



**UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO**

**FACULTAD DE INGENIERÍA**

**INGENIERÍA EN COMPUTACIÓN**

**Computación Gráfica e Interacción Humano – Computadora**

**Profesor: Ing. Carlos Aldair Román Balbuena**

**Proyecto Final:**

**Manual Técnico.**

**Alumno: 319252903**

**Grupo: 05**

**Fecha de entrega: 24 de noviembre de 2025**

# ÍNDICE

MANUAL TÉCNICO - PROYECTO FINAL.....	2
DESCRIPCIÓN DEL PROYECTO.....	2
ESTADO DEL ARTE.....	2
OBJETIVOS.....	2
DIAGRAMA DE FLUJO DEL SOFTWARE.....	3
DIAGRAMA DE GANTT.....	4
ALCANCE DEL PROYECTO.....	5
CUARTOS IMPLEMENTADOS Y SUS ELEMENTOS.....	5
ANIMACIONES IMPLEMENTADAS .....	7
LIMITANTES.....	7
ANALISIS DE COSTOS DEL MODELADO Y PRODUCCIÓN TÉCNICA .....	7
METODOLOGÍA DE SOFTWARE APLICADA.....	9
DOCUMENTACIÓN DEL CÓDIGO.....	23
CONCLUSIONES.....	38
REFERENCIAS.....	39

# **MANUAL TÉCNICO - PROYECTO FINAL.**

## **DESCRIPCIÓN DEL PROYECTO.**

Este proyecto consiste en la recreación tridimensional de una fachada ficticia inspirada en el universo de Hello Kitty, diseñada para representar un entorno colorido y amigable característico de su estética. El desarrollo se llevó a cabo utilizando OpenGL junto con el lenguaje de programación C++, empleando modelos creados en Blender, los cuales fueron exportados e integrados mediante la librería Assimp.

La escena está compuesta por cuatro áreas principales, cada una con un conjunto de objetos modelados, texturizados y en algunos casos animados, con el propósito de generar interactividad, dinamismo y coherencia visual dentro del entorno 3D. Sin embargo, en este manual técnico solo se describirán y explicarán dos de estas áreas: el pasillo principal y el cuarto de lavado, con el fin de profundizar en su estructura, componentes y funcionamiento dentro del proyecto.

Este proyecto tiene fines educativos y demostrativos, y busca mostrar la integración entre el modelado en Blender y la renderización en tiempo real a través de OpenGL.

## **ESTADO DEL ARTE.**

El desarrollo de entornos tridimensionales interactivos ha experimentado un crecimiento significativo en las últimas décadas, impulsado por avances en hardware gráfico, técnicas de renderizado en tiempo real y herramientas de modelado cada vez más accesibles. Este proyecto se enmarca dentro del campo de la computación gráfica aplicada, donde convergen disciplinas como el modelado 3D, la programación de shaders, la gestión de iluminación dinámica y la creación de sistemas de partículas para efectos visuales.

