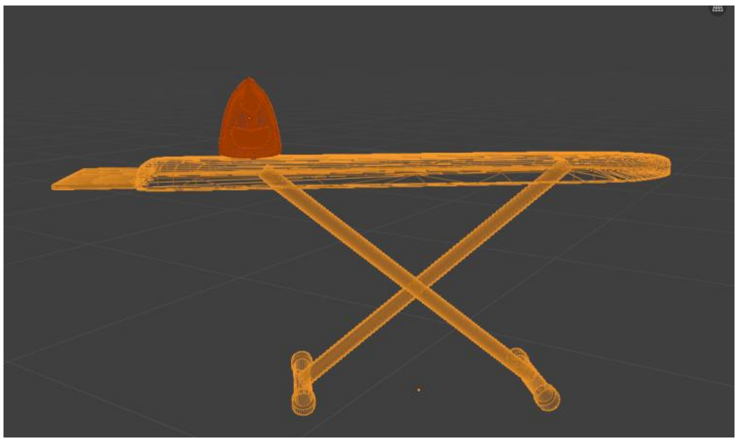
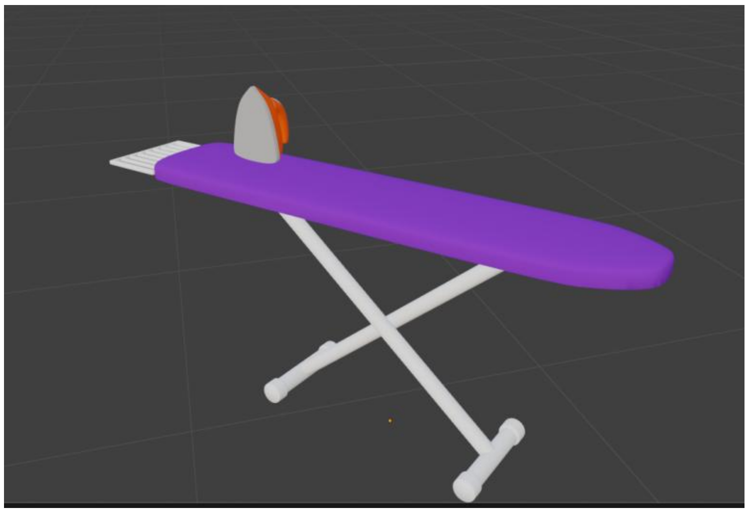
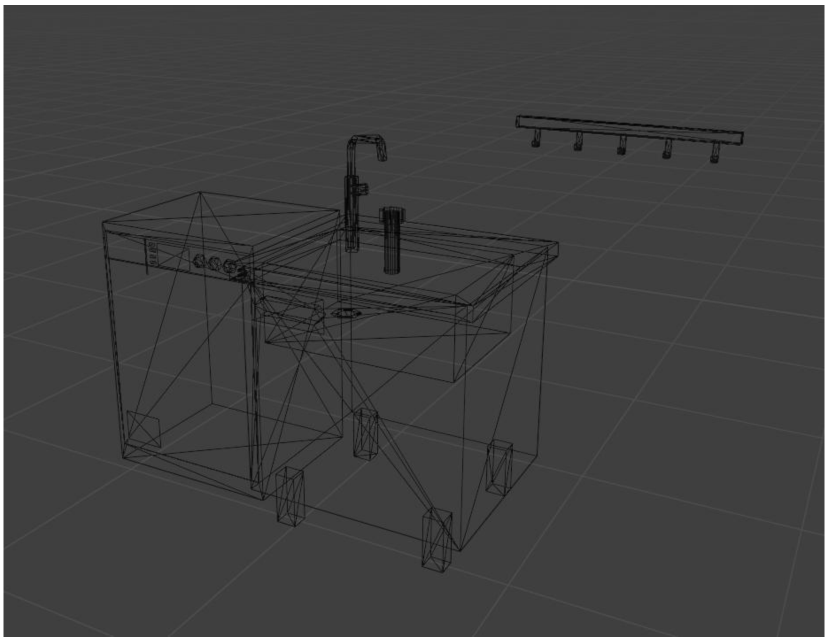



Image. Ironing board and clothes iron.



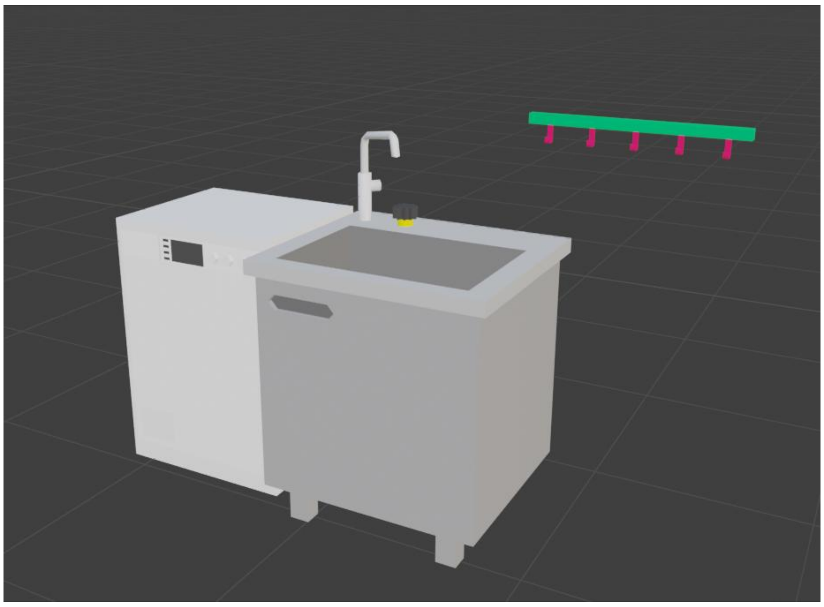


Image. Washing machine without drum, laundry room sink and wall coat rack.

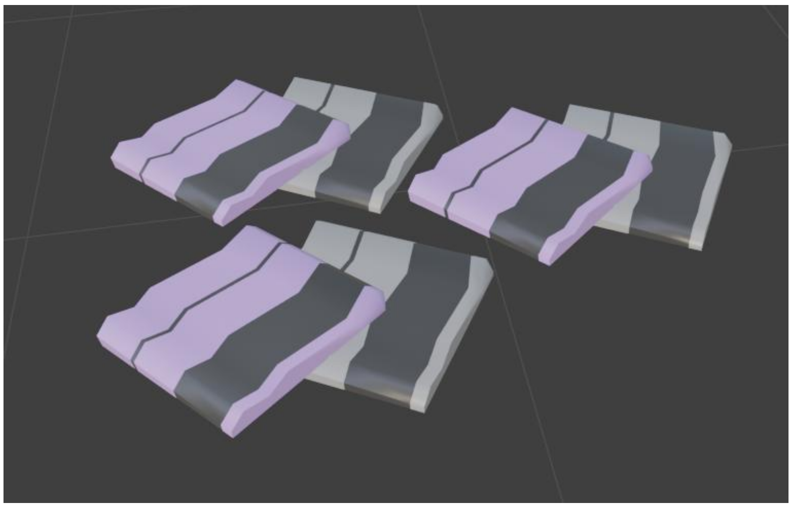
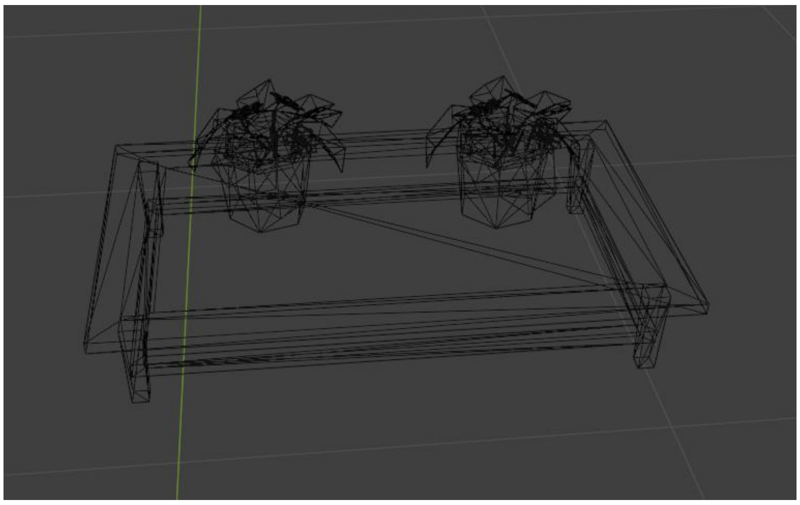


Image. Clothes thrown in the laundry room.



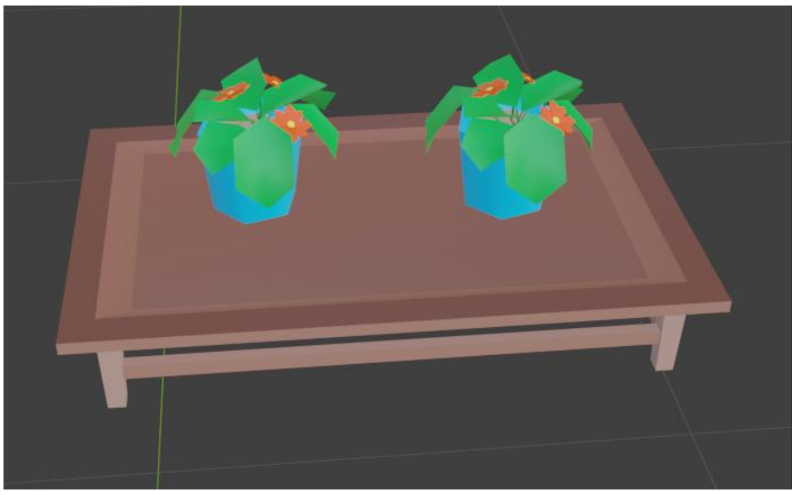


Image. Flowers on table.

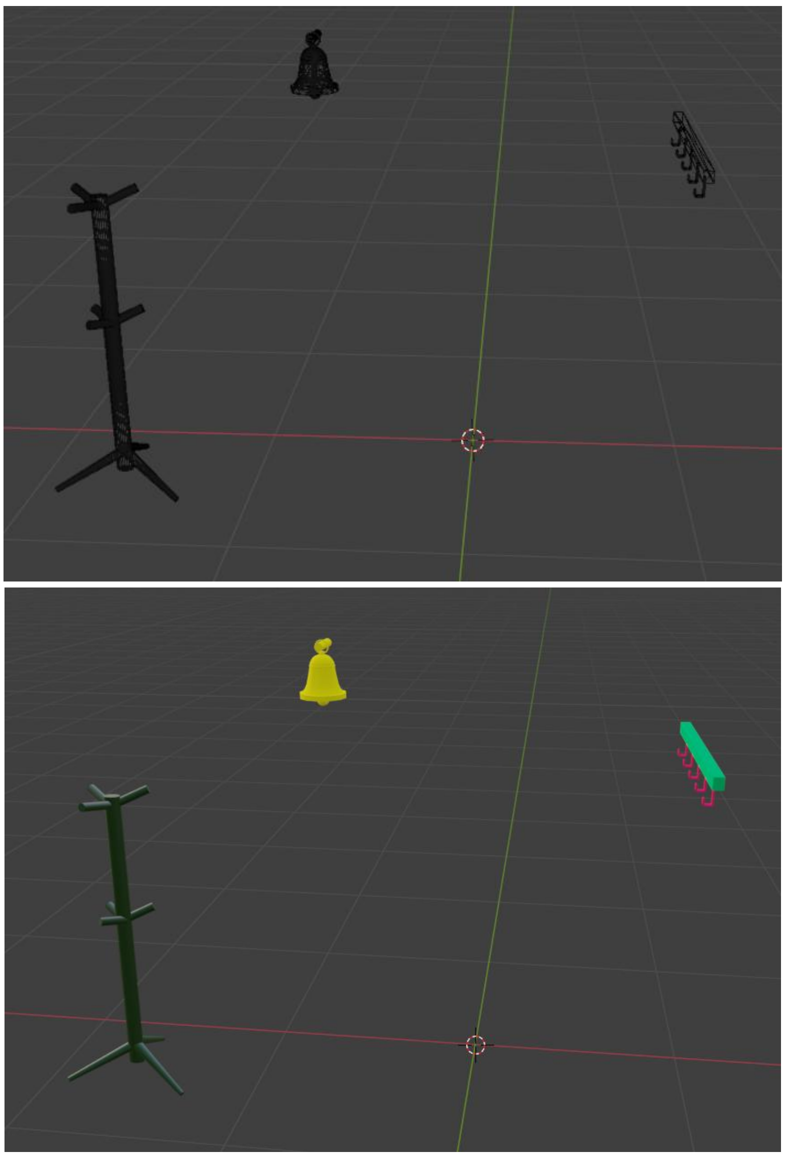


Image. Chest, wall coat rack and bell.

