311158894

Group Theory: 05

Laboratory Group: 13

Final Project
House with 2 rooms

Objective

Develop an interactive virtual tour using OpenGL that realistically simulates the exploration of an architectural structure composed of two rooms, based on previously selected real or fictional visual references (excluding specific themes indicated in the guidelines). The project will integrate a synthetic camera for navigation, and an artistic design consistent with the selected reference images. In addition, a complete documentation will be generated that includes a Gantt chart, a bilingual user manual (Spanish or English) with the objectives and description of the interaction, a technical manual, and a cost analysis with justification of the estimated sales price. The result will include a functional executable file, a well-structured GitHub repository with no compressed files or larger than 100 MB, meeting criteria of realism, presentation and organization.

Scope

This project is based on the knowledge acquired throughout the semester and uses technologies such as GLEW, GLFW and OpenGL. The implementation will be based on the reference images (included later in this document), which provide a clear basis for replicating the spaces as accurately as possible, considering the limitations of the level of knowledge achieved so far. This work has been conceived as a direct comparison with advanced systems such as Lumion, a state-of-the-art rendering engine widely used in architectural firms.

The project will meet all the characteristics set out in the overall objective and it is important to underline that it will be fully developed through direct programming in OpenGL. It will take place in the Visual Studio environment, where the different *shaders* and the necessary logic for the interaction and rendering of the virtual tour will be implemented.

An additional relevant aspect is the use of SketchUp as a support tool for modeling, since it is mainly oriented to architectural design, which will facilitate the precise creation of the required spaces.

During development, all project progress will be documented and uploaded to the general semester repository, where lab practices will also be available. However, the final product to be delivered will include only the necessary files, properly organized and documented, as requested.

Finally, since the reference images contain a considerable number of objects, some additional elements not explicitly required by the teacher will be included. This with the aim of achieving a greater degree of visual fidelity, technical quality and general presentation in the final product.

Schedule of activities

The first step of this project was to select a suitable base model to work with. To this end, various options were explored, and one was chosen that presented a sufficiently high degree of complexity, in accordance with the established objectives.

Next, the feasibility of integrating SketchUp with OpenGL seamlessly was verified. This step was crucial, as it was necessary to verify that the three-dimensional objects could be exported correctly from

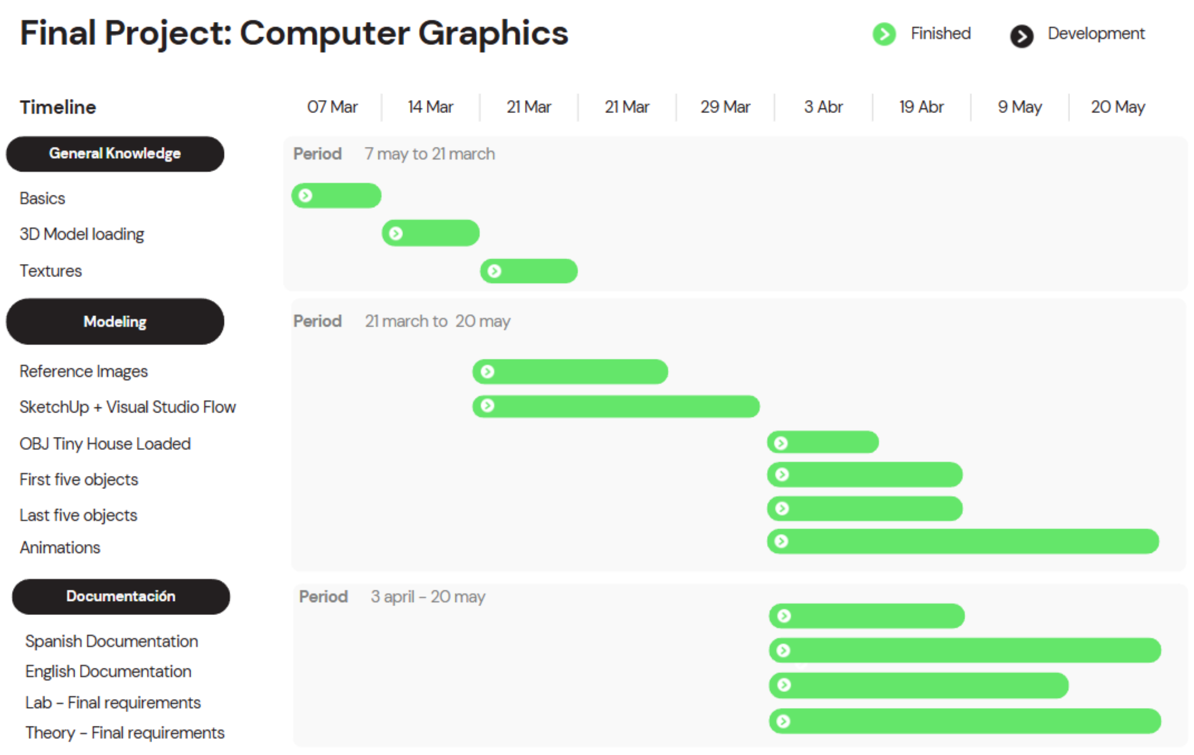
SketchUp and read without errors in the development environment. If this integration did not work properly, the rest of the project would not have been viable with the proposed tools.

Once this stage was over, the environment was modeled in SketchUp. During this phase, constant testing was performed to ensure that the models were loaded correctly into OpenGL using Visual Studio. Once the three-dimensional model was finalized, textures were applied based on the reference images that had been provided in the planning stage.

With the environment modeled and textured, the objects detailed in the list presented to the teacher were incorporated. Each one was individually verified to ensure its correct loading and display in the development environment.

Finally, the technical documentation of the project was developed, which provided the required professional support. This included the objectives of the project, its scope, schedule (Gantt chart), reference images, cost analysis, technical manual, user manual and the other elements requested.

Gantt Chart



Gantt chart.

Limiting

In this project, several challenges were presented that exceeded the previously acquired knowledge. One of the main challenges was mastering advanced rendering techniques, such as lighting. Several difficulties were faced when trying to implement the different lighting elements, as it passed through objects instead of being projected and reflected correctly on their surfaces.