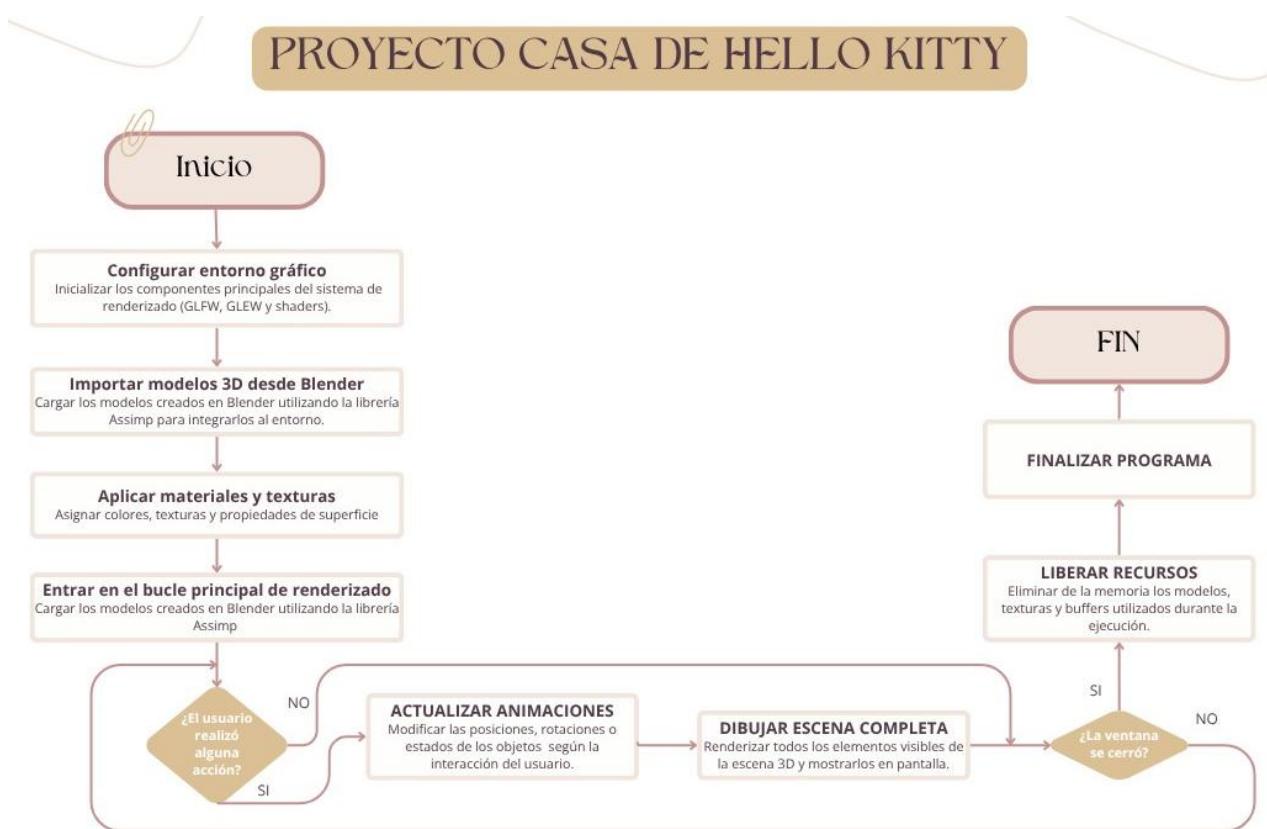
## **OBJETIVOS.**

Desarrollar una escena tridimensional completa y funcional inspirada en el universo de Hello Kitty, modelada en Blender e implementada en OpenGL mediante C++, integrando procesos de modelado, texturizado, animación e interacción, y aplicando de manera práctica los conocimientos adquiridos en las sesiones teóricas y de laboratorio de la materia de Computación Gráfica, con el fin de lograr un entorno visual coherente, dinámico y correctamente documentado a nivel técnico.

Los objetivos principales de este proyecto son los siguientes:

- **Modelar y desarrollar** una escena tridimensional completa y funcional inspirada en el universo de Hello Kitty, recreando un entorno visual atractivo y temático.
- **Aplicar técnicas y herramientas de modelado 3D profesional utilizando Blender**, garantizando una correcta topología, texturizado y exportación de los modelos.
- **Implementar animaciones e interacciones** mediante OpenGL y C++, con el fin de dotar a la escena de dinamismo y comportamiento visual coherente.
- **Elaborar la documentación técnica** correspondiente al desarrollo, estructura y funcionamiento del software, asegurando su correcta comprensión y mantenimiento futuro.

## DIAGRAMA DE FLUJO DEL SOFTWARE.



El diagrama muestra el flujo de ejecución del **software interactivo 3D “Casa de Hello Kitty”**, desde su inicio hasta el cierre del programa. El proceso comienza con la **configuración del entorno gráfico**, donde se inicializan los componentes principales de renderizado, como **GLFW, GLEW y shaders**, necesarios para crear la ventana de visualización y gestionar los gráficos en tiempo real.

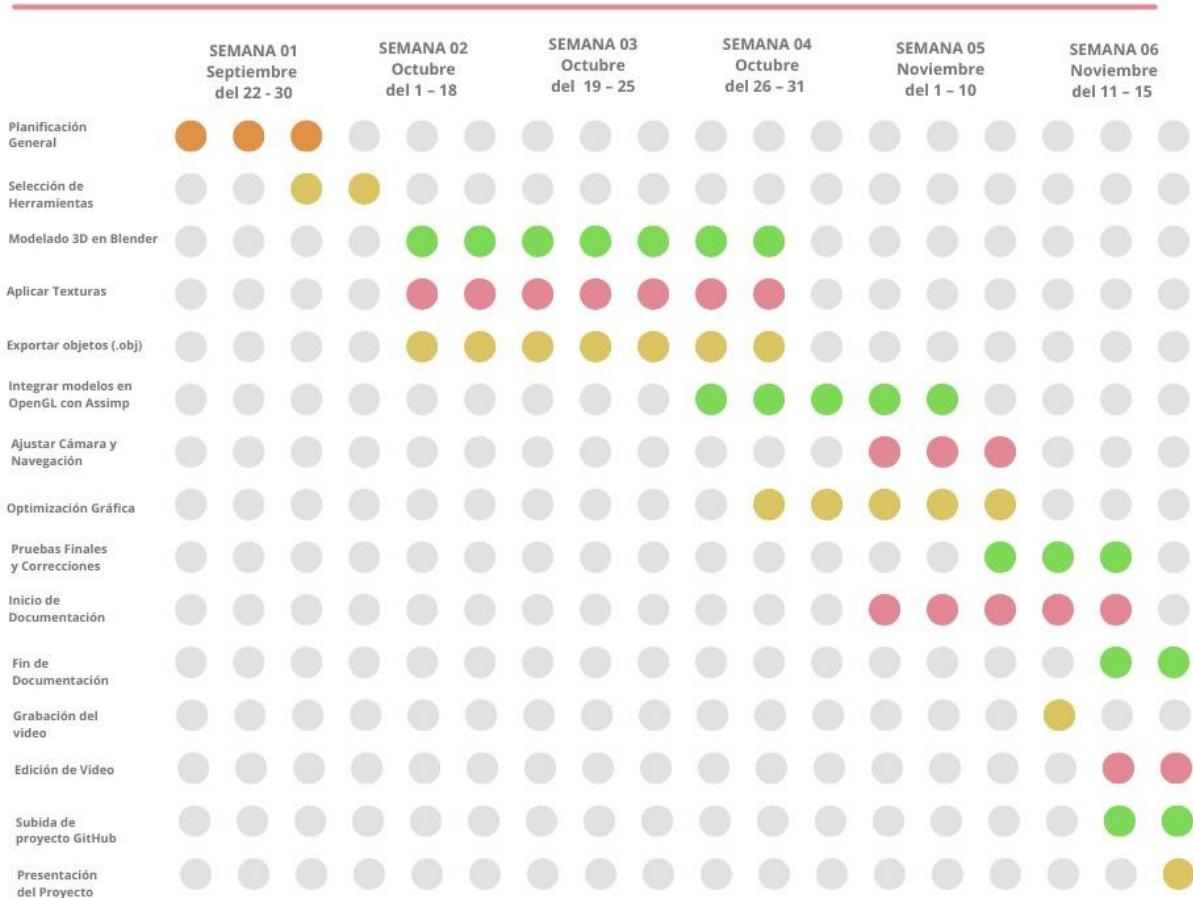
Posteriormente, el sistema **importa los modelos 3D creados en Blender**, integrándolos al entorno mediante la **librería Assimp**. A continuación, se **aplican los materiales y texturas** a los objetos, definiendo sus colores, superficies y propiedades visuales para darles una apariencia realista.