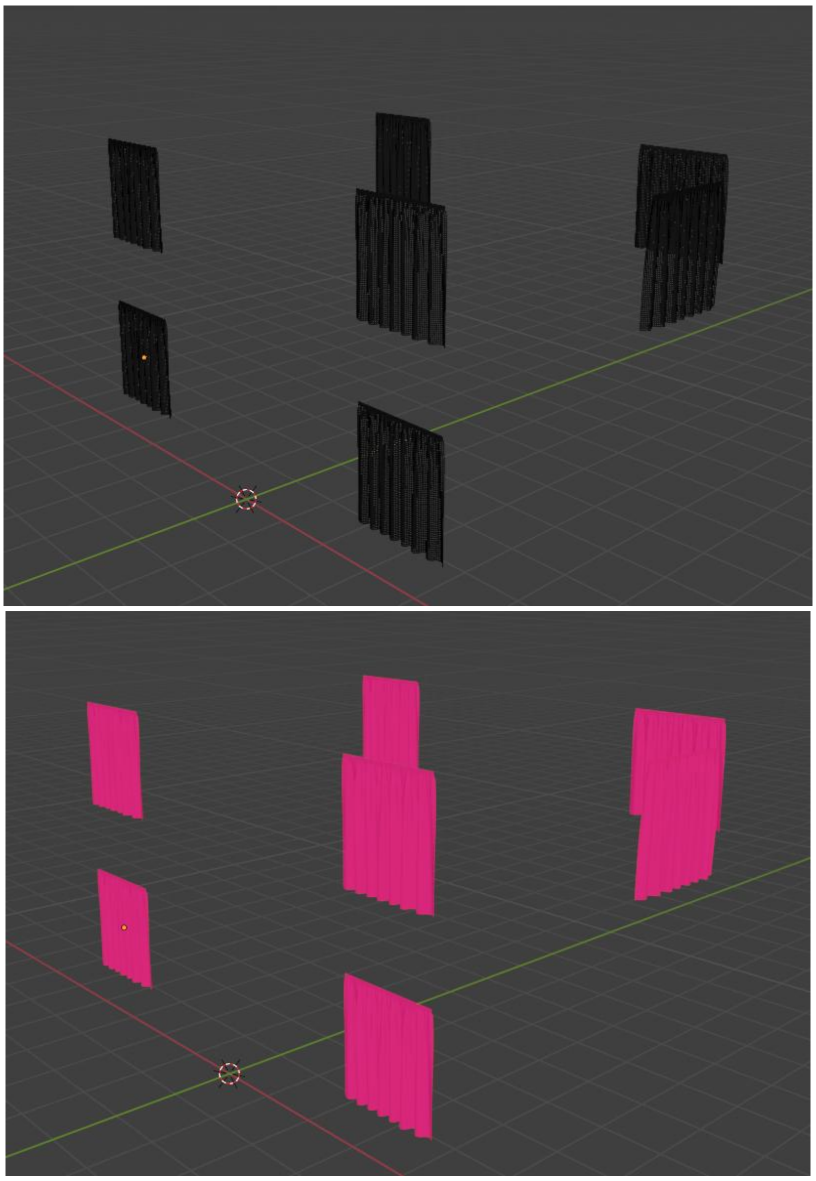
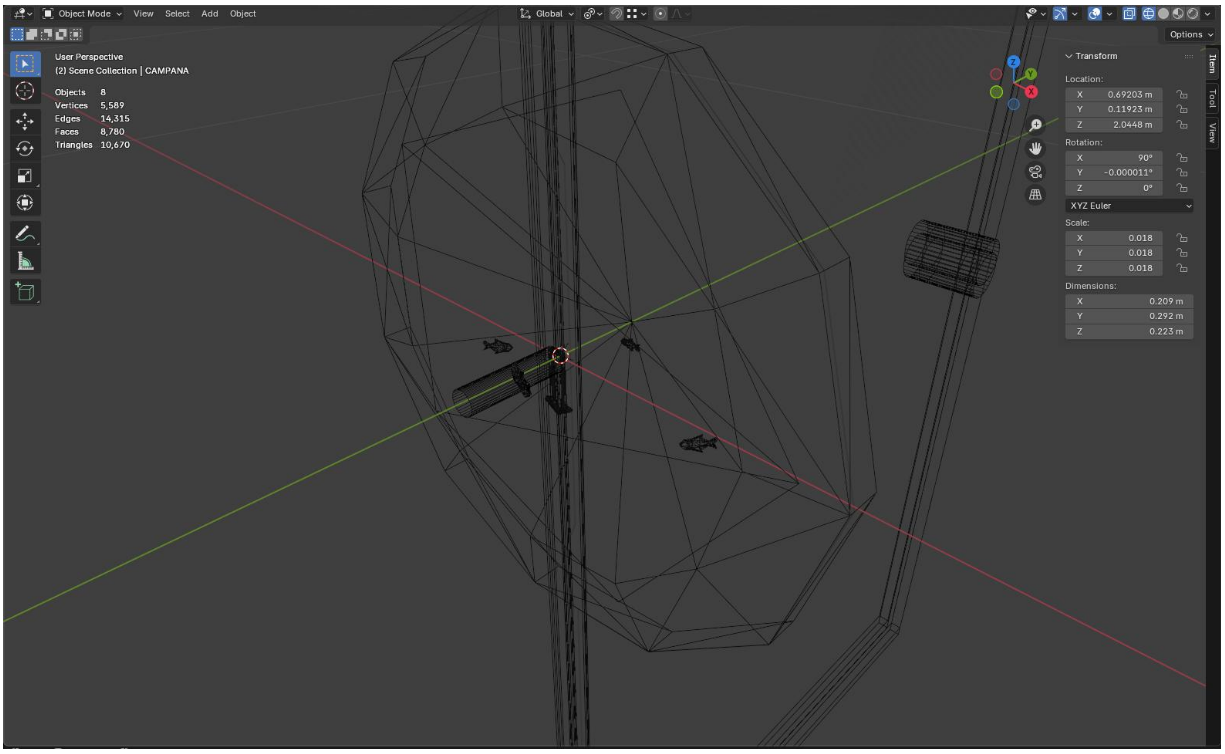


Image. Curtains.

The following objects are at the origin of our file because they had to be moved to that point to be able to move them to their original position and animate them.



User Perspective
(2) Scene Collection | CAMPANA

Objects 8
Vertices 5,589
Edges 14,315
Faces 8,780
Triangles 10,670

Transform

Location:
X 0.69203 m
Y 0.11923 m
Z 2.0448 m

Rotation:
X 90°
Y -0.000011°
Z 0°
XYZ Euler

Scale:
X 0.018
Y 0.018
Z 0.018

Dimensions:
X 0.209 m
Y 0.292 m
Z 0.223 m

Image. Washing machine drum, fishbowl fish.

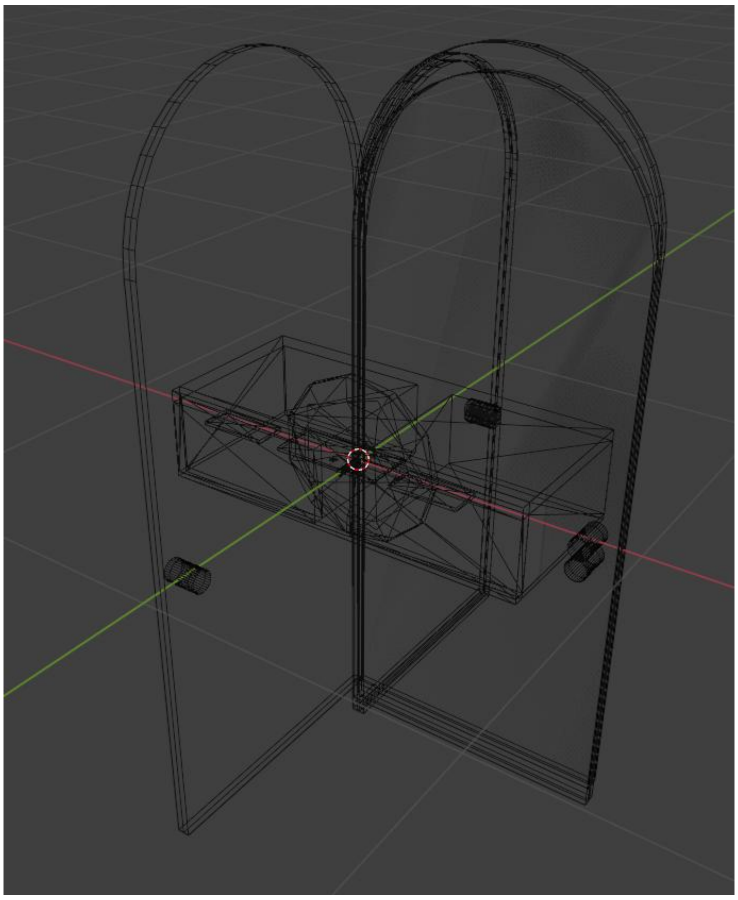


Image. House doors.

Once the modeling of the objects in Blender was finished, they were exported in .obj format, chosen for its wide compatibility with loading libraries such as Assimp and with the rendering pipeline in OpenGL.

To ensure correct import during the programming stage, the following technical points were considered:

3.2 Refined geometry: Before exporting, all meshes were reviewed to ensure they did not present inverted faces, inconsistent normals or non-manifold geometry. Unnecessary polygons were also reduced and objects were consolidated when useful to simplify the final structure.

3.3 Coherent scale: Each model was standardized in scale to maintain correct proportions when integrated into the general scene. This avoided additional transformations in OpenGL and ensured that all objects retained their expected size once loaded.

3.4 Pivots and correct orientation: Pivots were placed in strategic positions, especially in animatable objects since in these the pivot is necessary and were aligned with global axes to avoid unexpected displacements during code transformations.

3.5 Preparation for OpenGL: Prior to export, transformations were applied and histories were cleaned to ensure lightweight and consistent files.

4. Incremental software development.
Graphics environment base. The construction of the graphics environment began using C++ together with OpenGL 3.3 Core Profile, integrating essential libraries such as GLFW3, GLEW, GLM, STB_IMAGE and a model loading system based on Assimp. Once the 44 models exported from Blender were defined, the base structure of the graphics engine responsible for real-time rendering was developed.

For the management of the main window, the OpenGL context and user input capture, GLFW3 was used. Thanks to this, it was possible to establish a fixed resolution of 800×600 pixels, activate disabled cursor mode for FPS-type navigation and register multiple keys intended for animation control, lighting and camera movement.

Access to modern Core Profile extensions was achieved through GLEW, configured in experimental mode to ensure compatibility with custom shaders and advanced structures such as VBOs, VAOs and EBOs. In parallel, the GLM library provided all the mathematical functions necessary to manage spatial transformations, view and projection matrices, rotations, scalings and trigonometric operations used in both the camera system and in animations and particles.

For texture loading, STB_IMAGE was incorporated, responsible for processing PNG images with alpha channel support. This library allowed flipping textures vertically to adapt them to the OpenGL coordinate system and configure them appropriately for use in particle systems and other elements that required transparency.

Finally, the model loading system based on Assimp, implemented through the Model.h class, was responsible for importing .obj models, reading their geometries, materials and internal hierarchies. This infrastructure transformed models into structures compatible with OpenGL, even those composed of multiple meshes or materials, ensuring that their lighting properties were processed correctly.

Object Import. From the already functional graphics base in OpenGL, the integration of 3D models previously created and exported from Blender in .obj format began. This phase was carried out progressively through the Model class, adding elements one by one within the game loop to maintain control over their positioning, rotation and scaling through glm::mat4 transformation matrices. Each model was configured with its corresponding lighting system using shaders (lighting.vs/frag and lamp.vs/frag), applying specific transformations through glm::translate(), glm::rotate() and glm::scale() functions, and sending these matrices to the graphics pipeline with glUniformMatrix4fv() before being rendered with the Draw() method. Animated elements, such as doors, drawers, curtains and the washing machine drum were implemented through control variables that modify their transformations in real time within the Animation() function, while static objects are simply rendered with their base transformation matrix, thus maintaining a modular and organized structure that facilitated debugging and visual adjustments during development.

For this process, textures and materials were associated with each model through .obj files that include references to their materials and UV maps previously configured in Blender, while for special effects external textures (GOTA.png and VAPOR.png in .png format) were loaded through the stb_image library, applying texture filters with glTexParameteri() and generating mipmaps with glGenerateMipmap() to optimize rendering at different distances.

Three types of shaders were used in GLSL to handle different visual behaviors: the lightingShader implements the Phong lighting model that calculates ambient, diffuse and specular components in each fragment with configurable material properties (material.ambient, material.diffuse, material.specular, material.shininess); the lampShader renders light-emitting objects such as spotlights and the sun without lighting calculations; and the billboardShader handles water and steam particles with support for transparency through alpha blending.
Lighting was configured through a hybrid system that includes a directional light simulating the sun (controllable with U key) with adjustable properties according to its on/off state, five point lights strategically positioned in different rooms (bedroom, hallway, kitchen, laundry) with quadratic attenuation calculated through constant, linear and quadratic constants, and a deactivated spotlight that simulates a flashlight from the camera position, thus achieving dynamic and realistic lighting controllable in real time.

Operation verification. During development, periodic reviews were carried out to ensure that models loaded correctly and that the scene was displayed adequately. Failures related to lighting, interaction or animations were also immediately corrected.

Project management: In the final part of the project, the performance of the models and the entire scene was improved so that everything functioned more efficiently without losing quality. Clear documentation was also prepared explaining how the project was made and how to use it, in addition to a video showing its operation. To better organize ourselves, we used Git and GitHub, which allowed us to back up the code, keep track of changes and work in a coordinated manner. Thanks to these improvements and work organization, the project ended up being more stable, easier to understand and with better performance.

CODE DOCUMENTATION.

Below are the files that make up our project, along with a brief explanation of the function each one performs within the system.