On the other hand, texturing failed to offer the visual fidelity achieved by programs such as Lumion, which integrate normal maps to simulate the behavior of light more realistically. As a result, a considerably lower quality visual effect was generated.

In addition, the objects used had a very low geometric resolution, that is, a reduced number of vertices, edges and triangles. For this reason, more viable alternatives were chosen that would allow representing, as far as possible, the elements of an architectural scene with the available resources.

Finally, another important limitation was the setting of the environment. It was difficult to adequately simulate elements such as trees, sky, and grass with volume, which reduced the overall realism of the setting.

Technical Manual

This technical manual documents the development of a computer graphics project whose objective is to render an interactive three-dimensional scene using OpenGL. The program loads 3D models in OBJ format, implements a first-person camera, and simulates different types of lighting, including directional, spot, and spotlight lights.

The application has been developed in C++ with the use of specialized libraries such as GLEW for handling OpenGL extensions, GLFW for creating windows and capturing events, GLM for mathematical operations, and stb_image or SOIL2 for handling textures. In addition, custom classes such as Shader, Camera, and Model have been developed to facilitate the system's modular architecture.

This document describes the structure of the code, the configuration of lights and textures, the process of loading and rendering models, as well as the procedures for compiling and executing the project correctly.

Applied Development Methodology

A structured, modular, object-oriented programming approach was adopted for the development of this project. Although a formal software engineering methodology such as Scrum or Waterfall was not implemented, fundamental principles of good practices in graphic programming were as follows:

- Modularity: Clear separation of code into specific classes: Shader for shader management, Camera for scene navigation, and Model for model loading and rendering.

- Incremental iteration: The project was developed in functional stages: basic OpenGL configuration, camera implementation, model loading, and finally lighting.
- Code reuse: Third-party reusable libraries (GLFW, GLEW, stb_image, GLM, SOIL2) were used to focus the development on the logic of the project.
- Constant testing: Throughout development, frequent runs were made to verify the correct display of models, lights and camera movement.

This approach made it possible to maintain code that was readable, uncluttered, and easy to debug during all phases of development.

System Requirements and Settings in Visual Studio

To successfully build and run this project in Visual Studio, the following requirements must be met:

System Requirements:

- Operating system: Windows 10 or higher
- RAM: minimum 4 GB (recommended 8 GB or more)
- OpenGL 3.3 or higher compatible graphics card
- Free disk space: at least 200 MB for resources, libraries, and models

Settings in Visual Studio:

1. Recommended version: Visual Studio 2019 or 2022 with the desktop development package with C++ installed.
2. Create a new project:
   - Project type: "Console application in C++".
3. Add the project files: main.cpp, Shader.h/.cpp, Camera.h/.cpp, Model.h/.cpp, and the model and shader folders.
4. Configure additional routes:
   - In C/C++ → project properties → General → Additional Inclusion Directories:
     - Path to GLEW (includes /include folder)
     - Route to GLFW, GLM, stb_image, and SOIL2 (if applicable)
   - In project properties → General → Linker → Additional Library Directories:
     - Path to the (/lib) libraries of GLEW, GLFW, etc.
5. Add dependencies to the linker:
   - In project properties → Linker → Entry → Additional Dependencies:
   - opengl32.lib

        - glew32.lib
        - glfw3.lib
        - SOIL2.lib
        - Assimp

6. Make sure you have the necessary DLLs:
    - Copy files such as glew32.dll, glfw3.dll, SOIL2.dll. assimp-vc-140mt.dll according to the structure that is in place in the repository.

Once set up correctly, the project should compile and run showing an interactive 3D scene with camera and uploaded models.

Code Structure and Explanation of Operation

The project is organized in a modular and structured way to facilitate its understanding and maintenance. The main components are described below:

Main Archives:

- main.cpp: Contains the main program flow, window settings, render loop, and user input logic.
- Shader.h/.cpp: Implements a class for compiling, binding, and using shaders (GLSL).
- Camera.h/.cpp: Control camera movement and orientation using keyboard and mouse.
- Model.h/.cpp: Handles the loading of 3D models in .obj format and their rendering.

This organization ensures clear and modular operation of the developed graphics engine, facilitating the expansion with new functionalities such as animations or physics.

Code Documentation

Below is detailed documentation of the key sections of the project's source code:

main.cpp

- Libraries used: Includes standard libraries (iostream, cmath), graphics (GLEW, GLFW), image utilities (stb_image, SOIL2), mathematics (GLM), and custom classes (Shader, Camera, Model).

- Global variables: Constants are declared for window size, keyboard states, camera settings, and control of the time between frames (deltaTime).
- Light and color: Structures are defined to handle the position of point lights and visual effects such as dynamic color changes with trigonometric functions.

Key features

- KeyCallback(GLFWwindow*, int, int, int, int): Handles keyboard inputs and allows you to move lights or activate effects.
- MouseCallback(GLFWwindow*, double, double): Controls the rotation of the camera with the mouse.
- DoMovement(): Updates the camera position based on keystrokes.
- Animacion(): Updates the variables that control all the animations based on the time.

Main loop (while)

- Cleans buffers (glClear).
- Calculate time between frames.
- Apply transformations (view, projection, model).
- Update the properties of each type of light and pass the data to the shaders.
- Draw loaded models in the scene: Ground, Tiny, Crystal.
- Render a cube representing light with lampShader.

Shaders and Textures

- The Shader class is used to compile and link custom shaders.
- Diffuse and specular textures are assigned through texture units (GL_TEXTURE0, GL_TEXTURE1).

Models

- The Model class is used to import .obj objects and render them using Draw(shader).

Camera.h

Libraries used

This file includes standard C++ libraries (<vector>), as well as specific OpenGL and graphical math libraries such as GLEW and GLM (glm/glm.hpp, glm/gtc/matrix_transform.hpp).

Variables and constants

- Enumeration Camera_Movement: Defines the possible movements of the camera (forward, backward, left, right).
- Default Constants: Sets default values for YAW, PITCH, SPEED, SENSITIVITY, and ZOOM.