Una vez completada esta fase de preparación, el programa **entra en el bucle principal de renderizado**, donde se gestionan de forma continua los eventos, las animaciones y la visualización de la escena. Dentro de este bucle, el sistema evalúa si **el usuario realiza alguna acción** (como presionar teclas o interactuar con la cámara). Si ocurre, se **actualizan las animaciones** o los elementos dinámicos de la escena.

Después, el software procede a **dibujar la escena completa**, renderizando todos los modelos y objetos visibles con sus respectivas texturas e iluminación. Al finalizar cada ciclo, el sistema verifica si **la ventana ha sido cerrada** por el usuario. Si no es así, el bucle continúa; si se ha cerrado, el programa **libera los recursos utilizados** (como modelos, texturas y buffers) y **finaliza correctamente la ejecución**.

Este flujo garantiza una **experiencia interactiva fluida y optimizada**, combinando el modelado realizado en **Blender** con la potencia de **OpenGL** y **C++** para representar un entorno tridimensional inspirado en el universo de **Hello Kitty**.

# DIAGRAMA DE GANTT.



Este cronograma muestra la planificación de tareas clave para el desarrollo de un proyecto de software interactivo en 3D, distribuido en un periodo de seis semanas, desde el 22 de septiembre hasta el 15 de noviembre. Reflejando un flujo de trabajo progresivo, que abarca desde planeación inicial hasta la entrega y presentación final del proyecto.

## 1. Inicio del proyecto.

- Durante la **Semana 1 (22–30 de septiembre)** se realiza la **planificación general**, definiendo los objetivos, alcance, recursos, equipo y metodología del proyecto, estableciendo la base para el desarrollo posterior.

## 2. Preparación inicial.

- Entre las Semanas 1 y 2 (22 de septiembre al 18 de octubre) se realiza la selección de herramientas que se utilizarán a lo largo del proyecto. Esto incluye la elección del software de modelado (Blender), las librerías gráficas (OpenGL, Assimp), y las tecnologías auxiliares para texturizado y renderizado.

Con las herramientas definidas, se configuran los entornos de trabajo y se preparan los primeros archivos base.

- Paralelamente, se inicia el modelado 3D en Blender, comenzando con estructuras simples y bocetos tridimensionales de los elementos principales del entorno y los objetos del proyecto. **Modelado 3D en Blender**

## 3. Producción y texturizado 3D.

- Desde la Semana 2 hasta la Semana 4 (1 al 31 de octubre) se lleva a cabo una etapa de modelado 3D en Blender, donde se detallan las formas y características de los objetos.

- Simultáneamente, se desarrollan las tareas de **aplicación de texturas y materiales**, asignando colores, reflejos y propiedades superficiales que dan realismo a los modelos. Una vez finalizados, los modelos se **exportan en formato .obj** para facilitar su compatibilidad con OpenGL y la librería **Assimp**, encargada de importar los archivos dentro del motor gráfico.

#### **4. Integración y ajustes técnicos.**

- Durante las Semanas 4 y 5 (26 de octubre al 10 de noviembre) se realiza la integración de los modelos 3D en OpenGL mediante la librería Assimp. En este proceso se configuran las rutas de carga, se ajustan las coordenadas de textura, la iluminación y los shaders necesarios para la visualización en tiempo real.
- Posteriormente, se llevan a cabo los ajustes de cámara y navegación, definiendo los ángulos de visión, movimientos del usuario y los controles interactivos.
- También se ejecutan tareas de optimización gráfica, orientadas a mejorar el rendimiento del programa, reduciendo tiempos de carga y asegurando una experiencia fluida.

#### **5. Pruebas y Documentación.**

- Entre la **Semana 5 y la Semana 6 (1 al 15 de noviembre)** se realizan las **pruebas finales y correcciones** del proyecto, verificando el correcto funcionamiento y la estabilidad del entorno 3D.
- En paralelo, se elabora la **documentación técnica** y se crea el **video explicativo**, que muestra el resultado final del desarrollo.

#### **6. Cierre del Proyecto.**

- En la **Semana 6 (11–15 de noviembre)** se finalizan todas las actividades de cierre. El proyecto completo se **sube a la plataforma GitHub**, donde se alojan los archivos del código fuente, los modelos y la documentación.
- Finalmente, se prepara la **presentación del proyecto**, en la que se expondrán los resultados, se explicará el proceso de desarrollo, marcando el cierre oficial del proyecto.