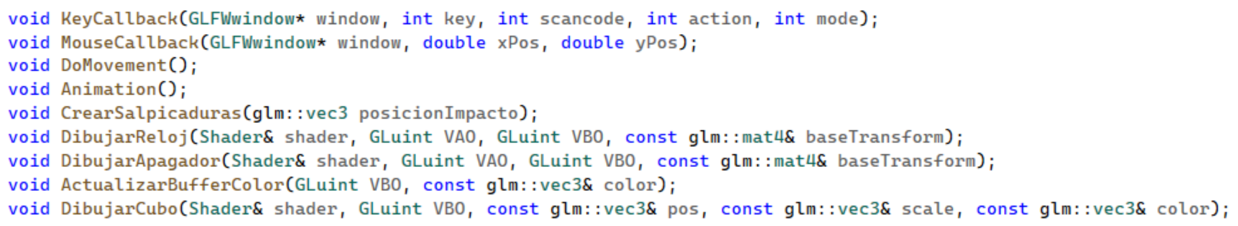
Main.cpp: This file is the main core of the program, responsible for initializing the OpenGL graphics environment, loading shaders and 3D models, managing interactive animations, implementing particle systems for visual effects, and rendering the complete scene of a domestic house with multiple rooms and animated objects. C++ is used together with OpenGL, GLFW, GLEW, GLM and the Model class for loading .obj files.

Libraries used:

- GLFW: Creates the application window and manages keyboard and mouse input.

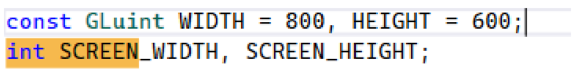
- GLEW: Allows the use of modern OpenGL functions, providing access to features such as VAOs, VBOs, programmable shaders and advanced textures.
- GLM: Provides mathematical operations for vectors and matrices.
- SOIL2 and stb_image.h: Load external textures in PNG format for effects
- Shader.h, Camera.h, Model.h: Custom files that encapsulate shader, camera and .obj model loading functions.

Function Prototypes. The program functions are grouped into five main categories. First, there are GLFW callbacks, responsible for managing user input via keyboard and mouse. Then, animation functions, which update the movement and behavior of dynamic objects in each frame. The particle system is also included, responsible for visual effects such as water and steam. In addition, rendering functions are integrated for objects created through procedural geometry, such as the clock and light switch. Finally, there is a set of auxiliary functions that handle buffers and primitive drawing.
All these functions work together and in an organized manner, allowing a modular system where each component fulfills a specific task, which facilitates code reading, maintenance and expansion.
.

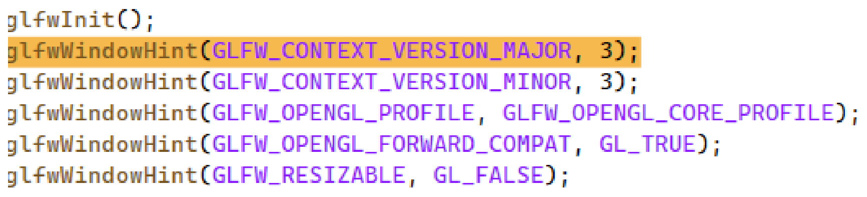


```
void KeyCallback(GLFWwindow* window, int key, int scancode, int action, int mode);
void MouseCallback(GLFWwindow* window, double xPos, double yPos);
void DoMovement();
void Animation();
void CrearSalpicaduras(glm::vec3 posicionImpacto);
void DibujarReloj(Shader& shader, GLuint VAO, GLuint VBO, const glm::mat4& baseTransform);
void DibujarApagador(Shader& shader, GLuint VAO, GLuint VBO, const glm::mat4& baseTransform);
void ActualizarBufferColor(GLuint VBO, const glm::vec3& color);
void DibujarCubo(Shader& shader, GLuint VBO, const glm::vec3& pos, const glm::vec3& scale, const glm::vec3& color);
```

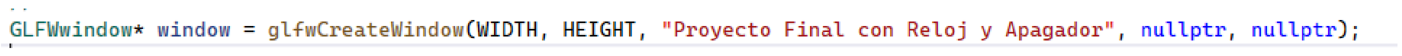
Window Dimensions, Camera and Movement. The graphics system starts by creating a fixed 800×600 pixel window, which cannot be resized to maintain a constant aspect ratio and avoid problems in projection calculations. This window works with OpenGL 3.3 Core Profile, with the forward compatibility option activated to ensure operation on macOS. In addition, the cursor is configured in disabled mode (hidden and captured), which allows moving the camera in first person continuously without being limited by the screen edges.



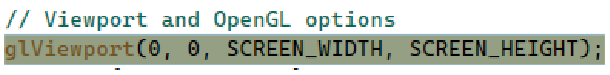
```
const GLuint WIDTH = 800, HEIGHT = 600;
int SCREEN_WIDTH, SCREEN_HEIGHT;
```



```
glfwInit();
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
glfwWindowHint(GLFW_RESIZABLE, GL_FALSE);
```

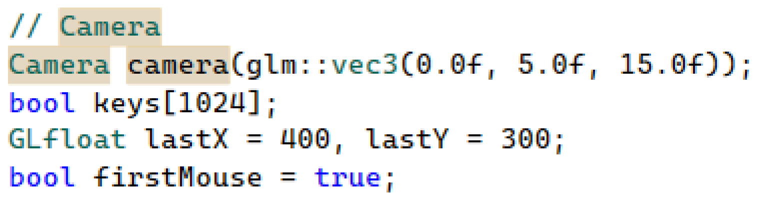


```
GLFWwindow* window = glfwCreateWindow(WIDTH, HEIGHT, "Proyecto Final con Reloj y Apagador", nullptr, nullptr);
```



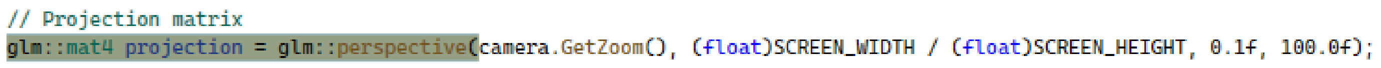
```
// Viewport and OpenGL options
glViewport(0, 0, SCREEN_WIDTH, SCREEN_HEIGHT);
```

Camera configuration.
The camera uses a first-person movement model (FPS), starting from an elevated position (y = 5) and slightly away (z = 15) to obtain an overview of the entire scene.



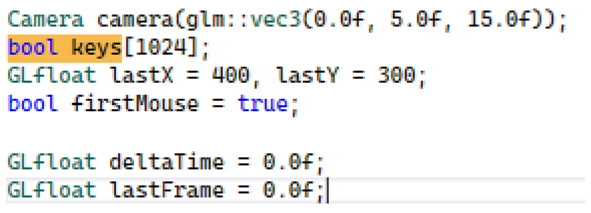
```
// Camera
Camera camera(glm::vec3(0.0f, 5.0f, 15.0f));
bool keys[1024];
GLfloat lastX = 400, lastY = 300;
bool firstMouse = true;
```

The project works with a right-hand coordinate system: Y points upward, X to the right and Z to the front (with negative values moving away from the camera).
Camera rotation is controlled with mouse movement, calculating the difference between its current position and the previous one. To avoid abrupt jumps in the first movement, the firstMouse variable is used, which stabilizes the initial cursor reading.

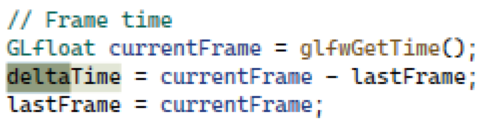


```
// Projection matrix
glm::mat4 projection = glm::perspective(camera.GetZoom(), (float)SCREEN_WIDTH / (float)SCREEN_HEIGHT, 0.1f, 100.0f);
```

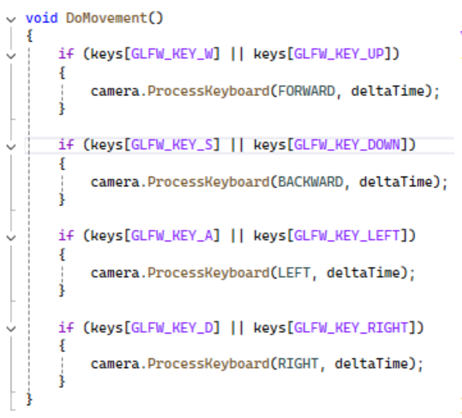
Movement system and DeltaTime usage.
Movement is managed through a double buffer system of key states that allows detecting multiple keys pressed simultaneously, essential for diagonal movement and control combinations. The DeltaTime system ensures that movement is independent of the frame rate (FPS), multiplying speeds by the time elapsed since the last frame, thus guaranteeing that an object moves the same distance in 1 second regardless of whether the game runs at 30 FPS or 144 FPS.



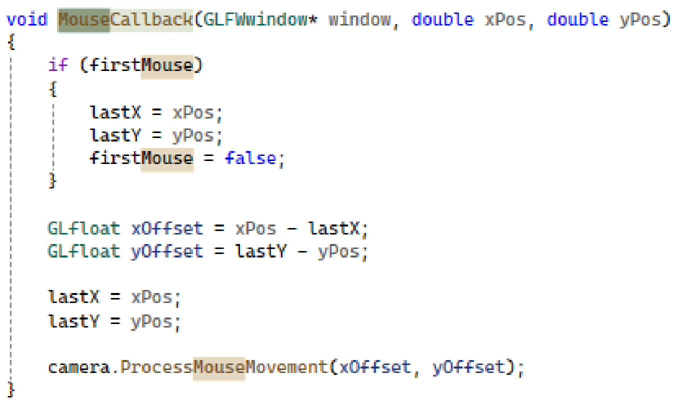
```
Camera camera(glm::vec3(0.0f, 5.0f, 15.0f));
bool keys[1024];
GLfloat lastX = 400, lastY = 300;
bool firstMouse = true;

GLfloat deltaTime = 0.0f;
GLfloat lastFrame = 0.0f;
```



```
// Frame time
GLfloat currentFrame = glfwGetTime();
deltaTime = currentFrame - lastFrame;
lastFrame = currentFrame;
```

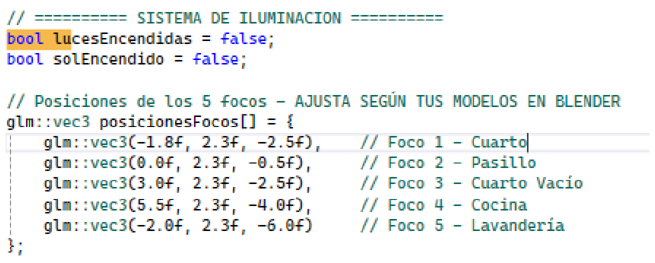


```
void DoMovement()
{
    if (keys[GLFW_KEY_W] || keys[GLFW_KEY_UP])
    {
        camera.ProcessKeyboard(FORWARD, deltaTime);
    }

    if (keys[GLFW_KEY_S] || keys[GLFW_KEY_DOWN])
    {
        camera.ProcessKeyboard(BACKWARD, deltaTime);
    }

    if (keys[GLFW_KEY_A] || keys[GLFW_KEY_LEFT])
    {
        camera.ProcessKeyboard(LEFT, deltaTime);
    }

    if (keys[GLFW_KEY_D] || keys[GLFW_KEY_RIGHT])
    {
        camera.ProcessKeyboard(RIGHT, deltaTime);
    }
}
```

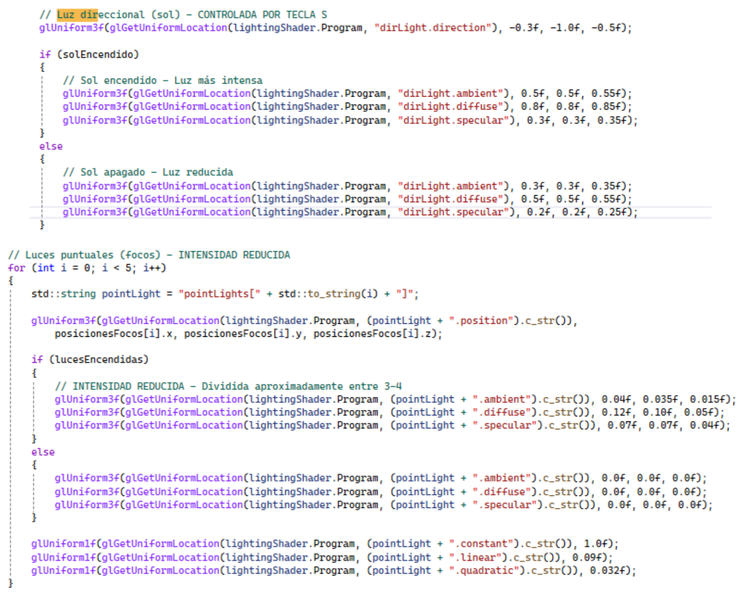


```
void MouseCallback(GLFWwindow* window, double xPos, double yPos)
{
    if (firstMouse)
    {
        lastX = xPos;
        lastY = yPos;
        firstMouse = false;
    }

    GLfloat xOffset = xPos - lastX;
    GLfloat yOffset = lastY - yPos;

    lastX = xPos;
    lastY = yPos;

    camera.ProcessMouseMovement(xOffset, yOffset);
}
```

Lighting System.
The system implements Phong lighting with three types of sources: a directional light simulating the sun (controllable with U key), five point lights on the ceiling of each room with quadratic attenuation (controllable with L key), and a deactivated spotlight reserved for camera flashlight. Each light contributes ambient, diffuse and specular components that are added to calculate the final color per fragment.



```
// ========== SISTEMA DE ILUMINACION ==========
bool lucesEncendidas = false;
bool solEncendido = false;

// Posiciones de los 5 focos - AJUSTA SEGÚN TUS MODELOS EN BLENDER
glm::vec3 posicionesFocos[] = {
    glm::vec3(-1.8f, 2.3f, -2.5f),     // Foco 1 - Cuarto
    glm::vec3(0.0f, 2.3f, -0.5f),      // Foco 2 - Pasillo
    glm::vec3(3.0f, 2.3f, -2.5f),      // Foco 3 - Cuarto Vacío
    glm::vec3(5.5f, 2.3f, -4.0f),      // Foco 4 - Cocina
    glm::vec3(-2.0f, 2.3f, -6.0f)      // Foco 5 - Lavandería
};
```