Camera Class

A class that abstracts the logic of a camera in a three-dimensional environment, in charge of managing its position, orientation and projection. It allows you to receive keyboard and mouse inputs to freely navigate the scene.

Main attributes:

- position, front, up, right, worldUp: Vectors that define the position and spatial orientation of the camera.
- yaw, pitch: Camera rotation angles.
- movementSpeed, mouseSensitivity, zoom: Dynamic camera configuration parameters.

Main methods:

- Constructors: Allow you to initialize the camera with vectors or scalar values.
- GetViewMatrix(): Returns the view array using the glm::lookAt function.
- ProcessKeyboard(Camera_Movement, GLfloat): Processes keyboard input to move the camera based on direction and time.
- ProcessMouseMovement(GLfloat, GLfloat, GLboolean): Updates the yaw and pitch angles to rotate the camera with the mouse.
- ProcessMouseScroll(GLfloat): Method prepared to zoom with the mouse wheel (currently empty).
- Getters: Methods such as GetZoom(), GetPosition(), and GetFront() allow you to get the current state of the camera.
- updateCameraVectors(): A private method that recalculates the front, right, and up vectors from the current angles. It's crucial for maintaining consistent visual orientation when rotating the camera.

Mesh.h

General purpose

This class encapsulates the structure and rendering of a 3D mesh composed of vertices, indexes, and textures. It is designed to be used in conjunction with models imported through Assimp.

Main structures

- Vertex: Contains position (vec3), normal, and texture coordinates.
- Texture: Saves the texture ID, type (fuzzy or specular), and original path.

Key functionality

- Mesh() constructor: Receives vertex, index, and texture data; initializes buffers using setupMesh().
- Draw(Shader): Renders the mesh by linking textures and sending data to the shader.
- setupMesh() (private): Configures VAO, VBO, and EBO, and defines vertex attribute pointers.

Model.h

General purpose

The Model class allows you to import, process, and render entire 3D models from .obj files using the Assimp library. Automate the handling of multiple mesh, textures, and hierarchical structures.

Key functionality

- Constructor Model(path): Loads the model from file and processes all of its meshes recursively.
- Draw(Shader): Draws each loaded mesh using the shader provided.

Internal processes

- loadModel(): Uses Assimp to read the file, extract nodes, and loop through them recursively.
- processNode(): Loops through the nodes of the model, extracts each mesh (aiMesh), and transforms it into a Mesh instance.
- processMesh(): Translates vertices, normals, UV coordinates, and indexes from the Assimp format to the format used by OpenGL.
- loadMaterialTextures(): Avoid loading duplicate textures, assign fuzzy and specular textures per mesh.

Loading external textures

- TextureFromFile(): Helper function that loads images from disk using SOIL2 and generates texture GLuints in OpenGL.

Shader.h

General purpose

The Shader class encapsulates the logic for loading, compiling, debugging, and using vertex and fragment shaders in OpenGL. It provides a simple interface for working with shader programs during rendering.

Key functionality

- Shader Constructor(vertexPath, fragmentPath):Loads GLSL files from disk, compiles the shaders, creates the shader program, and binds both modules. It also handles and displays build and binding errors if they occur.
- Use():Enables the current shader to be used in drawing operations (glUseProgram).
- getColorLocation():Returns the location of the uniform variable "color" in the shader, useful for manipulating colors from C++.

Internal variables

- Program: Identifier of the shader program created and bound.
- uniformColor: Location of the uniform color variable within the fragment shader (if used).

---

core.frag

General purpose

This fragment shader defines the final color of each pixel using an interpolated color input (ourColor) from the vertex shader or a uniform variable.

Functionality

- Input (in vec3 ourColor): Receives the vertex color interpolated by rasterization.
- Output (out vec4 color): Defines the final color of the fragment that will be displayed on the screen, including a fixed alpha channel of 1.0 (full opacity).
- main(): Directly assigns the value of ourColor with full opacity to the fragment.

---

core.vs (Vertex Shader)

General purpose

This vertex shader transforms the position of each model vertex to clip coordinates using the model, view, and projection matrices, and transmits a uniform color to the fragment shader.

Functionality

- Input (layout(location = 0) in vec3 position): Receives the position of each vertex from the VBO.
- Uniforms:
  - mat4 model: Transformation of the local object into the world.
  - mat4 view: World-to-view transformation (camera).
  - mat4 projection: Perspective or orthographic projection.
  - vec3 color: Uniform color to be used per vertex.
- Output (out vec3 ourColor): Passes the color to the fragment shader.

Main function():

- Calculates gl_Position as the product of the transformation matrices applied to the vertex position.
- Assigns the uniform color to the ourColor variable, which will be interpolated on a per-fragment basis.

---

lighting.frag

This shader applies Phong lighting by combining directional light, four spot lights, and a spotlight light. It uses diffuse and specular textures, includes distance dimming, and allows you to discard transparent fragments if the transparency option is enabled. The illumination is calculated using auxiliary functions according to the type of light and is combined into the final color of the fragment.

---

lighting.vs

This shader transforms the position of each vertex to the clip space (gl_Position) using the model, view, and projection matrices. It calculates the position of the fragment in the world (FragPos), the correctly transformed normal (Normal), and transmits the texture coordinates (TexCoords) to the fragment shader for lighting and texturing.