## **ALCANCE DEL PROYECTO.**

- Escena completa de una casa temática de Hello Kitty con varias habitaciones y ambientaciones distintas.
- Integración de modelos elaborados en Blender y exportados en formato .obj.
- Implementación de animaciones interactivas: puertas, cortinas, cajones, tambor de lavadora, manecillas del reloj, etc.
- Sistema de iluminación dinámica con luz direccional y cinco luces puntuales.
- Efectos gráficos avanzados mediante sistemas de partículas (agua y vapor).
- Geometría nativa generada mediante OpenGL (silla, sillón, reloj y apagador).
- Renderizado en tiempo real con técnicas de iluminación Phong, blending, depth testing y shaders personalizados.
- Interacción total por teclado y mouse mediante una cámara de tipo primera persona (FPS).
- Carga de texturas, shaders y modelos desde estructura de carpetas organizada (Images, Models, Shader).

## **CUARTOS IMPLEMENTADOS Y SUS ELEMENTOS**

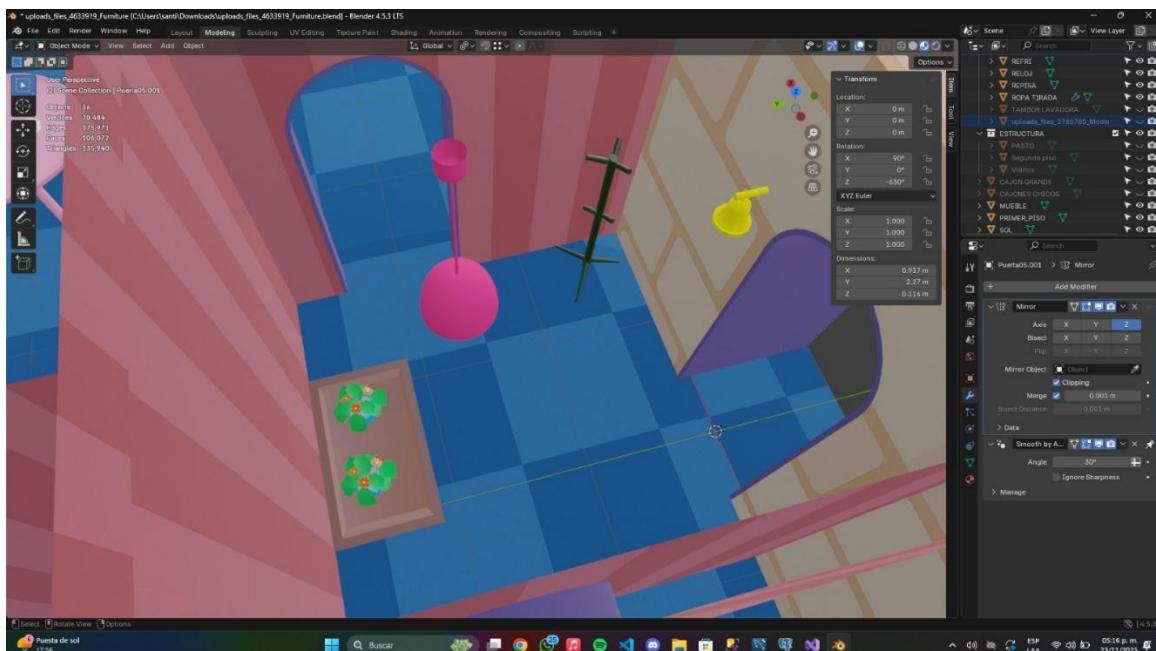
En esta sección se describen los cuartos desarrollados dentro de la escena 3D, así como los objetos modelados, texturizados e integrados en cada espacio. Se incluyen también las animaciones implementadas y los elementos interactivos definidos durante el desarrollo del proyecto.

## Cuarto 1: Recibidor

En esta área se integraron los elementos principales que componen el recibidor de la casa, todos modelados y texturizados en Blender, y posteriormente exportados para su implementación en OpenGL. Este espacio funciona como la entrada principal y contiene objetos decorativos y funcionales que aportan identidad visual al entorno.