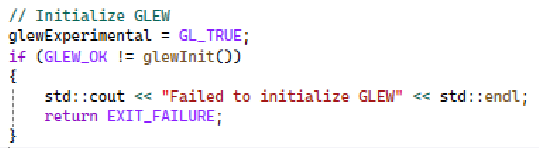


```
    // Luz direccional (sol) - CONTROLADA POR TECLA S
    glUniform3f(glGetUniformLocation(lightingShader.Program, "dirLight.direction"), -0.3f, -1.0f, -0.5f);

    if (solEncendido)
    {
        // Sol encendido - Luz más intensa
        glUniform3f(glGetUniformLocation(lightingShader.Program, "dirLight.ambient"), 0.5f, 0.5f, 0.55f);
        glUniform3f(glGetUniformLocation(lightingShader.Program, "dirLight.diffuse"), 0.8f, 0.8f, 0.85f);
        glUniform3f(glGetUniformLocation(lightingShader.Program, "dirLight.specular"), 0.3f, 0.3f, 0.35f);
    }
    else
    {
        // Sol apagado - Luz reducida
        glUniform3f(glGetUniformLocation(lightingShader.Program, "dirLight.ambient"), 0.3f, 0.3f, 0.35f);
        glUniform3f(glGetUniformLocation(lightingShader.Program, "dirLight.diffuse"), 0.5f, 0.5f, 0.55f);
        glUniform3f(glGetUniformLocation(lightingShader.Program, "dirLight.specular"), 0.2f, 0.2f, 0.25f);
    }

// Luces puntuales (focos) - INTENSIDAD REDUCIDA
for (int i = 0; i < 5; i++)
{
    std::string pointLight = "pointLights[" + std::to_string(i) + "]";

    glUniform3f(glGetUniformLocation(lightingShader.Program, (pointLight + ".position").c_str()),
        posicionesFocos[i].x, posicionesFocos[i].y, posicionesFocos[i].z);

    if (lucesEncendidas)
    {
        // INTENSIDAD REDUCIDA - Dividida aproximadamente entre 3-4
        glUniform3f(glGetUniformLocation(lightingShader.Program, (pointLight + ".ambient").c_str()), 0.04f, 0.035f, 0.015f);
        glUniform3f(glGetUniformLocation(lightingShader.Program, (pointLight + ".diffuse").c_str()), 0.12f, 0.10f, 0.05f);
        glUniform3f(glGetUniformLocation(lightingShader.Program, (pointLight + ".specular").c_str()), 0.07f, 0.07f, 0.04f);
    }
    else
    {
        glUniform3f(glGetUniformLocation(lightingShader.Program, (pointLight + ".ambient").c_str()), 0.0f, 0.0f, 0.0f);
        glUniform3f(glGetUniformLocation(lightingShader.Program, (pointLight + ".diffuse").c_str()), 0.0f, 0.0f, 0.0f);
        glUniform3f(glGetUniformLocation(lightingShader.Program, (pointLight + ".specular").c_str()), 0.0f, 0.0f, 0.0f);
    }

    glUniform1f(glGetUniformLocation(lightingShader.Program, (pointLight + ".constant").c_str()), 1.0f);
    glUniform1f(glGetUniformLocation(lightingShader.Program, (pointLight + ".linear").c_str()), 0.09f);
    glUniform1f(glGetUniformLocation(lightingShader.Program, (pointLight + ".quadratic").c_str()), 0.032f);
}
```

The following code block corresponds to the beginning of the main() function in a C++ project with OpenGL. Its purpose is to configure the graphics environment and prepare the rendering window.

- Initializes GLFW and GLEW.
- Creates an OpenGL window with an active context.
- Assigns input functions (keyboard and mouse).
- Defines the drawing area (viewport).
- Loads the shaders to use for lighting (lightingShader) and lamp (lampShader).

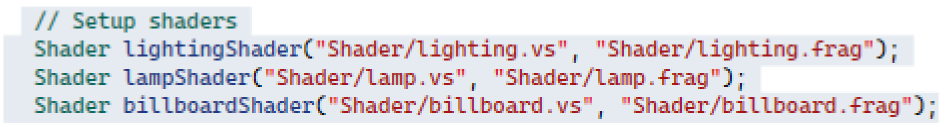
This code block is responsible for initializing the GLEW library, which allows access to modern OpenGL functions.



```
// Initialize GLEW
glewExperimental = GL_TRUE;
if (GLEW_OK != glewInit())
{
    std::cout << "Failed to initialize GLEW" << std::endl;
    return EXIT_FAILURE;
}
```

In this part of the code, three Shader objects are created, each associated with a pair of GLSL files.

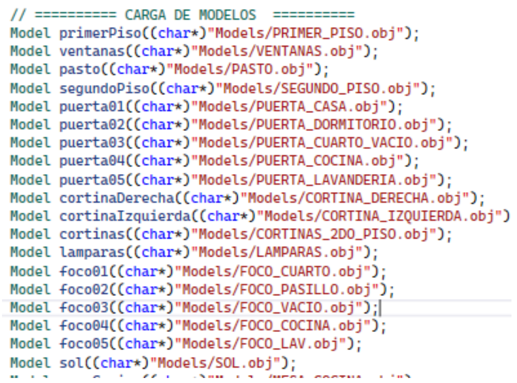
- The lightingShader manages all scene lighting (directional lights, point lights and materials).
- The lampShader is used to draw light-emitting objects without lighting calculations.
- Finally, the billboardShader controls particle rendering through the billboarding technique, ensuring they always face the camera.



```
// Setup shaders
Shader lightingShader("Shader/lighting.vs", "Shader/lighting.frag");
Shader lampShader("Shader/lamp.vs", "Shader/lamp.frag");
Shader billboardShader("Shader/billboard.vs", "Shader/billboard.frag");
```

3D model loading.

In this block all models used in the scene are initialized. Each line creates a Model object and loads a .obj file from the Models folder. When doing so, Assimp automatically imports the geometry, materials and hierarchy of the model so that it can then be rendered with OpenGL. The models include the main structure of the house (first floor, second floor, windows, grass), as well as interactive and animated elements (doors, curtains and lights). Each object is ready to be drawn within the rendering cycle and manipulated through transformations, lighting and animations.



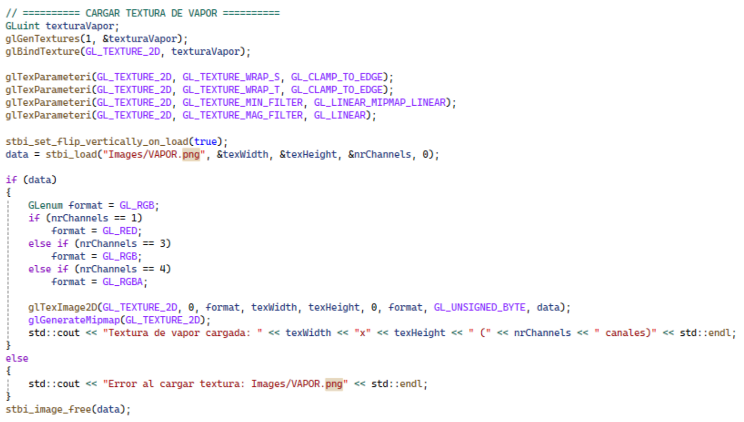
```
// ========== CARGA DE MODELOS  ==========
Model primerPiso((char*)"Models/PRIMER_PISO.obj");
Model ventanas((char*)"Models/VENTANAS.obj");
Model pasto((char*)"Models/PASTO.obj");
Model segundoPiso((char*)"Models/SEGUNDO_PISO.obj");
Model puerta01((char*)"Models/PUERTA_CASA.obj");
Model puerta02((char*)"Models/PUERTA_DORMITORIO.obj");
Model puerta03((char*)"Models/PUERTA_CUARTO_VACIO.obj");
Model puerta04((char*)"Models/PUERTA_COCINA.obj");
Model puerta05((char*)"Models/PUERTA_LAVANDERIA.obj");
Model cortinaDerecha((char*)"Models/CORTINA_DERECHA.obj");
Model cortinaIzquierda((char*)"Models/CORTINA_IZQUIERDA.obj")
Model cortinas((char*)"Models/CORTINAS_2DO_PISO.obj");
Model lamparas((char*)"Models/LAMPARAS.obj");
Model foco01((char*)"Models/FOCO_CUARTO.obj");
Model foco02((char*)"Models/FOCO_PASILLO.obj");
Model foco03((char*)"Models/FOCO_VACIO.obj");
Model foco04((char*)"Models/FOCO_COCINA.obj");
Model foco05((char*)"Models/FOCO_LAV.obj");
Model sol((char*)"Models/SOL.obj");
```

Steam Animation:

The steam animation was implemented through a particle system specifically designed to simulate the visual effect generated by the iron inside the laundry room. For this, an array of particles is initialized, where each one contains essential data for its behavior: position, velocity, size, lifetime and active state. When starting the program, all particles remain inactive and with lifetime equal to zero, ready to activate when the user starts the corresponding animation.

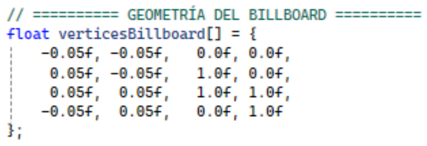
Once steam is activated, particles begin to generate progressively, moving upward with a smooth velocity and gradually reducing their size until disappearing, replicating the natural behavior of light steam. Color, transparency and attenuation parameters are controlled from the shader to achieve a smooth visual effect consistent with the environment.

The steam uses a specific texture (VAPOR.png) applied through a billboard system, which consists of a plane that always faces the camera regardless of its orientation. This allows obtaining a convincing volumetric effect without the need to use complex geometries.



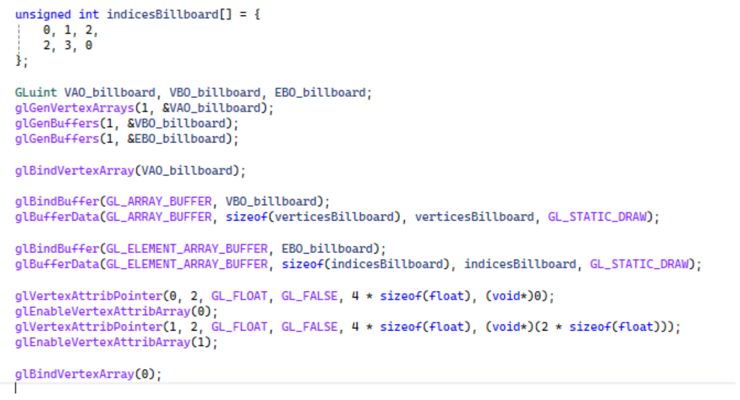
```
// ========== CARGAR TEXTURA DE VAPOR ==========
GLuint texturaVapor;
glGenTextures(1, &texturaVapor);
glBindTexture(GL_TEXTURE_2D, texturaVapor);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

stbi_set_flip_vertically_on_load(true);
data = stbi_load("Images/VAPOR.png", &texWidth, &texHeight, &nrChannels, 0);

if (data)
{
    GLenum format = GL_RGB;
    if (nrChannels == 1)
        format = GL_RED;
    else if (nrChannels == 3)
        format = GL_RGB;
    else if (nrChannels == 4)
        format = GL_RGBA;

    glTexImage2D(GL_TEXTURE_2D, 0, format, texWidth, texHeight, 0, format, GL_UNSIGNED_BYTE, data);
    glGenerateMipmap(GL_TEXTURE_2D);
    std::cout << "Textura de vapor cargada: " << texWidth << "x" << texHeight << " (" << nrChannels << " canales)" << std::endl;
}
else
{
    std::cout << "Error al cargar textura: Images/VAPOR.png" << std::endl;
}
stbi_image_free(data);
```

The billboard geometry is made up of a flat square of simple vertices, accompanied by a set of texture coordinates that define how the steam image is mapped onto each particle. For its representation, a VAO, VBO and EBO are used, which store the structure and order of triangles necessary to render each particle on screen.



```
// ========= GEOMETRÍA DEL BILLBOARD =========
float verticesBillboard[] = {
    -0.05f, -0.05f,   0.0f, 0.0f,
     0.05f, -0.05f,   1.0f, 0.0f,
     0.05f,  0.05f,   1.0f, 1.0f,
    -0.05f,  0.05f,   0.0f, 1.0f
};
```

The billboard is drawn using two triangles defined by an index array:



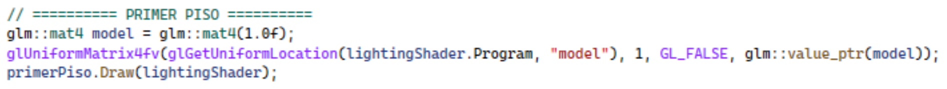
```
unsigned int indicesBillboard[] = {
    0, 1, 2,
    2, 3, 0
};

GLuint VAO_billboard, VBO_billboard, EBO_billboard;
glGenVertexArrays(1, &VAO_billboard);
glGenBuffers(1, &VBO_billboard);
glGenBuffers(1, &EBO_billboard);

glBindVertexArray(VAO_billboard);

glBindBuffer(GL_ARRAY_BUFFER, VBO_billboard);
glBufferData(GL_ARRAY_BUFFER, sizeof(verticesBillboard), verticesBillboard, GL_STATIC_DRAW);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO_billboard);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indicesBillboard), indicesBillboard, GL_STATIC_DRAW);

glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 4 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 4 * sizeof(float), (void*)(2 * sizeof(float)));
glEnableVertexAttribArray(1);

glBindVertexArray(0);
```

To render it, a VAO, a VBO and an EBO are configured, which store vertex information and triangle structure. In addition, two attributes are enabled:

- Attribute 0: vertex position
- Attribute 1: texture coordinates This process allows each particle to be drawn as a small plane with an applied texture, maintaining high performance and a visually consistent system.