Libraries and Bookstores Used:

```
#include <iostream> // Console input/output

#include <cmath> // Standard Math Functions

#include <GL/glew.h> // Handling Modern OpenGL Extensions

#include <GLFW/glfw3.h> // Windows and User Input

#include "stb_image.h" // Loading Textures

#include <glm/glm.hpp> // Mathematical types (vec3, mat4, etc.)
```

```
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
#include "SOIL2/SOIL2.h" // Alternate texture loading
#include "Shader.h" // Custom class for handling shaders
#include "Camera.h" // Class to control the camera
#include "Model.h" // Class for loading and drawing .obj models
```

User Manual

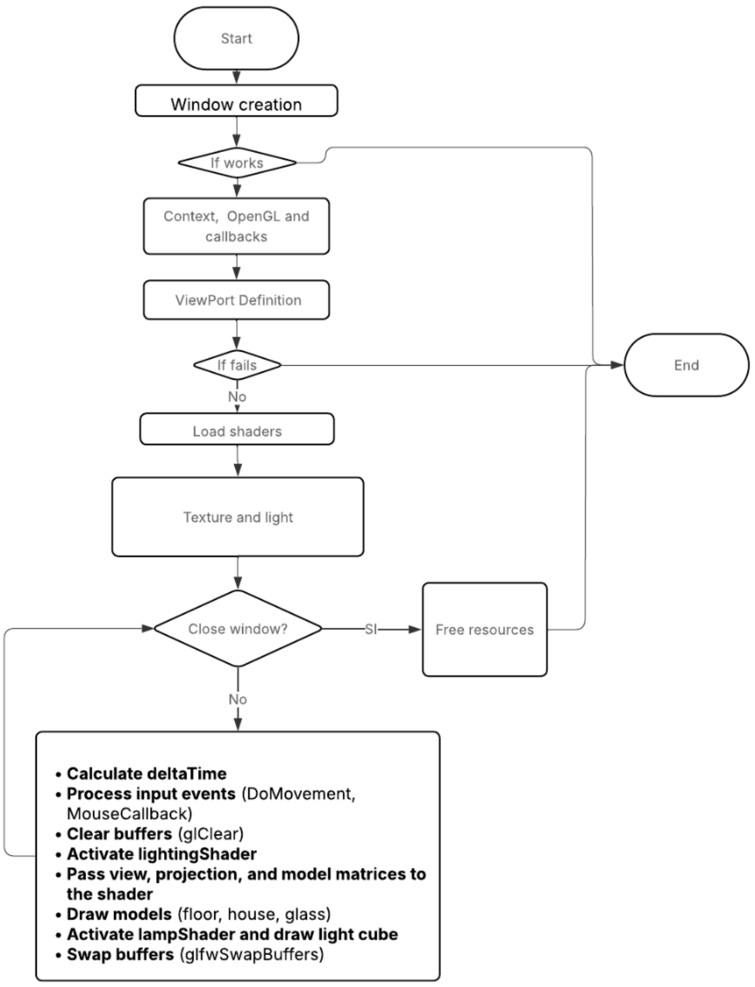
To use this program is very simple. All you need is a keyboard and mouse. The following keys allow you to navigate the three-dimensional environment:

- **W**: Move Forward
- **S**: Move backwards
- **A**: Move to the left
- **D**: Move to the right
- **1**: Open Front Door
- **2**: Close window facing the entrance
- **3**: Open microwave door
- **4**: Move the entire RV (simulating moving forward)

Also, you can use the **mouse** to rotate the camera and observe the surroundings in all directions.

The navigation experience is designed in first person, like a 3D video game. It is recommended to use a minimum screen resolution of 1920x1080 for optimal viewing.

Code Flowchart



Start
Window creation
If works
Context, OpenGL and callbacks
ViewPort Definition
If fails
No
End
Load shaders
Texture and light
Close window?
SI
Free resources
No
• Calculate deltaTime
• Process input events (DoMovement, MouseCallback)
• Clear buffers (glClear)
• Activate lightingShader
• Pass view, projection, and model matrices to the shader
• Draw models (floor, house, glass)
• Activate lampShader and draw light cube
• Swap buffers (glfwSwapBuffers)

Flowchart

Reference Images and Results

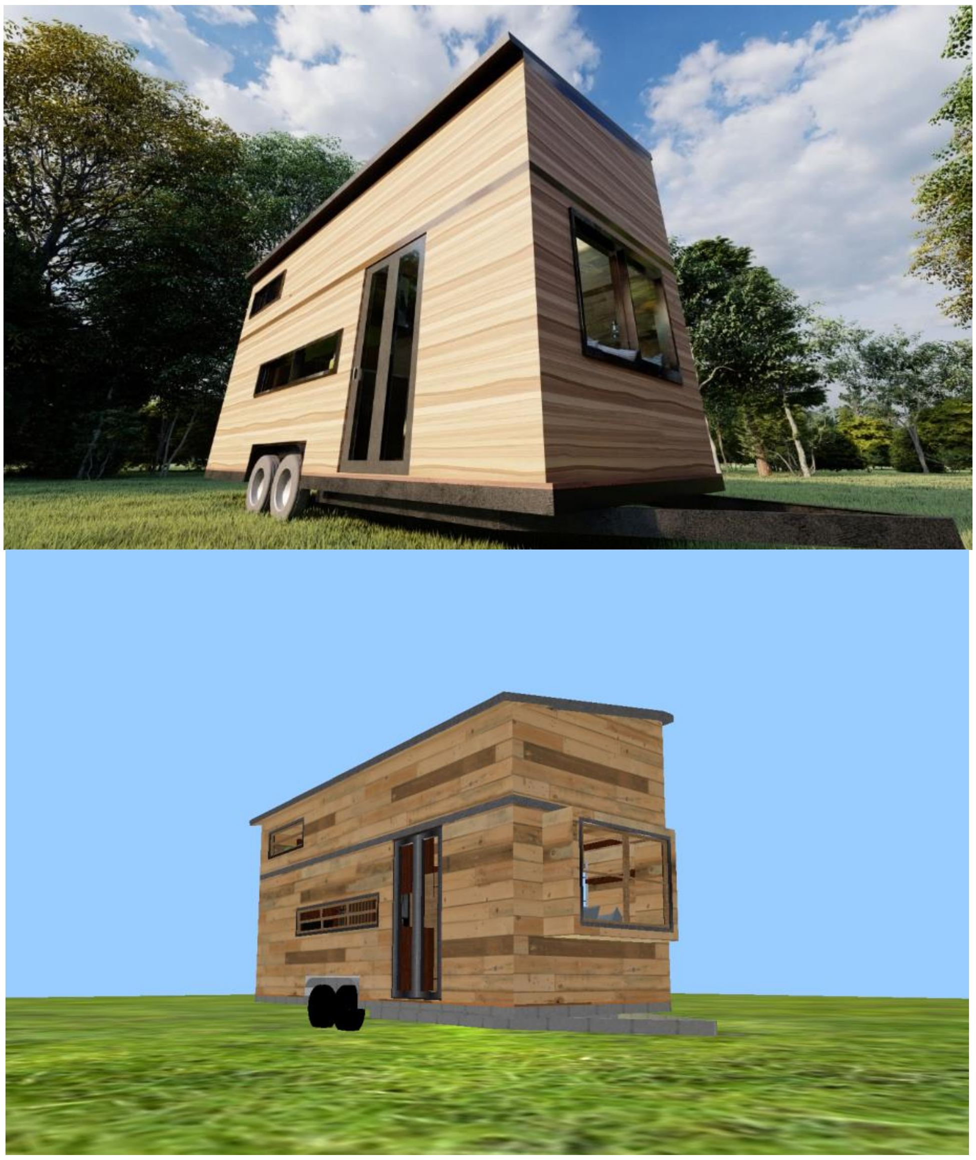


Figure 1. Outside.