### Elementos modelados e integrados:

- Campana
- Mueble dos puertas
- Perchero
- Floreros
- Flores



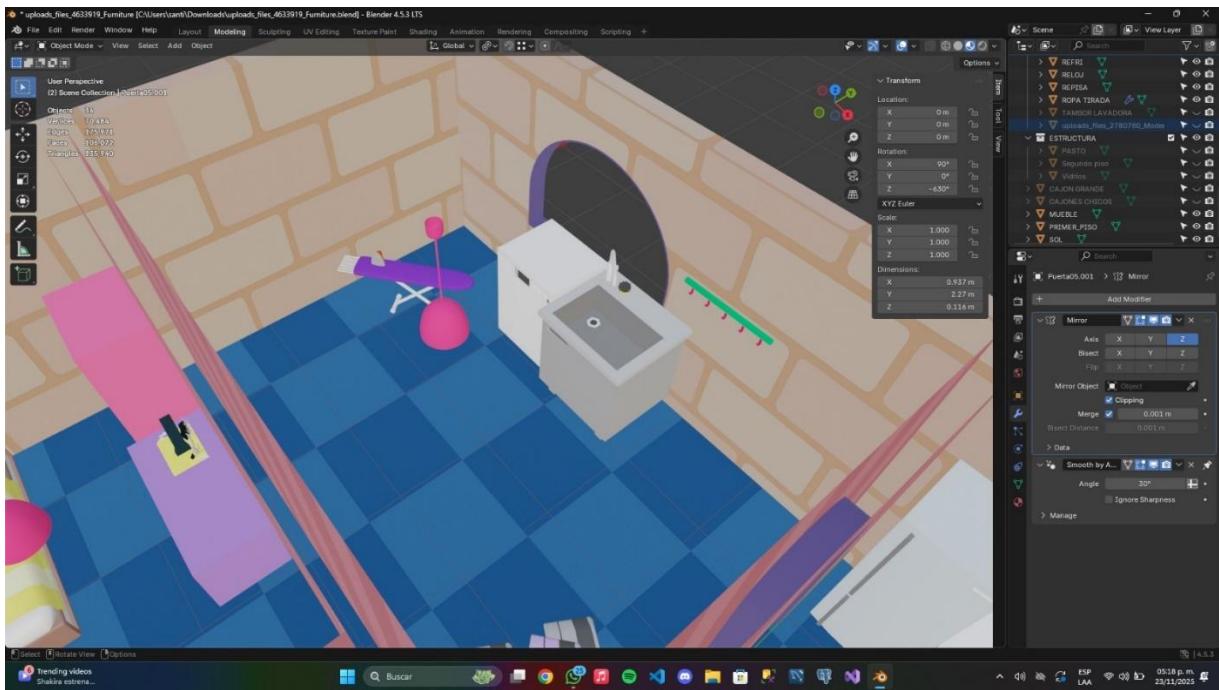
Imágenes Cuarto 01

## Cuarto 2: Cuarto de Lavado

El cuarto de lavado corresponde a la segunda habitación desarrollada dentro del proyecto. Este espacio integra objetos funcionales y elementos propios de un área doméstica destinada al lavado y planchado de ropa. Todos los modelos fueron creados en Blender, optimizados y posteriormente exportados para su implementación dentro del entorno 3D en OpenGL.

### Elementos modelados e integrados:

- Lavadora
- Lavabo
- Burro de planchar
- Plancha
- Perchero de pared



Imágenes Cuarto 02

## ANIMACIONES IMPLEMENTADAS

Dentro del entorno 3D se integraron distintas animaciones que permiten dotar de dinamismo e interactividad a los objetos presentes en las habitaciones desarrolladas. Cada una de estas animaciones fue programada en OpenGL mediante transformaciones aplicadas en tiempo real, controladas por variables internas y teclas asignadas dentro del sistema de entrada.

### ANIMACIONES DESARROLLADAS:

- Apertura y cierre de puertas
- Rotación del tambor de la lavadora
- Apertura y cierre de cortinas en las ventanas
- Plancha soltando vapor

### LIMITANTES.

- No había conocimiento previo en Blender.
- Aún no teníamos conocimiento de como texturizar en Blender antes de empezar esta parte.
- Recursos visuales limitados a fines académicos.
- No había conocimiento previo para utilizar Git y GitHub.
- El programa fue desarrollado y probado en Windows. Su ejecución en otros sistemas operativos como macOS o Linux podría requerir ajustes.
- No se permite recrear escenarios reales (UNAM, empresas).
- El desarrollo se realizó en plataformas compatibles con Windows.

## ANALISIS DE COSTOS DEL MODELADO Y PRODUCCIÓN TÉCNICA

### 1. Costos por producción técnica

#### a) Modelado y texturizado en Blender

Incluye la creación completa de la estructura, habitaciones, mobiliario y objetos animables.

- Modelado del entorno completo (fachada, interiores, objetos interactivos):
  - ↳ 18 horas aprox.

- Texturizado, organización de UVs y materiales:  
↳ **8 horas aprox.**
- Simulaciones físicas con *nCloth* (sábanas, tetera):  
↳ **4 horas aprox.**
- Total: **30 horas**

#### **b) Exportación y ajustes**

- Separación de objetos individuales
- Ajuste de pivotes para animaciones
- Verificación final de escalas, normales y orientación

Total: **6 horas**

#### **c) Integración en OpenGL**

- Implementación de shaders (Phong, partículas, materiales)
- Sistema de iluminación (direccional, puntual, spotlight)
- Configuración de cámara FPS
- Carga de modelos con Assimp
- Programación de animaciones (puertas, cajones, cortinas, tambor, vapor)
- Manejo de teclado/mouse y eventos
- Efectos visuales y ajustes de rendimiento

Total: **45 horas**

#### **d) Documentación y depuración**

- Incluye procesos de estructura, correcciones y documentación técnica.
- Organización de carpetas y parámetros base
- Corrección de errores visuales y de integración
- Elaboración de documentación técnica

Total: **12 horas**

## **2. Tiempo total invertido**

**30 + 6 + 45 + 12 = 93 horas totales aproximadamente**

Esta cifra engloba todo el desarrollo, pruebas, errores y aprendizaje.

## **3. Valor del conocimiento técnico**

Incluye habilidades combinadas de:

- **Artista Técnico 3D (Technical Artist):**  
modelado, UVs, texturas, estructura modular.
- **Programador Gráfico (Graphics Developer):**  
shaders, iluminación, transformaciones 3D, animación, eventos, partículas.

Tecnologías involucradas:

- **Blender**
- **OpenGL + C++**
- **GLFW, GLEW, GLM, Assimp**
- **Shaders GLSL, sistema de partículas**

El nivel técnico aplicado es equivalente al trabajo conjunto de dos perfiles profesionales.