Scene Rendering (Object Drawing):

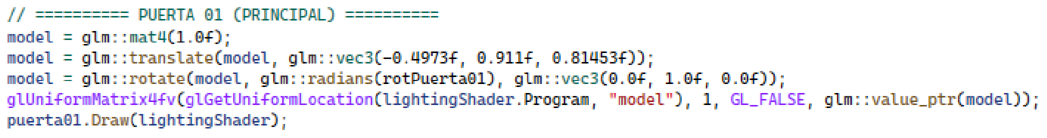
Static objects (without animation) Objects that do not move or rotate in real time, such as the first floor, grass or second floor, are drawn with an identity model matrix:



```
// ========= PRIMER PISO =========
glm::mat4 model = glm::mat4(1.0f);
glUniformMatrix4fv(glGetUniformLocation(lightingShader.Program, "model"), 1, GL_FALSE, glm::value_ptr(model));
primerPiso.Draw(lightingShader);
```

Objects animated by rotation: doors and drum.

In the case of doors, in addition to positioning them, a rotation is applied that depends on the animation variable (for example rotPuerta01):



```
// ========== PUERTA 01 (PRINCIPAL) ==========
model = glm::mat4(1.0f);
model = glm::translate(model, glm::vec3(-0.4973f, 0.911f, 0.81453f));
model = glm::rotate(model, glm::radians(rotPuerta01), glm::vec3(0.0f, 1.0f, 0.0f));
glUniformMatrix4fv(glGetUniformLocation(lightingShader.Program, "model"), 1, GL_FALSE, glm::value_ptr(model));
puerta01.Draw(lightingShader);
```

- translate places the door in its position within the house.
- rotate applies the opening/closing around the Y axis, using rotPuerta01 which is updated in the animation function.

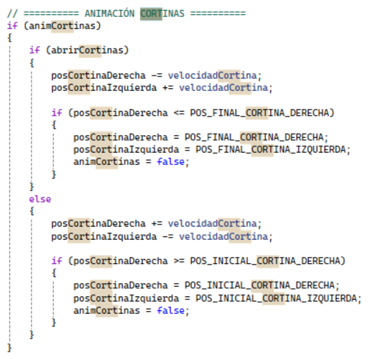
Something similar happens with the washing machine drum, only it rotates on the Z axis.

Objects animated by translation: curtains

Curtains are part of the objects animated by translation within the scene. Their movement is performed exclusively on the X axis, using two independent variables: posCortinaDerecha and posCortinaIzquierda, which control the horizontal position of each curtain separately.

To manage this behavior, a flag is used that determines if the animation is active (animCortinas) and another that defines if the movement corresponds to opening or closing (abrirCortinas). While animation is enabled, each curtain moves at a constant speed defined by velocidadCortina, moving in opposite directions to simulate opening or closing.

.



```
// ========== ANIMACIÓN CORTINAS ==========
if (animCortinas)
{
    if (abrirCortinas)
    {
        posCortinaDerecha -= velocidadCortina;
        posCortinaIzquierda += velocidadCortina;

        if (posCortinaDerecha <= POS_FINAL_CORTINA_DERECHA)
        {
            posCortinaDerecha = POS_FINAL_CORTINA_DERECHA;
            posCortinaIzquierda = POS_FINAL_CORTINA_IZQUIERDA;
            animCortinas = false;
        }
    }
    else
    {
        posCortinaDerecha += velocidadCortina;
        posCortinaIzquierda -= velocidadCortina;

        if (posCortinaDerecha >= POS_INICIAL_CORTINA_DERECHA)
        {
            posCortinaDerecha = POS_INICIAL_CORTINA_DERECHA;
            posCortinaIzquierda = POS_INICIAL_CORTINA_IZQUIERDA;
            animCortinas = false;
        }
    }
}
```

The system continuously verifies if curtains have reached their limit position. In the case of opening, both move outward until reaching POS_FINAL_CORTINA_DERECHA and POS_FINAL_CORTINA_IZQUIERDA. For closing, they move inward until reaching POS_INICIAL_CORTINA_DERECHA and POS_INICIAL_CORTINA_IZQUIERDA. Once the target point is reached, animation stops automatically.

Rest of furniture and decorative objects.

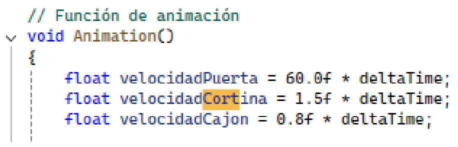
The rest of objects (bed, wardrobes, table, shelf, fishbowl, books, fridge, etc.) are almost always drawn with:

model = glm::mat4(1.0f); glUniformMatrix4fv(glGetUniformLocation(lightingShader.Program, "model"), 1, GL_FALSE, glm::value_ptr(model)); objeto.Draw(lightingShader);

Animation() Function.

The Animation() function is responsible for updating all scene animations in each frame, using deltaTime so that movements are smooth and independent of FPS. Within this function three types of elements are controlled: transformation animations (rotations and translations).

At the beginning, base speeds are calculated according to the time elapsed between frames:



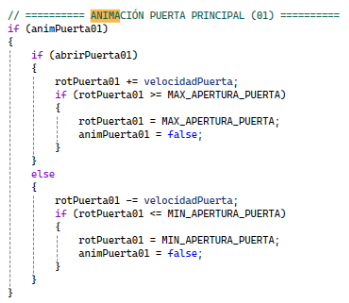
```
// Función de animación
void Animation()
{
    float velocidadPuerta = 60.0f * deltaTime;
    float velocidadCortina = 1.5f * deltaTime;
    float velocidadCajon = 0.8f * deltaTime;
```

With this, door opening, curtain displacement automatically adapt to machine performance.

Then continuous animations are updated:

- The washing machine drum (rotTambor) increases its rotation angle while its flags (animTambor) are active.
- When the value reaches 360°, it "restarts" by subtracting 360 to prevent the angle from growing indefinitely. The following sections control bidirectional animations (open/close) of doors and curtains. Each group has a flag indicating if animation is active and another indicating if movement is opening or closing:
- Main door (rotPuerta01, animPuerta01, abrirPuerta01)
- Internal doors (02 to 05) synchronized with animPuertasInternas and abrirPuertasInternas
- Curtains (posCortinaDerecha, posCortinaIzquierda, animCortinas, abrirCortinas)

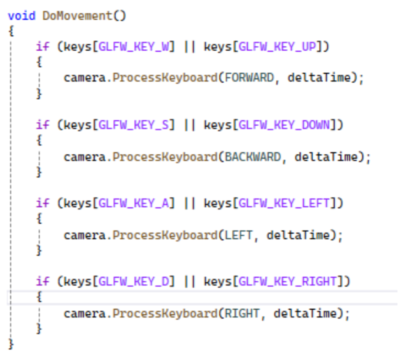
In each case, the value (rotation or position) is increased or decreased until reaching a defined limit, for example:



```cpp
// ========== ANIMACIÓN PUERTA PRINCIPAL (01) ==========
if (animPuerta01)
{
    if (abrirPuerta01)
    {
        rotPuerta01 += velocidadPuerta;
        if (rotPuerta01 >= MAX_APERTURA_PUERTA)
        {
            rotPuerta01 = MAX_APERTURA_PUERTA;
            animPuerta01 = false;
        }
    }
    else
    {
        rotPuerta01 -= velocidadPuerta;
        if (rotPuerta01 <= MIN_APERTURA_PUERTA)
        {
            rotPuerta01 = MIN_APERTURA_PUERTA;
            animPuerta01 = false;
        }
    }
}
```

Movement handling: DoMovement()

The DoMovement() function is responsible for moving the camera according to the keys the user keeps pressed. It does not directly check keyboard events, but the global keys[] array, which is updated in KeyCallback.



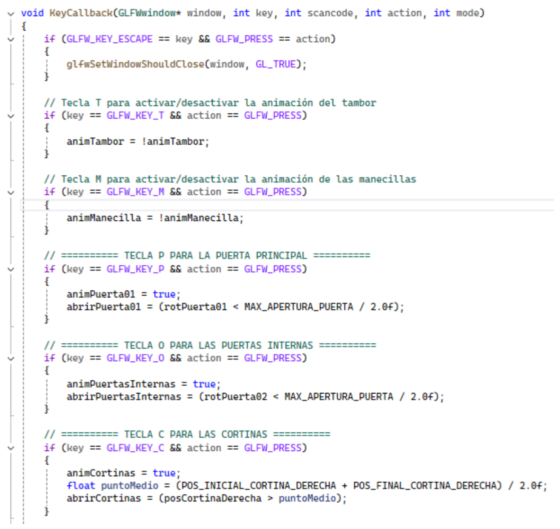
```cpp
void DoMovement()
{
    if (keys[GLFW_KEY_W] || keys[GLFW_KEY_UP])
    {
        camera.ProcessKeyboard(FORWARD, deltaTime);
    }

    if (keys[GLFW_KEY_S] || keys[GLFW_KEY_DOWN])
    {
        camera.ProcessKeyboard(BACKWARD, deltaTime);
    }

    if (keys[GLFW_KEY_A] || keys[GLFW_KEY_LEFT])
    {
        camera.ProcessKeyboard(LEFT, deltaTime);
    }

    if (keys[GLFW_KEY_D] || keys[GLFW_KEY_RIGHT])
    {
        camera.ProcessKeyboard(RIGHT, deltaTime);
    }
}
```

This way, the camera can move forward, backward and laterally using WASD or arrows, and the use of deltaTime makes speed consistent regardless of FPS.

Keyboard handling and toggles: KeyCallback()

KeyCallback() is the function registered in GLFW to process each keyboard event. Here two things are resolved:

1. Immediate actions (for example, closing the window with ESC).
2. Change of boolean flags that activate or deactivate animations or effects, which are then processed in Animation().



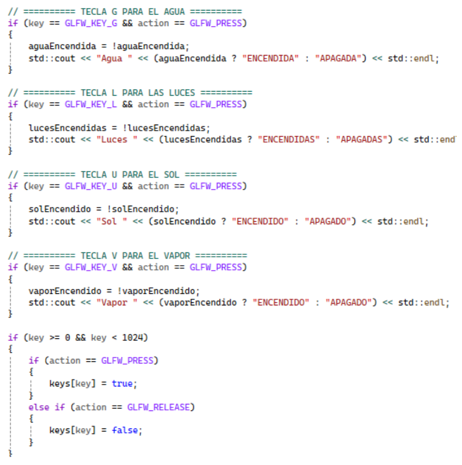
```cpp
void KeyCallback(GLFWwindow* window, int key, int scancode, int action, int mode)
{
    if (GLFW_KEY_ESCAPE == key && GLFW_PRESS == action)
    {
        glfwSetWindowShouldClose(window, GL_TRUE);
    }

    // Tecla T para activar/desactivar la animación del tambor
    if (key == GLFW_KEY_T && action == GLFW_PRESS)
    {
        animTambor = !animTambor;
    }

    // Tecla M para activar/desactivar la animación de las manecillas
    if (key == GLFW_KEY_M && action == GLFW_PRESS)
    {
        animManecilla = !animManecilla;
    }

    // ========== TECLA P PARA LA PUERTA PRINCIPAL ==========
    if (key == GLFW_KEY_P && action == GLFW_PRESS)
    {
        animPuerta01 = true;
        abrirPuerta01 = (rotPuerta01 < MAX_APERTURA_PUERTA / 2.0f);
    }

    // ========== TECLA O PARA LAS PUERTAS INTERNAS ==========
    if (key == GLFW_KEY_O && action == GLFW_PRESS)
    {
        animPuertasInternas = true;
        abrirPuertasInternas = (rotPuerta02 < MAX_APERTURA_PUERTA / 2.0f);
    }

    // ========== TECLA C PARA LAS CORTINAS ==========
    if (key == GLFW_KEY_C && action == GLFW_PRESS)
    {
        animCortinas = true;
        float puntoMedio = (POS_INICIAL_CORTINA_DERECHA + POS_FINAL_CORTINA_DERECHA) / 2.0f;
        abrirCortinas = (posCortinaDerecha > puntoMedio);
    }
```

For animations, the general pattern is:

- Detect the key and the GLFW_PRESS event.
- Change a boolean variable or indicate if something is going to "open" or "close".

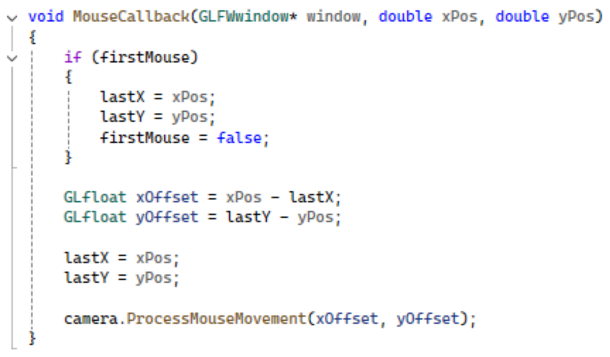
What changes is that the washing machine drum uses a simple toggle, while doors and curtains use a smart toggle, which decides whether to open or close according to their current position.