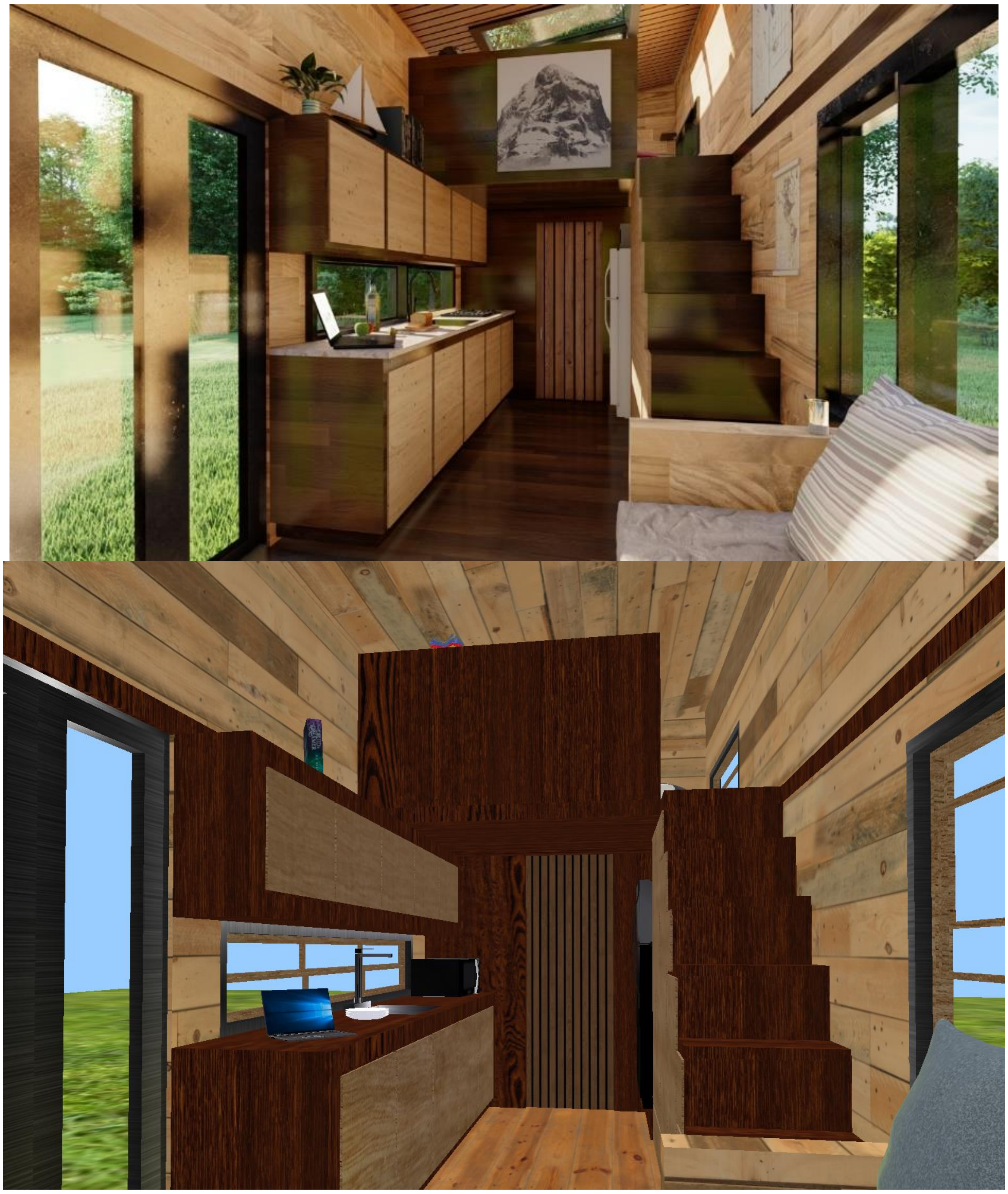


Figure 2. Interior overlooking the kitchen and stairs.