## **4. Estimación del precio por hora**

Según estándares del mercado:

- Principiante / Estudiante: **\$14–20 USD/hora**
- Intermedio con conocimientos autodidactas (tu caso): **\$25–40 USD/hora**
- Profesional Freelance: **\$50–80 USD/hora**

Usando un valor justo: **\$25USD/hora × 93 horas = \$2,325 USD**

## **5. Ajuste por valor entregable**

El proyecto no consiste solo en modelos o código, sino en un entregable completo, que incluye:

- Modelado 3D integral
- Texturizado y UV mapping
- Integración total en OpenGL
- Motores de animación propios
- Sistema de iluminación programado
- Cámara FPS
- Efectos visuales y partículas
- Documentación técnica estructurada
- Organización modular escalable

Esto lo convierte en un **prototipo funcional**, comparable al desarrollo de una escena de videojuego.

Por lo tanto, se considera un rango profesional:

**Valor final recomendado del producto:**

**\$2,300 – \$2,600 USD**

### **Conclusión del análisis de costos**

El proyecto representa un desarrollo técnico sólido que combina modelado 3D, programación gráfica y producción documental. Con un total estimado de 93 horas y un costo actualizado al 2025, el valor final del proyecto oscila entre \$2,300 y \$2,600 USD, reflejando con precisión la complejidad del trabajo realizado y la especialización requerida para construir un entorno interactivo completamente funcional.

## **METODOLOGÍA DE SOFTWARE APLICADA.**

Para el desarrollo de este proyecto se adoptó una metodología incremental y colaborativa, alineada con los tiempos y requerimientos del curso. Este enfoque permitió avanzar mediante ciclos sucesivos de diseño, modelado, programación, integración y prueba, garantizando que cada iteración añadiera nuevas funcionalidades sin comprometer la estabilidad del sistema.

### **1. Planificación inicial.**

#### **1.1 Selección de la temática**

La temática elegida para el proyecto fue una casa inspirada en el universo **Hello Kitty**. Esta decisión se tomó debido a las características visuales del concepto: un estilo colorido, armónico, reconocible y con un atractivo estético adecuado para un entorno 3D.

Su diseño permite integrar elementos decorativos y funcionales que se prestan naturalmente al modelado tridimensional, además de ofrecer oportunidades para implementar animaciones y efectos de manera coherente.

#### **1.2 Definición del contenido esencial.**

Durante las primeras sesiones de trabajo se estableció que la escena debía incluir:

- Cuatro habitaciones principales.
- Elementos estructurales, mobiliario y objetos decorativos.
- Objetos animables para interacción (puertas, cortinas, cajones, tambor, manecillas).
- Sistema de partículas representando el vapor.

Estos componentes fueron seleccionados debido a que aportaban valor visual al interior de la casa y permitían implementar animaciones pertinentes dentro del proyecto.

### **2. Diseño y modelado 3D.**

El diseño y modelado 3D del proyecto se realizó en Blender, tomando como base un conjunto de referencias visuales y bocetos que permitieron establecer un estilo coherente con la estética colorida y

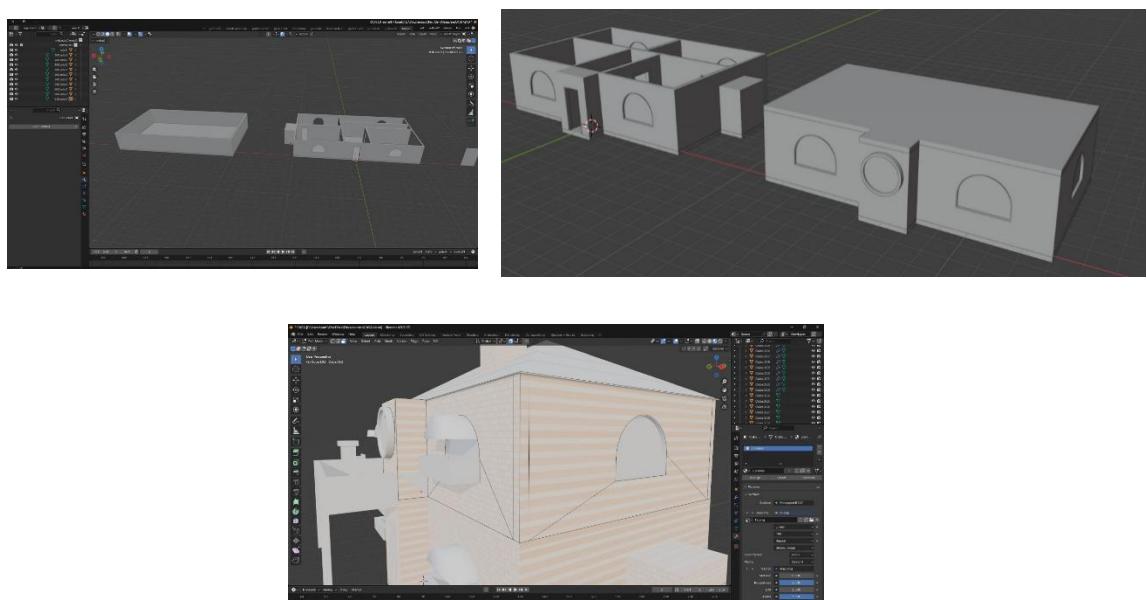
distintiva de Hello Kitty. A partir de estas referencias se construyó primero la estructura general de la casa y, posteriormente, cada uno de los objetos que conforman las habitaciones.

El proceso incluyó la creación de modelos optimizados, coherentes en escala y adecuados para su exportación en formato .obj, facilitando así su correcta integración en el entorno programado con OpenGL.

## 2.1 Construcción Inicial de la Fachada.

La estructura principal se modeló a partir de un cubo inicial, al cual se le aplicaron extrusiones para formar las paredes exteriores y los volúmenes de la entrada. Sin embargo, esta primera versión presentó problemas relacionados con el grosor de las paredes: al haber sido generadas mediante múltiples extrusiones, sus normales y UVs quedaron desalineadas.