The L, U, V keys activate or deactivate complete systems (water, lights, sun, steam) simply by inverting a boolean:



```cpp
// ========== TECLA G PARA EL AGUA ==========
if (key == GLFW_KEY_G && action == GLFW_PRESS)
{
    aguaEncendida = !aguaEncendida;
    std::cout << "Agua " << (aguaEncendida ? "ENCENDIDA" : "APAGADA") << std::endl;
}

// ========== TECLA L PARA LAS LUCES ==========
if (key == GLFW_KEY_L && action == GLFW_PRESS)
{
    lucesEncendidas = !lucesEncendidas;
    std::cout << "Luces " << (lucesEncendidas ? "ENCENDIDAS" : "APAGADAS") << std::end
}

// ========== TECLA U PARA EL SOL ==========
if (key == GLFW_KEY_U && action == GLFW_PRESS)
{
    solEncendido = !solEncendido;
    std::cout << "Sol " << (solEncendido ? "ENCENDIDO" : "APAGADO") << std::endl;
}

// ========== TECLA V PARA EL VAPOR ==========
if (key == GLFW_KEY_V && action == GLFW_PRESS)
{
    vaporEncendido = !vaporEncendido;
    std::cout << "Vapor " << (vaporEncendido ? "ENCENDIDO" : "APAGADO") << std::endl;
}

if (key >= 0 && key < 1024)
{
    if (action == GLFW_PRESS)
    {
        keys[key] = true;
    }
    else if (action == GLFW_RELEASE)
    {
        keys[key] = false;
    }
}
```

View control with mouse:
MouseCallback() MouseCallback() controls FPS-type camera rotation using mouse position:

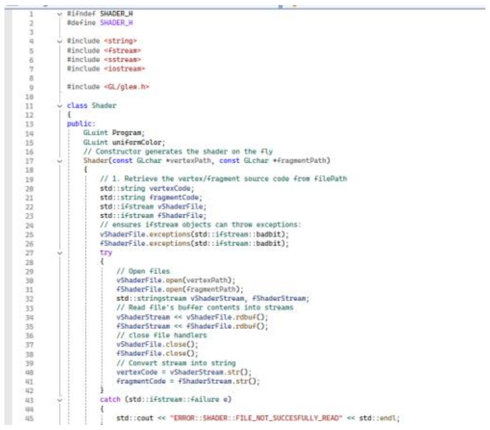


```cpp
void MouseCallback(GLFWwindow* window, double xPos, double yPos)
{
    if (firstMouse)
    {
        lastX = xPos;
        lastY = yPos;
        firstMouse = false;
    }

    GLfloat xOffset = xPos - lastX;
    GLfloat yOffset = lastY - yPos;

    lastX = xPos;
    lastY = yPos;

    camera.ProcessMouseMovement(xOffset, yOffset);
}
```

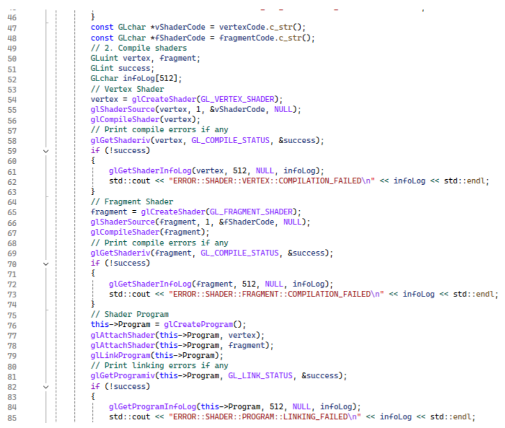
- firstMouse avoids an abrupt "jump" in the first movement.
- Offsets in X and Y are calculated with respect to the previous position.
- These offsets are sent to camera.ProcessMouseMovement(), which updates yaw and pitch.

Shader.h

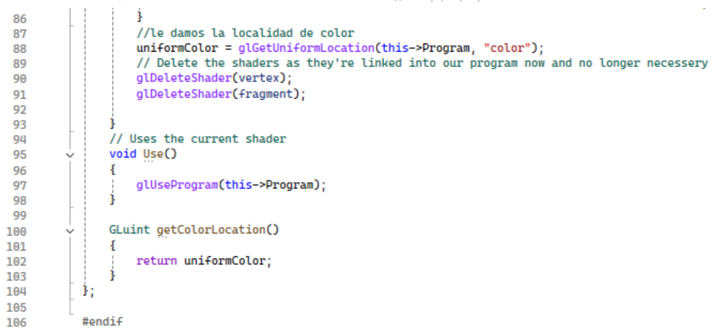
The Shader class is responsible for loading, compiling and activating shading programs necessary for rendering in OpenGL. When a Shader object is created, the system reads from file the vertex shader and fragment shader code, compiles them verifying possible errors and then links them within a single program executable by the GPU. Once built, the shader provides functions to activate it during drawing (Use()) and to obtain locations of uniform variables, such as the color parameter, which allows sending values from C++ to the shader. In summary, this class automates the entire technical process of shader management, facilitating its use within the graphics engine..



```cpp
#ifndef SHADER_H
#define SHADER_H

#include <string>
#include <fstream>
#include <sstream>
#include <iostream>

#include <GL/glew.h>

class Shader
{
public:
    GLuint Program;
    GLuint uniformColor;
    // Constructor generates the shader on the fly
    Shader(const GLchar *vertexPath, const GLchar *fragmentPath)
    {
        // 1. Retrieve the vertex/fragment source code from filePath
        std::string vertexCode;
        std::string fragmentCode;
        std::ifstream vShaderFile;
        std::ifstream fShaderFile;
        // ensures ifstream objects can throw exceptions:
        vShaderFile.exceptions(std::ifstream::badbit);
        fShaderFile.exceptions(std::ifstream::badbit);
        try
        {
            // Open files
            vShaderFile.open(vertexPath);
            fShaderFile.open(fragmentPath);
            std::stringstream vShaderStream, fShaderStream;
            // Read file's buffer contents into streams
            vShaderStream << vShaderFile.rdbuf();
            fShaderStream << fShaderFile.rdbuf();
            // close file handlers
            vShaderFile.close();
            fShaderFile.close();
            // Convert stream into string
            vertexCode = vShaderStream.str();
            fragmentCode = fShaderStream.str();
        }
        catch (std::ifstream::failure e)
        {
            std::cout << "ERROR::SHADER::FILE_NOT_SUCCESFULLY_READ" << std::endl;
```



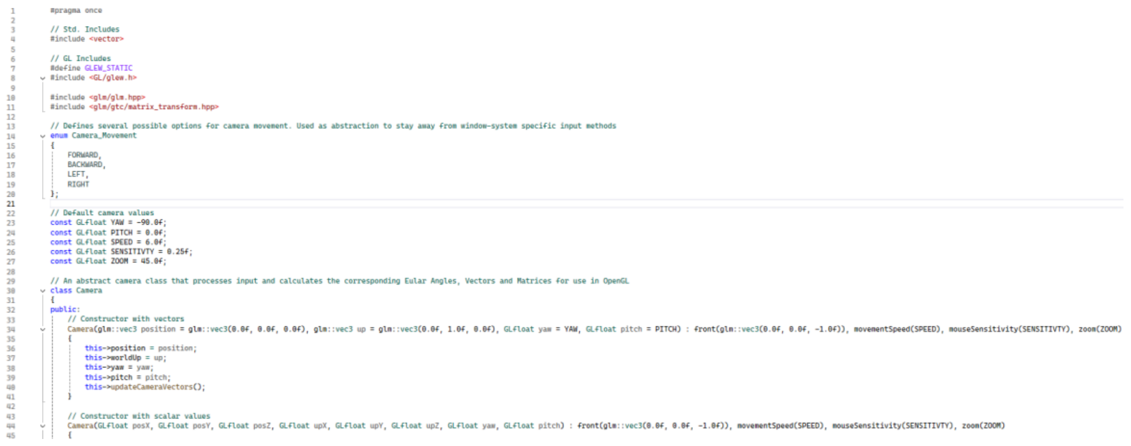
```cpp
        }
        const GLchar *vShaderCode = vertexCode.c_str();
        const GLchar *fShaderCode = fragmentCode.c_str();
        // 2. Compile shaders
        GLuint vertex, fragment;
        GLint success;
        GLchar infoLog[512];
        // Vertex Shader
        vertex = glCreateShader(GL_VERTEX_SHADER);
        glShaderSource(vertex, 1, &vShaderCode, NULL);
        glCompileShader(vertex);
        // Print compile errors if any
        glGetShaderiv(vertex, GL_COMPILE_STATUS, &success);
        if (!success)
        {
            glGetShaderInfoLog(vertex, 512, NULL, infoLog);
            std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog << std::endl;
        }
        // Fragment Shader
        fragment = glCreateShader(GL_FRAGMENT_SHADER);
        glShaderSource(fragment, 1, &fShaderCode, NULL);
        glCompileShader(fragment);
        // Print compile errors if any
        glGetShaderiv(fragment, GL_COMPILE_STATUS, &success);
        if (!success)
        {
            glGetShaderInfoLog(fragment, 512, NULL, infoLog);
            std::cout << "ERROR::SHADER::FRAGMENT::COMPILATION_FAILED\n" << infoLog << std::endl;
        }
        // Shader Program
        this->Program = glCreateProgram();
        glAttachShader(this->Program, vertex);
        glAttachShader(this->Program, fragment);
        glLinkProgram(this->Program);
        // Print linking errors if any
        glGetProgramiv(this->Program, GL_LINK_STATUS, &success);
        if (!success)
        {
            glGetProgramInfoLog(this->Program, 512, NULL, infoLog);
            std::cout << "ERROR::SHADER::PROGRAM::LINKING_FAILED\n" << infoLog << std::endl;
```



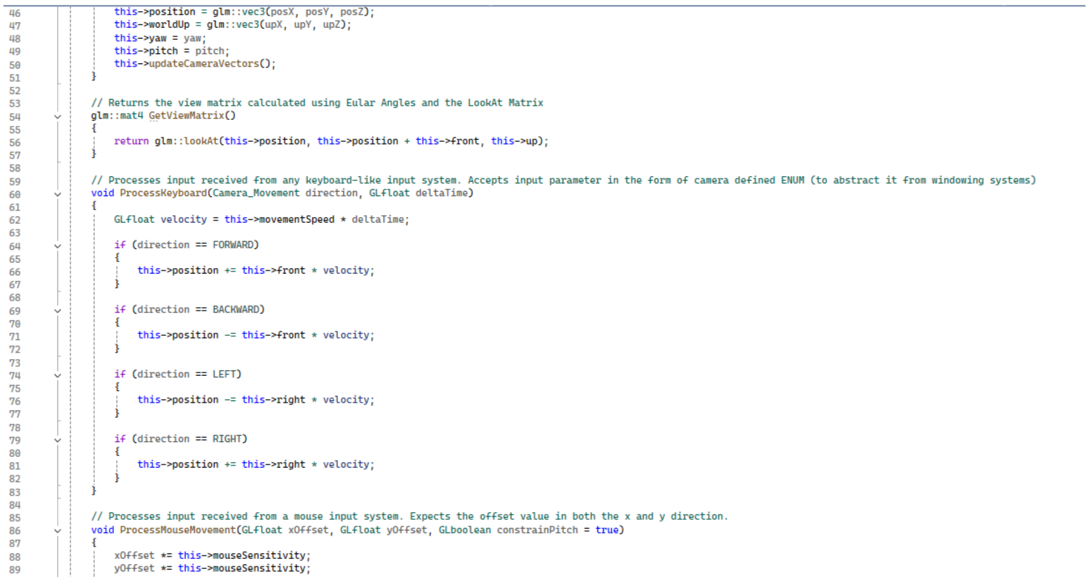
```cpp
        }
        //le damos la localidad de color
        uniformColor = glGetUniformLocation(this->Program, "color");
        // Delete the shaders as they're linked into our program now and no longer necessery
        glDeleteShader(vertex);
        glDeleteShader(fragment);

    }
    // Uses the current shader
    void Use()
    {
        glUseProgram(this->Program);
    }

    GLuint getColorLocation()
    {
        return uniformColor;
    }
};

#endif
```