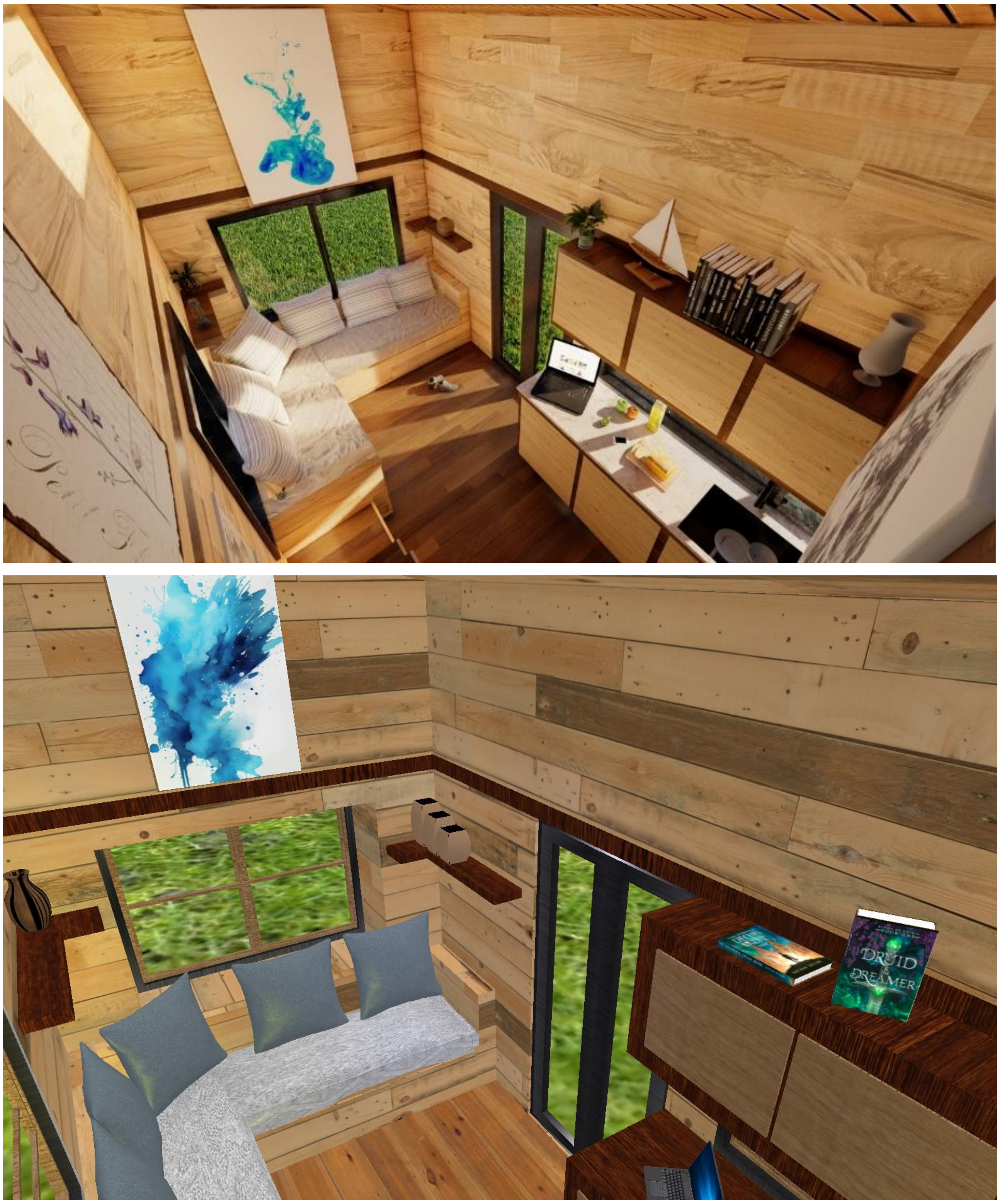


Figure 3. Top view.