Esta situación provocó que, al intentar aplicar texturas por primera vez, estas se deformaran o aparecieran proyectadas de manera incorrecta sobre las superficies.

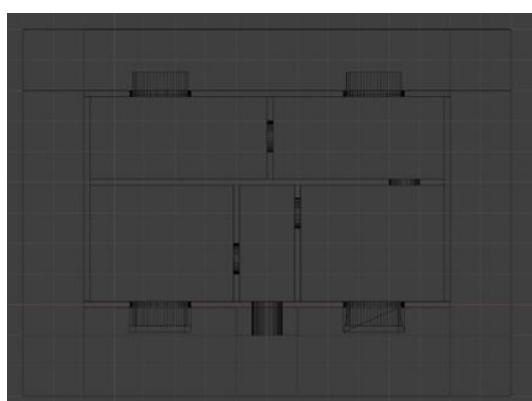


Imágenes iniciales de la estructura de la casa.

## 2.2 Solución Implementada: Remodelado de la Estructura.

Estos defectos hicieron evidente la necesidad de reconstruir la estructura desde cero para garantizar un flujo de trabajo limpio y compatible con el sistema de UV mapping.

Entonces se realizó la reconstrucción de desde un cubo base, esto para asegurar una topología uniforme.



Imagenes Ensamble inicial de elementos básicos de la casa.

Para esto se siguió el siguiente proceso:

1. **Creación del cubo inicial** como volumen base de la casa.
2. **Ingreso al modo Edición** para trabajar directamente sobre la geometría.
3. **Activación de la opción *Correct Face Attributes***, lo cual garantizó que futuras extrusiones conservaran la orientación y atributos de las caras sin deformaciones.
4. **Delimitación de muros, aperturas y niveles**, utilizando cortes controlados (Loop Cuts) y extrusiones limpias.
5. **Construcción de marcos de puertas y ventanas** empleando extrusiones inversas, cortes verticales y control de proporciones desde vista ortográfica.
6. **Revisión constante de la topología**, asegurando un flujo de polígonos sencillo que facilitaría el UV mapping posterior.

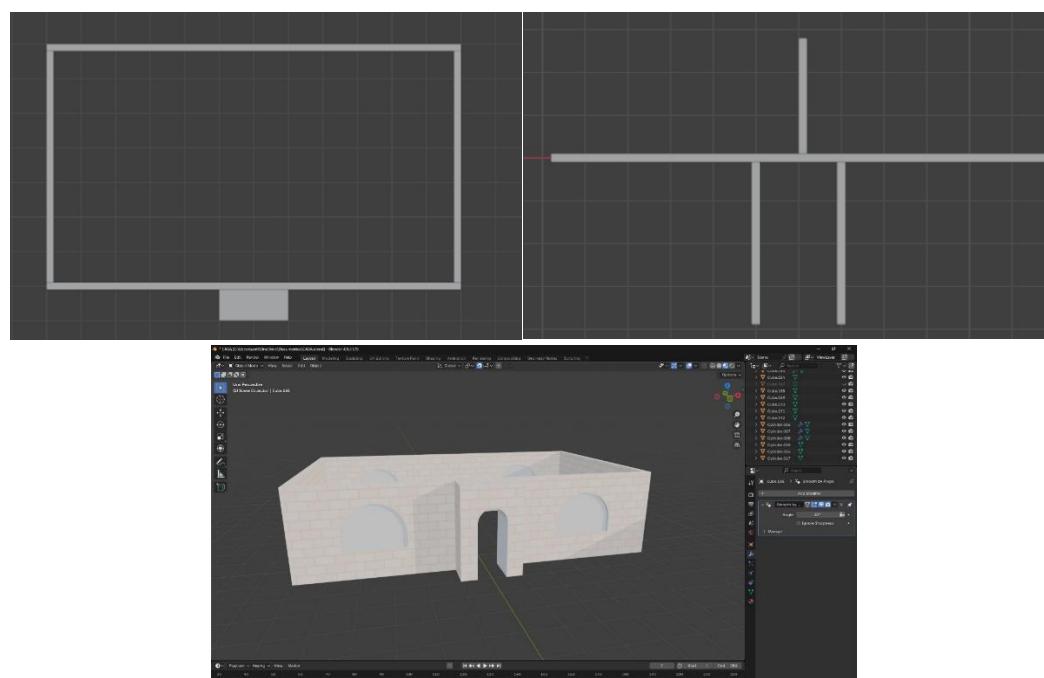
Este procedimiento permitió construir una estructura sólida, topológicamente coherente y libre de los problemas detectados en las primeras versiones.

### 2.3 Desarrollo de la Fachada y Muros Interiores.

Tras la reconstrucción de la estructura principal, se procedió a modelar nuevamente los elementos fundamentales de la vivienda. Se generaron:

- Las fachadas frontales, incluyendo la puerta principal, a partir de extrusiones limpias sobre el volumen base.
- Las ventanas curvas, creadas mediante extrusión controlada y la aplicación de bevels suaves para obtener bordes redondeados y una silueta uniforme.
- Los muros interiores, definidos inicialmente desde un plano general en vista superior (top view).

Para establecer la distribución de los cuartos, se trazaron líneas guía en vista superior, las cuales posteriormente se convirtieron en volúmenes mediante extrusiones verticales. El criterio principal para esta distribución fue que los cuatro cuartos de la casa conservaran proporciones similares en cuanto a espacio útil, aunque no necesariamente compartieran la misma forma. Esto dio lugar a la siguiente organización interna.