Camara.h

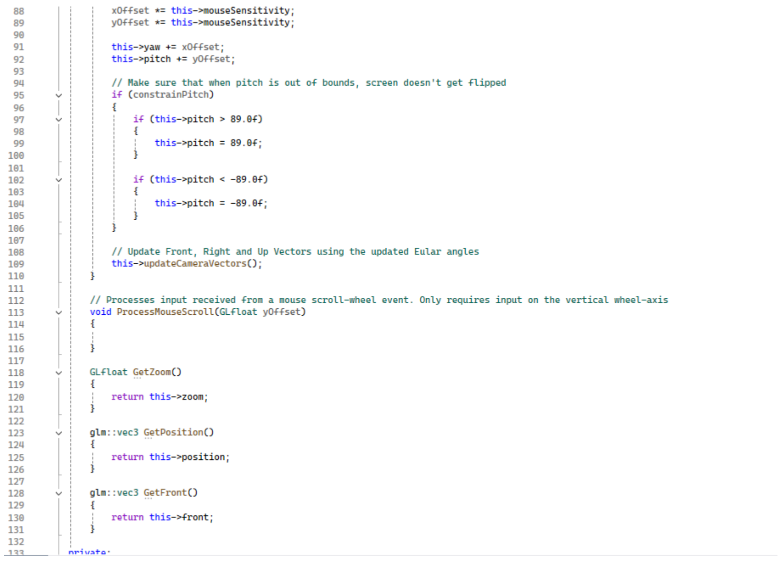
The Camera class implements the first-person camera we use in the scene. This class stores the camera position, where it is looking (front vector), as well as the up and right vectors that define its orientation in space. From this data, the GetViewMatrix() function generates the view matrix using glm::lookAt, which is sent to the shader to render everything from the user's perspective. The camera is updated with two types of input: with ProcessKeyboard() movement is processed in the FORWARD, BACKWARD, LEFT, RIGHT directions, moving the position according to speed and deltaTime; and with ProcessMouseMovement() the yaw and pitch angles are updated from mouse movement, limiting pitch to prevent the camera from "flipping" completely. Each time these angles change, the private updateCameraVectors() function recalculates the front, right and up vectors, keeping the camera orientation coherent during navigation.



```cpp
#pragma once

// Std. Includes
#include <vector>

// GL Includes
#define GLEW_STATIC
#include <GL/glew.h>

#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>

// Defines several possible options for camera movement. Used as abstraction to stay away from window-system specific input methods
enum Camera_Movement
{
    FORWARD,
    BACKWARD,
    LEFT,
    RIGHT
};

// Default camera values
const GLfloat YAW = -90.0f;
const GLfloat PITCH = 0.0f;
const GLfloat SPEED = 6.0f;
const GLfloat SENSITIVTY = 0.25f;
const GLfloat ZOOM = 45.0f;

// An abstract camera class that processes input and calculates the corresponding Eular Angles, Vectors and Matrices for use in OpenGL
class Camera
{
public:
    // Constructor with vectors
    Camera(glm::vec3 position = glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3 up = glm::vec3(0.0f, 1.0f, 0.0f), GLfloat yaw = YAW, GLfloat pitch = PITCH) : front(glm::vec3(0.0f, 0.0f, -1.0f)), movementSpeed(SPEED), mouseSensitivity(SENSITIVTY), zoom(ZOOM)
    {
        this->position = position;
        this->worldUp = up;
        this->yaw = yaw;
        this->pitch = pitch;
        this->updateCameraVectors();
    }

    // Constructor with scalar values
    Camera(GLfloat posX, GLfloat posY, GLfloat posZ, GLfloat upX, GLfloat upY, GLfloat upZ, GLfloat yaw, GLfloat pitch) : front(glm::vec3(0.0f, 0.0f, -1.0f)), movementSpeed(SPEED), mouseSensitivity(SENSITIVTY), zoom(ZOOM)
    {
```



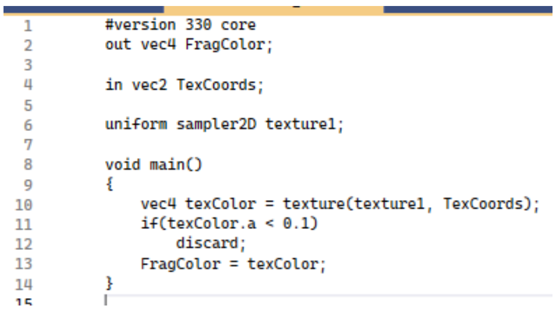
```
            this->position = glm::vec3(posX, posY, posZ);
            this->worldUp = glm::vec3(upX, upY, upZ);
            this->yaw = yaw;
            this->pitch = pitch;
            this->updateCameraVectors();
        }

        // Returns the view matrix calculated using Eular Angles and the LookAt Matrix
        glm::mat4 GetViewMatrix()
        {
            return glm::lookAt(this->position, this->position + this->front, this->up);
        }

        // Processes input received from any keyboard-like input system. Accepts input parameter in the form of camera defined ENUM (to abstract it from windowing systems)
        void ProcessKeyboard(Camera_Movement direction, GLfloat deltaTime)
        {
            GLfloat velocity = this->movementSpeed * deltaTime;

            if (direction == FORWARD)
            {
                this->position += this->front * velocity;
            }

            if (direction == BACKWARD)
            {
                this->position -= this->front * velocity;
            }

            if (direction == LEFT)
            {
                this->position -= this->right * velocity;
            }

            if (direction == RIGHT)
            {
                this->position += this->right * velocity;
            }
        }

        // Processes input received from a mouse input system. Expects the offset value in both the x and y direction.
        void ProcessMouseMovement(GLfloat xOffset, GLfloat yOffset, GLboolean constrainPitch = true)
        {
            xOffset *= this->mouseSensitivity;
            yOffset *= this->mouseSensitivity;
```



```
            xOffset *= this->mouseSensitivity;
            yOffset *= this->mouseSensitivity;

            this->yaw += xOffset;
            this->pitch += yOffset;

            // Make sure that when pitch is out of bounds, screen doesn't get flipped
            if (constrainPitch)
            {
                if (this->pitch > 89.0f)
                {
                    this->pitch = 89.0f;
                }

                if (this->pitch < -89.0f)
                {
                    this->pitch = -89.0f;
                }
            }

            // Update Front, Right and Up Vectors using the updated Eular angles
            this->updateCameraVectors();
        }

        // Processes input received from a mouse scroll-wheel event. Only requires input on the vertical wheel-axis
        void ProcessMouseScroll(GLfloat yOffset)
        {

        }

        GLfloat GetZoom()
        {
            return this->zoom;
        }

        glm::vec3 GetPosition()
        {
            return this->position;
        }

        glm::vec3 GetFront()
        {
            return this->front;
        }

    private:
```

Billboard.frag

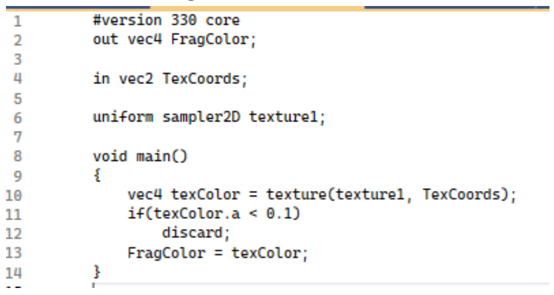
This fragment shader is designed to render textures with transparency, very useful for effects such as water drops, steam or any billboard. First it receives texture coordinates (TexCoords) generated in the vertex shader and uses the texture1 sampler to obtain the pixel color from the image. Then it applies a key condition: if the pixel's alpha component (texColor.a) is less than 0.1, the shader discards that fragment using discard, avoiding drawing completely transparent parts and achieving smooth edges without unwanted squares. Finally, if the pixel is valid, it places it on screen by assigning it to FragColor. In summary, this shader allows drawing particles and semi-transparent objects cleanly, correctly respecting the transparency of their texture.



```
#version 330 core
out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D texture1;

void main()
{
    vec4 texColor = texture(texture1, TexCoords);
    if(texColor.a < 0.1)
        discard;
    FragColor = texColor;
}
```

Billboard.vs

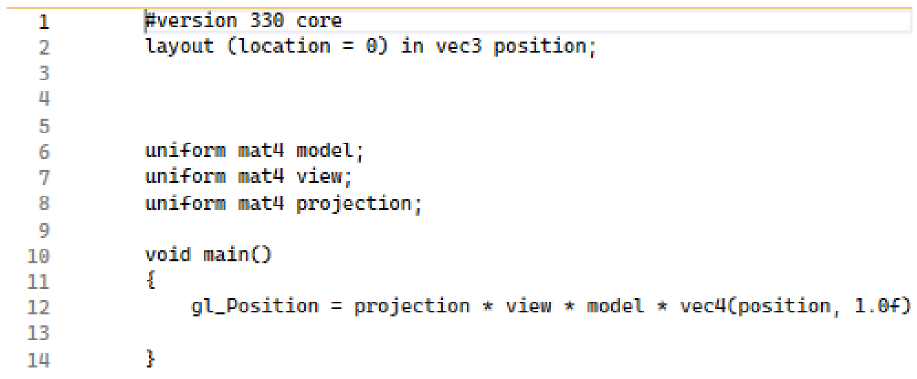
This vertex shader is responsible for preparing each vertex of a billboard or particle to be drawn in the 3D scene. It receives as input the vertex position (aPos) and its texture coordinates (aTexCoords) and sends the latter to the fragment shader through the TexCoords variable. Then it takes the vertex position and transforms it through the model, view and projection matrices, which locate, orient and correctly project the object in the 3D world and camera. The final result is assigned to gl_Position, which determines the exact place where the particle will appear. In summary, this shader positions the billboard geometry in 3D space and passes the texture coordinates necessary for final rendering.



```glsl
#version 330 core
out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D texture1;

void main()
{
    vec4 texColor = texture(texture1, TexCoords);
    if(texColor.a < 0.1)
        discard;
    FragColor = texColor;
}
```

Lamp.vs:

This vertex shader is responsible for transforming the cubes or spheres that represent light sources in the 3D world. Although they do not directly affect lighting, their correct visualization facilitates spatial orientation, positioning adjustment of point lights and visual debugging of the implemented lighting system.



```glsl
#version 330 core
layout (location = 0) in vec3 position;



uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    gl_Position = projection * view * model * vec4(position, 1.0f)

}
```

lighting.frag

This code corresponds to a fragment shader in GLSL responsible for calculating the final color of each fragment (pixel) of models rendered in the scene using the Phong lighting model. Its main objective is to simulate realistic light behavior by combining three types of light sources: a directional light, several point lights and a spotlight. In addition, it incorporates support for diffuse textures, specular textures and transparency, allowing dynamic visual effects within the 3D environment.

The shader starts by declaring several structures (struct) that group the necessary data to represent materials and light sources.

The Material structure contains:

- a diffuse texture (diffuse) that defines the base color of the object,
- a specular texture (specular) that controls in which areas the object reflects more light,
- and a shininess parameter that determines how concentrated or dispersed the specular brightness is.

Then three types of lights are defined:

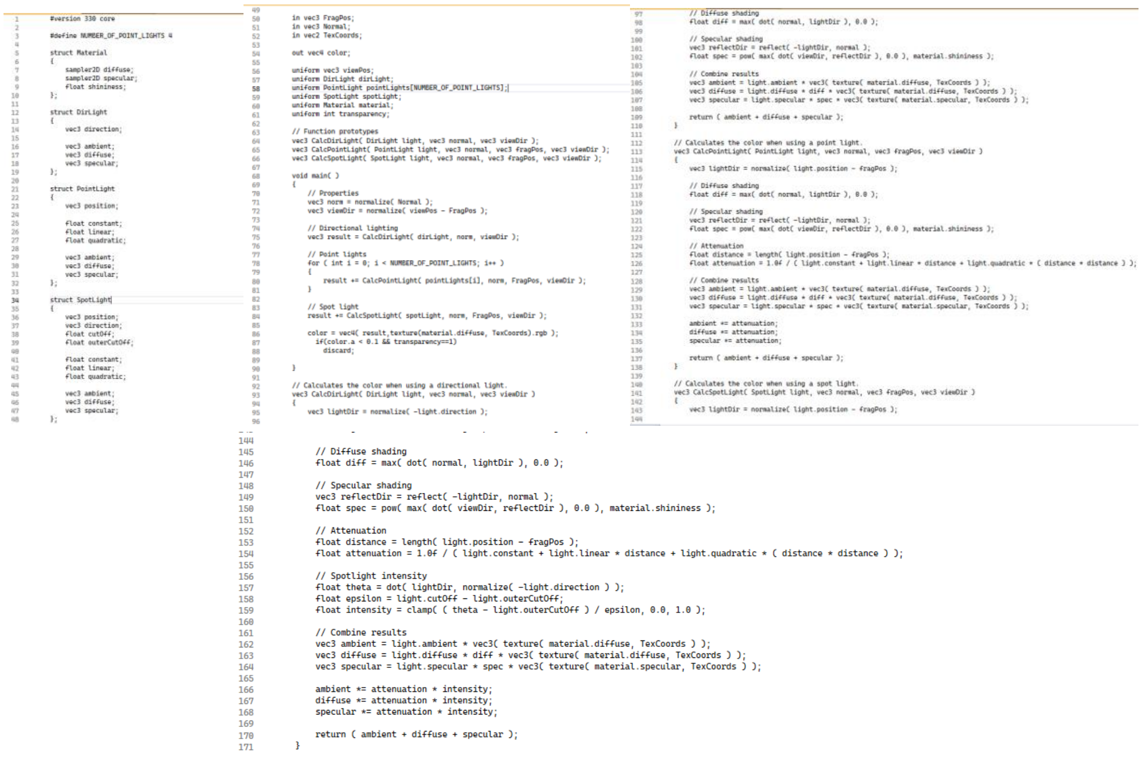
1. DirLight (directional light): Represents a global light similar to the sun. It has no position, only a fixed direction, and provides lighting throughout the scene. Its ambient, diffuse and specular components are applied uniformly to fragments.

2. PointLight (point light): Simulates spotlights or lamps placed within the scenario. Each point light has a position in 3D space, in addition to attenuation parameters (constant, linear and quadratic) that allow intensity to decrease as distance between light and fragment increases. This prevents distant objects from being illuminated with the same strength as nearby ones, providing realism.
3. SpotLight (flashlight type light): It is similar to a point light but with direction and conical shape. It has an internal angle (cutOff) where light is strong, and an external angle (outerCutOff) where light gradually fades. This allows simulating a flashlight or directed spotlight that illuminates only a specific area.

The shader receives from the vertex shader three interpolated values:
- FragPos: fragment position in the world,
- Normal: fragment normal (to know where the surface is facing),
- and TexCoords: UV coordinates to read textures. In main(), the shader normalizes the normal and calculates the view direction (viewDir) from the fragment to the camera. With this data, light calculation begins:
- First it applies directional light using CalcDirLight, obtaining the first lighting contribution.
- Then it goes through an array of point lights (pointLights[]) and adds the result of each one using CalcPointLight.
- Finally it adds the spotlight contribution through CalcSpotLight.