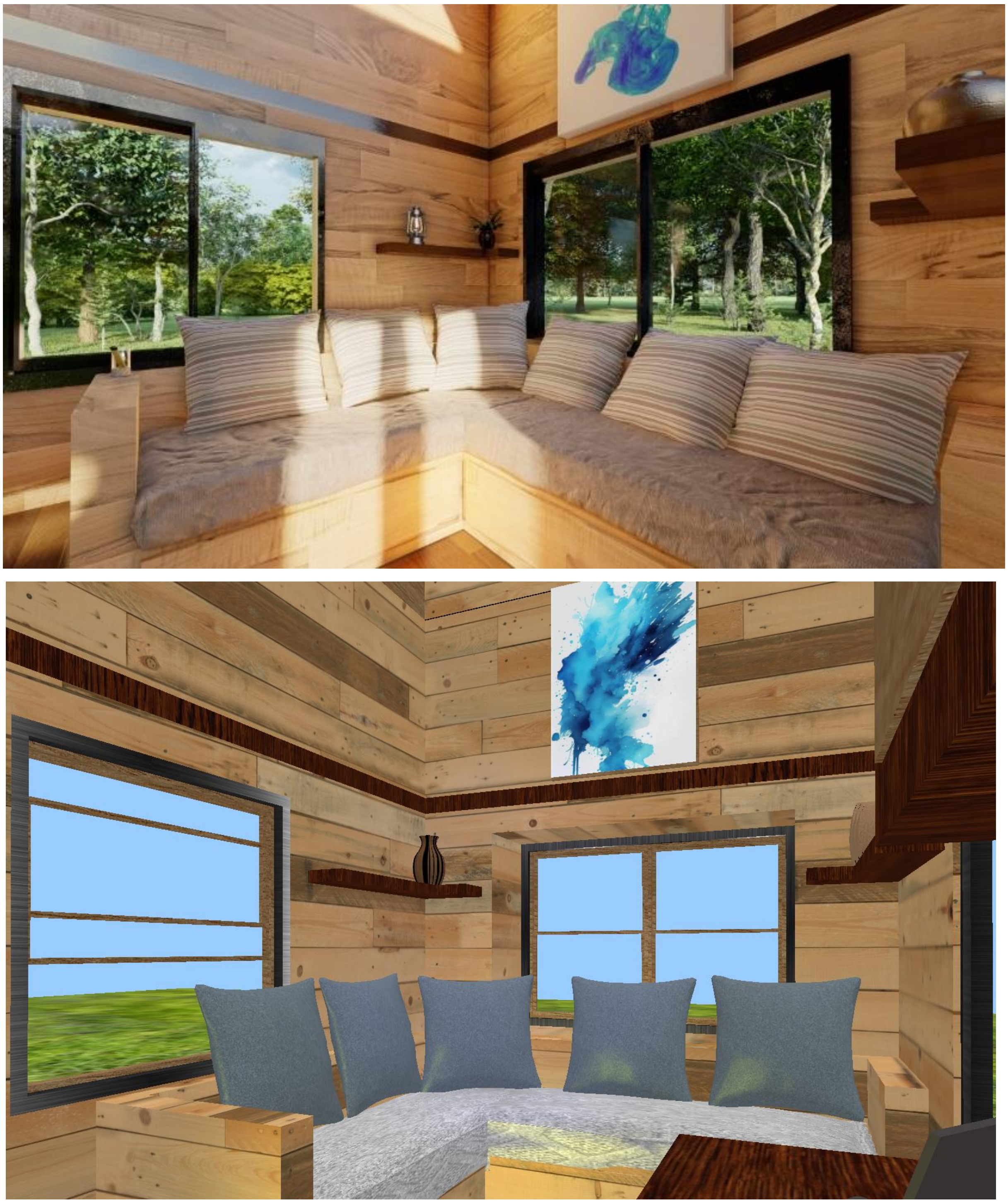


Figure 4. Living room

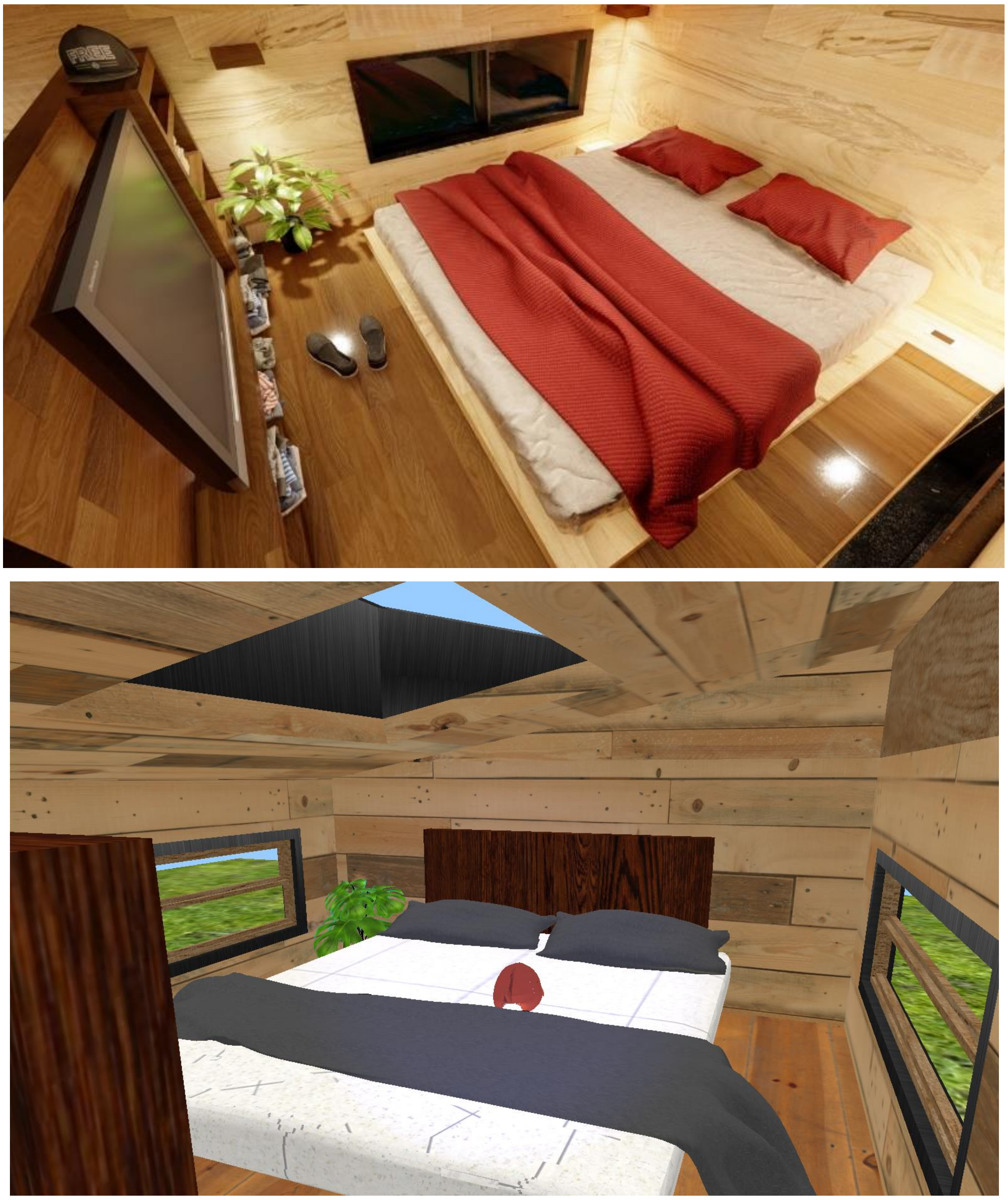


Figure 5. Room.