Each light function applies the classic Phong model:
1. Ambient component (ambient): constant base lighting that avoids completely black areas.
2. Diffuse component (diffuse): depends on the angle between the object normal and light direction; simulates how light "bounces" smoothly on matte surfaces.
3. Specular component (specular): simulates bright reflections; depends on the angle between light reflection and direction toward camera, and is controlled with shininess and specular texture. In point light and spotlight calculations, distance attenuation is included, which makes light gradually lose strength as the fragment is further from the spotlight. In spotlight, in addition to that, a cone intensity is calculated, which determines if the fragment is within the illuminated area and how strong the light should appear at that point.

At the end, the total result (result) represents the sum of all light contributions. That color is sent as shader output in color. In addition, the shader includes a transparency control: if the fragment's alpha is low and the transparency variable is activated, the fragment is discarded (discard). This is used for objects with transparent texture such as steam effects, particles or elements with alpha channel.



```
#version 330 core

#define NUMBER_OF_POINT_LIGHTS 4

struct Material
{
    sampler2D diffuse;
    sampler2D specular;
    float shininess;
};

struct DirLight
{
    vec3 direction;

    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};

struct PointLight
{
    vec3 position;

    float constant;
    float linear;
    float quadratic;

    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};

struct SpotLight
{
    vec3 position;
    vec3 direction;
    float cutOff;
    float outerCutOff;

    float constant;
    float linear;
    float quadratic;

    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};

in vec3 FragPos;
in vec3 Normal;
in vec2 TexCoords;

out vec4 color;

uniform vec3 viewPos;
uniform DirLight dirLight;
uniform PointLight pointLights[NUMBER_OF_POINT_LIGHTS];
uniform SpotLight spotLight;
uniform Material material;
uniform int transparency;

// Function prototypes
vec3 CalcDirLight( DirLight light, vec3 normal, vec3 viewDir );
vec3 CalcPointLight( PointLight light, vec3 normal, vec3 fragPos, vec3 viewDir );
vec3 CalcSpotLight( SpotLight light, vec3 normal, vec3 fragPos, vec3 viewDir );

void main( )
{
    // Properties
    vec3 norm = normalize( Normal );
    vec3 viewDir = normalize( viewPos - FragPos );

    // Directional lighting
    vec3 result = CalcDirLight( dirLight, norm, viewDir );

    // Point lights
    for ( int i = 0; i < NUMBER_OF_POINT_LIGHTS; i++ )
    {
        result += CalcPointLight( pointLights[i], norm, FragPos, viewDir );
    }

    // Spot light
    result += CalcSpotLight( spotLight, norm, FragPos, viewDir );

    color = vec4( result,texture(material.diffuse, TexCoords).rgb );
      if(color.a < 0.1 && transparency==1)
        discard;

}

// Calculates the color when using a directional light.
vec3 CalcDirLight( DirLight light, vec3 normal, vec3 viewDir )
{
    vec3 lightDir = normalize( -light.direction );

    // Diffuse shading
    float diff = max( dot( normal, lightDir ), 0.0 );

    // Specular shading
    vec3 reflectDir = reflect( -lightDir, normal );
    float spec = pow( max( dot( viewDir, reflectDir ), 0.0 ), material.shininess );

    // Combine results
    vec3 ambient = light.ambient * vec3( texture( material.diffuse, TexCoords ) );
    vec3 diffuse = light.diffuse * diff * vec3( texture( material.diffuse, TexCoords ) );
    vec3 specular = light.specular * spec * vec3( texture( material.specular, TexCoords ) );

    return ( ambient + diffuse + specular );
}

// Calculates the color when using a point light.
vec3 CalcPointLight( PointLight light, vec3 normal, vec3 fragPos, vec3 viewDir )
{
    vec3 lightDir = normalize( light.position - fragPos );

    // Diffuse shading
    float diff = max( dot( normal, lightDir ), 0.0 );

    // Specular shading
    vec3 reflectDir = reflect( -lightDir, normal );
    float spec = pow( max( dot( viewDir, reflectDir ), 0.0 ), material.shininess );

    // Attenuation
    float distance = length( light.position - fragPos );
    float attenuation = 1.0f / ( light.constant + light.linear * distance + light.quadratic * ( distance * distance ) );

    // Combine results
    vec3 ambient = light.ambient * vec3( texture( material.diffuse, TexCoords ) );
    vec3 diffuse = light.diffuse * diff * vec3( texture( material.diffuse, TexCoords ) );
    vec3 specular = light.specular * spec * vec3( texture( material.specular, TexCoords ) );

    ambient *= attenuation;
    diffuse *= attenuation;
    specular *= attenuation;

    return ( ambient + diffuse + specular );
}

// Calculates the color when using a spot light.
vec3 CalcSpotLight( SpotLight light, vec3 normal, vec3 fragPos, vec3 viewDir )
{
    vec3 lightDir = normalize( light.position - fragPos );

    // Diffuse shading
    float diff = max( dot( normal, lightDir ), 0.0 );

    // Specular shading
    vec3 reflectDir = reflect( -lightDir, normal );
    float spec = pow( max( dot( viewDir, reflectDir ), 0.0 ), material.shininess );

    // Attenuation
    float distance = length( light.position - fragPos );
    float attenuation = 1.0f / ( light.constant + light.linear * distance + light.quadratic * ( distance * distance ) );

    // Spotlight intensity
    float theta = dot( lightDir, normalize( -light.direction ) );
    float epsilon = light.cutOff - light.outerCutOff;
    float intensity = clamp( ( theta - light.outerCutOff ) / epsilon, 0.0, 1.0 );

    // Combine results
    vec3 ambient = light.ambient * vec3( texture( material.diffuse, TexCoords ) );
    vec3 diffuse = light.diffuse * diff * vec3( texture( material.diffuse, TexCoords ) );
    vec3 specular = light.specular * spec * vec3( texture( material.specular, TexCoords ) );

    ambient *= attenuation * intensity;
    diffuse *= attenuation * intensity;
    specular *= attenuation * intensity;

    return ( ambient + diffuse + specular );
}
```

lighting.vs

The lighting.vs file corresponds to the vertex shader responsible for processing each of the vertices used in the 3D scene. Its main function is to transform the original position of vertices, prepare the necessary information for lighting calculations and ensure that textures are applied correctly in the fragment shader. This shader constitutes the first stage of OpenGL's programmable graphics pipeline.

The shader receives three fundamental attributes:

- position: represents the vertex position in the model's local space.
- normal: normal vector associated with the vertex, necessary to determine light interaction with surfaces.
- texCoords: UV coordinates that indicate which part of the texture corresponds to each vertex.

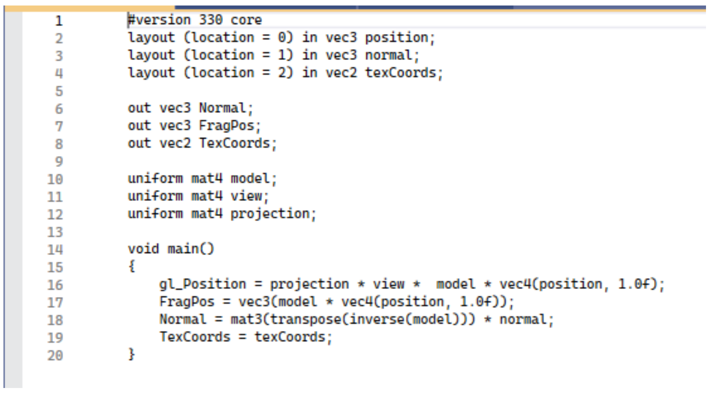
Similarly, the shader uses three matrices sent from the main application:

- model, which transforms the object from its local space to global space;
- view, which corresponds to the camera transformation within the scene;
- projection, which applies the necessary perspective to generate the visual depth of the scene.

Within the main() function, the vertex shader executes the essential operations to prepare the vertex:

1. Position transformation: gl_Position = projection * view * model * vec4(position, 1.0f);With this instruction, the vertex passes from local space to clip space, allowing its correct representation on screen.
2. Calculation of the vertex global position: FragPos = vec3(model * vec4(position, 1.0f)); This position is essential for lighting calculations in the fragment shader.
3. Normal transformation: Normal = mat3(transpose(inverse(model))) * normal; Normals require a special transformation through the normal matrix to preserve their orientation even when the model has been scaled or rotated. 4. Texture coordinate assignment: 5. TexCoords = texCoords;

UV coordinates are sent without modification to ensure precise texturing in the following stages.



```
#version 330 core
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec2 texCoords;

out vec3 Normal;
out vec3 FragPos;
out vec2 TexCoords;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    gl_Position = projection * view *  model * vec4(position, 1.0f);
    FragPos = vec3(model * vec4(position, 1.0f));
    Normal = mat3(transpose(inverse(model))) * normal;
    TexCoords = texCoords;
}
```

modelLoading.frag

The modelLoading.frag file corresponds to the fragment shader used during loading and rendering of models imported through the Model class. Its main function is to obtain the final color of each fragment from a diffuse texture, also applying a transparency control to discard fragments that should not be rendered. This shader is used on objects that do not require complex lighting calculations and whose appearance depends directly on the applied texture.

The shader receives as input the texture coordinates (TexCoords) generated and previously sent from the vertex shader. From these coordinates, it accesses the associated texture through the uniform:
uniform sampler2D texture_diffuse1;

The main process is performed within the main() function. First, the shader obtains the fragment color by querying the texture:

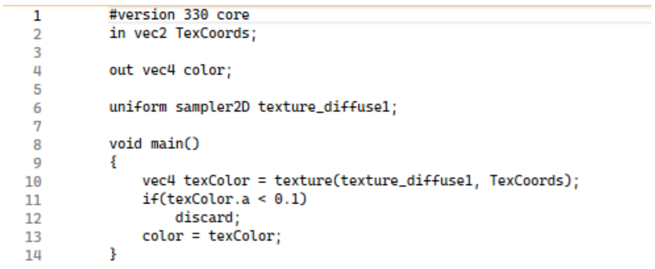
vec4 texColor = texture(texture_diffuse1, TexCoords);

This value contains RGB components and the alpha channel, which is used to determine if the fragment should be displayed or discarded. The shader applies a transparency verification:

if(texColor.a < 0.1) discard;

If the alpha value is very low (less than 0.1), the fragment is discarded, which allows transparent areas of the texture not to appear on screen. This is fundamental for models that contain partially transparent objects, cutouts or contours defined by alpha. Finally, if the fragment is not discarded, the shader assigns the obtained color directly to the output pixel:

color = texColor; This way, the final color will depend solely on the texture, without additional modifications or light calculations.



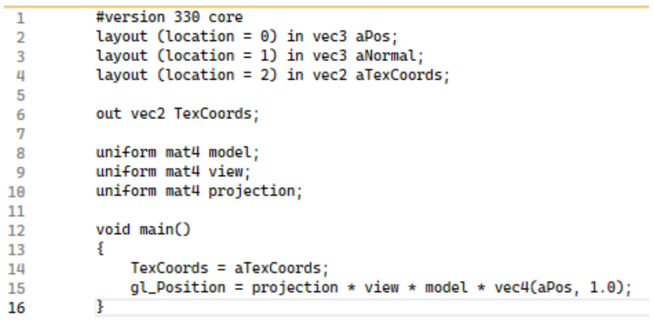
```
#version 330 core
in vec2 TexCoords;

out vec4 color;

uniform sampler2D texture_diffuse1;

void main()
{
    vec4 texColor = texture(texture_diffuse1, TexCoords);
    if(texColor.a < 0.1)
        discard;
    color = texColor;
}
```

modelLoading.vs

This vertex shader has the purpose of transforming the vertex positions of the model for their correct projection on screen, in addition to sending texture coordinates to the fragment shader. The shader receives the vertex position, its normal and UV coordinates, although in this case it only uses the texture coordinates.
Within the main() function, the shader directly assigns UV coordinates:

TexCoords = aTexCoords;

Subsequently, it calculates the final vertex position by multiplying it by the model, view and projection matrices, which places the object in its position within the scene and applies the necessary perspective: gl_Position = projection * view * model * vec4(aPos, 1.0); This shader is simple and is designed for models that only require spatial transformation and texture application, without the need for lighting calculations at this stage.



```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoords;

out vec2 TexCoords;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    TexCoords = aTexCoords;
    gl_Position = projection * view * model * vec4(aPos, 1.0);
}
```

CONCLUSIONS.

The development of this project allowed the practical integration of knowledge acquired in 3D modeling and graphics programming, covering the entire process from object creation in Blender to its implementation and interaction within an environment rendered with OpenGL. Model construction, texturing, animation application, use of custom shaders and configuration of different types of lighting provided a complete understanding of the workflow in graphics applications. This project consolidated technical skills and reinforced the importance of planning, stage-by-stage organization and visual coherence, allowing the creation of a functional, dynamic and aesthetically integrated scene.

REFERENCES.

➔ Ing. Espinoza Urzúa, E. Computer Graphics Course Semester 2025-2.

➔ Angel, E., & Shreiner, D. (2012). Interactive Computer Graphics: A Top-Down Approach with Shader-Based OpenGL (6th ed.). Pearson Education.

➔ Assimp. (n.d.). Open Asset Import Library (Assimp). Retrieved from https://www.assimp.org/

➔ LearnOpenGL. (n.d.). LearnOpenGL. Retrieved from https://learnopengl.com/

➔ GLFW. (n.d.). Graphics Library Framework. Retrieved from https://www.glfw.org/

➔ GLEW. (n.d.). The OpenGL Extension Wrangler Library. Retrieved from http://glew.sourceforge.net/

➔ GitHub Docs. (n.d.). Managing repositories. Retrieved from https://docs.github.com/en/repositories