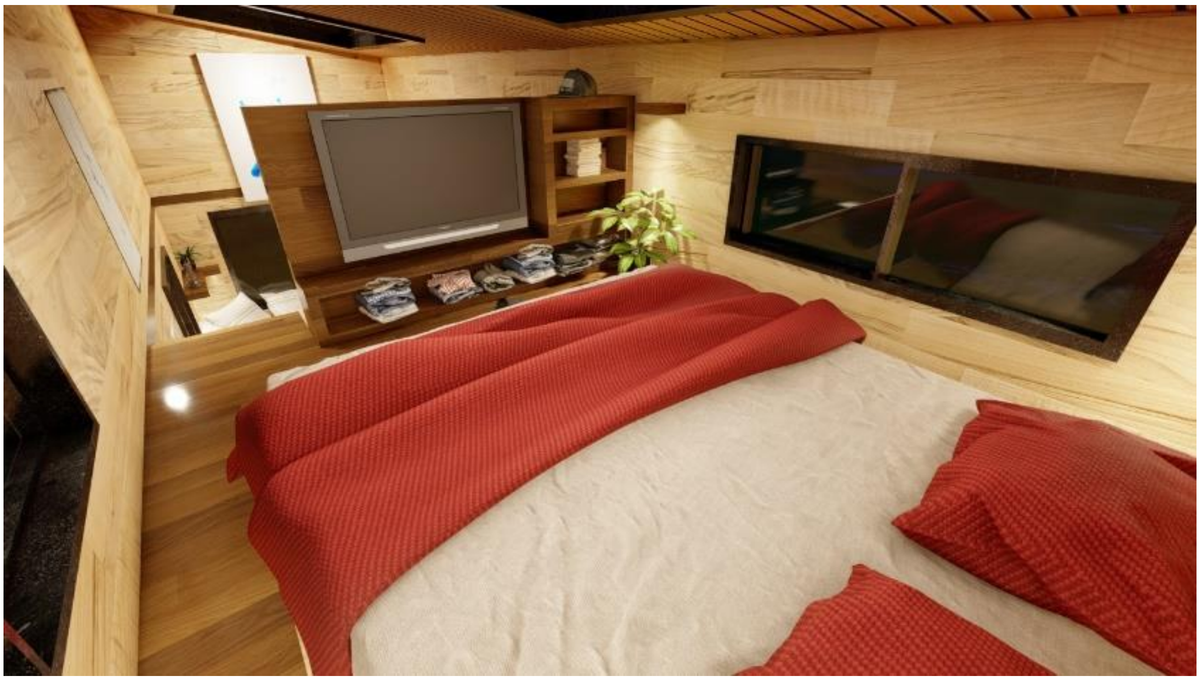
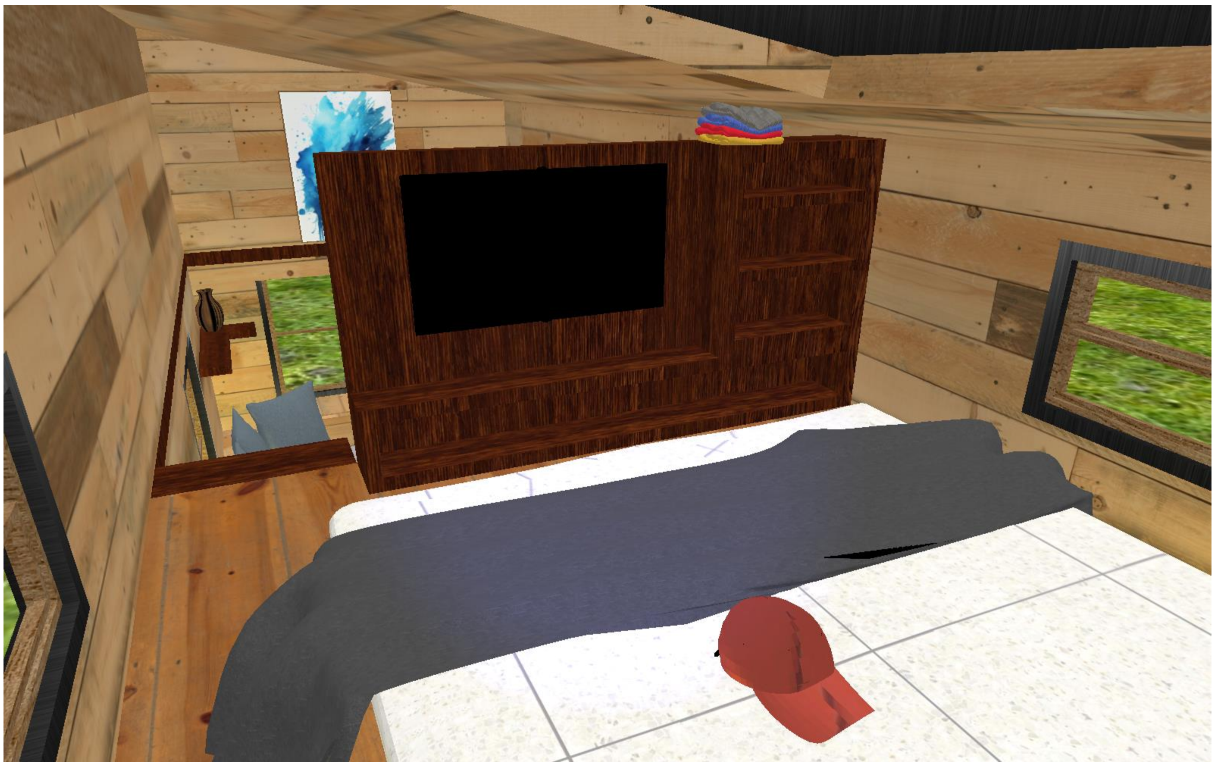



Figure 6. Alternative view room.

Cost Analysis

Concept	**Detail**	**Approximate Cost (MXN)**
Computer	Proportional use of own equipment valued at $15,000 (20% for this project)	$3,000
Sketchup License	Free trial	$0
Textures	Free download from the web	$0
Total labor (month and a half)	Estimated 100 hours * $100/hour	$10,000
Learning OpenGL animations and using Sketchup	5 hours self-paced * $100/hour	$500
Project Documentation	Preparation of manual, image editing and final PDF	$200
Estimated total		**$13,700**

Conclusions

The project managed to meet all the established objectives, consolidating itself as a comprehensive demonstration of the knowledge acquired in the Computer Graphics course. Through the use of OpenGL and its ecosystem of tools, key systems were implemented that enabled the creation of a three-dimensional interactive experience: from the import and manipulation of 3D models to the simulation of different types of lighting and a fully functional first-person camera.

The successful generation of an executable file represents not only the technical closure of the project, but also its transformation into an autonomous application accessible to the end user. This achievement would not have been possible without a deep understanding of system dependencies, library management, and careful configuration of the development environment.

During the development, various challenges arose, especially due to the academic suspension of one month due to unemployment, which forced them to rethink times, priorities and work strategies. Despite this, it was possible to stay on track thanks to detailed planning reflected in the Gantt chart, which served as an organizational compass to continue advancing on the different fronts of the project: modeling, texturing, rendering and navigation.

This virtual tour is not only a technical simulation: it is a gateway to more complex worlds. The foundation was laid for understanding next-level technologies such as Vulkan or ray tracing techniques, which require deep familiarity with concepts such as buffers, shaders, and rendering pipelines. In this way, the project not only fulfilled its pedagogical function, but also awakened a broader vocation for the technical and artistic exploration of three-dimensional development.

Finally, this experience also left lessons about the value of artisanal work in graphic programming. Opting for a build from scratch instead of using commercial engines like Unity or Unreal allowed you to value the actual effort behind each line of code, as well as understand the production costs in time and resources. This knowledge is key to any future professional or academic endeavor in the field of visual technology.

Annexes

Repository:

https://github.com/acasanova009/CompuGrafica

Free SketchUp:

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