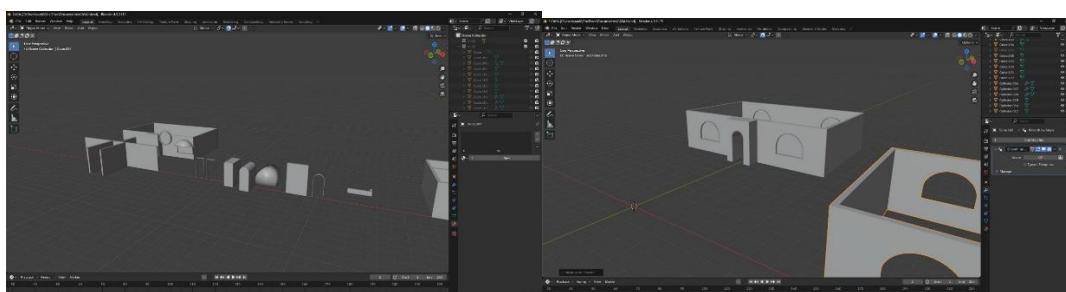
Imágenes Vista de la casa tras su reconstrucción con texturas base y distribución de habitaciones.

## 2.4 Construcción de Ventanas y Marcos.

El modelado de puertas y ventanas se realizó siguiendo un enfoque modular para asegurar precisión geométrica y facilitar su posterior texturizado y animación. Primero se generaron los huecos correspondientes en las paredes mediante cortes controlados y extrusiones limpias, manteniendo una topología sencilla que evitara distorsiones.

### 2.4.1 Ventanas.

Las ventanas se construyeron a partir de cubos básicos, ajustando su forma al contorno del hueco en el muro. Para las ventanas curvas se añadieron cortes adicionales y, cuando fue necesario, un *bevel* para suavizar los bordes. Cada marco fue escalado y alineado con precisión dentro de su abertura.



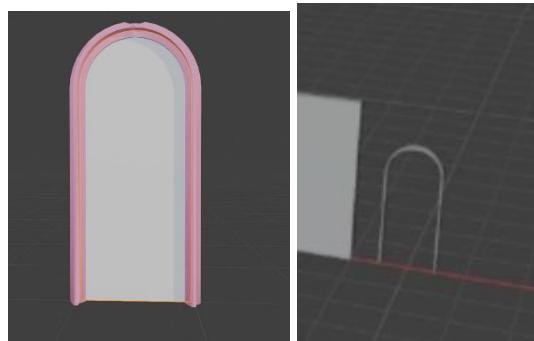
Imágenes Vista de la casa tras su reconstrucción con texturas base y distribución de habitaciones.

### 2.4.2 Modelado de la Puerta.

La puerta base se modeló a partir de un cubo, ajustando sus dimensiones hasta obtener la forma final que serviría como plantilla. Esta puerta funcionó como modelo maestro, ya que todas las demás puertas de la vivienda se generaron como duplicados derivados (puertas hijas). Esto garantizó que cada puerta mantuviera una geometría consistente y una escala uniforme en toda la escena.

Para integrar la puerta en los muros, se aplicó un modificador booleano sobre la pared correspondiente, recortando el hueco exacto donde la puerta se insertaría. Al utilizar una única puerta como modelo principal, fue posible optimizar el proceso de modelado y reducir el peso total de la escena.

Durante la preparación para exportar, la puerta original fue llevada al origen para alinear correctamente su pivote, el cual se colocó en el borde de giro, y asegurar una animación adecuada dentro de OpenGL. Los duplicados conservaron esta configuración, facilitando su integración en los distintos espacios interiores.



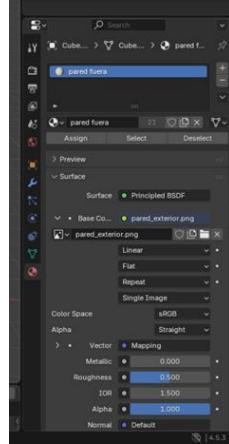
Imágenes Vista de la casa tras su reconstrucción con texturas base y distribución de habitaciones.

## 2.5. Aplicación de Texturas

Una parte esencial del proyecto fue la asignación de texturas. El proceso de texturizado se realizó íntegramente en Blender, utilizando materiales basados en imágenes y configuraciones de UV mapping adaptadas a las características de cada superficie. Este procedimiento se aplicó tanto a la estructura principal de la casa como a los objetos adicionales exportados posteriormente hacia OpenGL.

Para aplicar una textura a cualquier elemento del modelo:

1. Selección del objeto desde el Viewport.
2. En el panel derecho, dentro de Material Properties, se añadió un nuevo material.
3. En el apartado Base Color, se seleccionó Image Texture y posteriormente la imagen correspondiente.
4. Una vez cargada, la textura se ajustó desde el panel de UV Editing para asegurar el mapeo correcto.



Imágenes de Asignación de la textura desde el parámetro *Base Color* en el panel de materiales.

#### 2.5.1 Ajuste del mapeo mediante nodos.

Una vez cargada la imagen, se accedió al editor de nodos (*Shading Workspace*). Blender únicamente genera tres nodos por defecto (Texture → Principled BSDF → Output), por lo que se añadieron los nodos:

- **Texture Coordinate**
- **Mapping**

Estos permiten controlar la orientación, repetición (tiling) y escala de la textura sobre cada superficie.

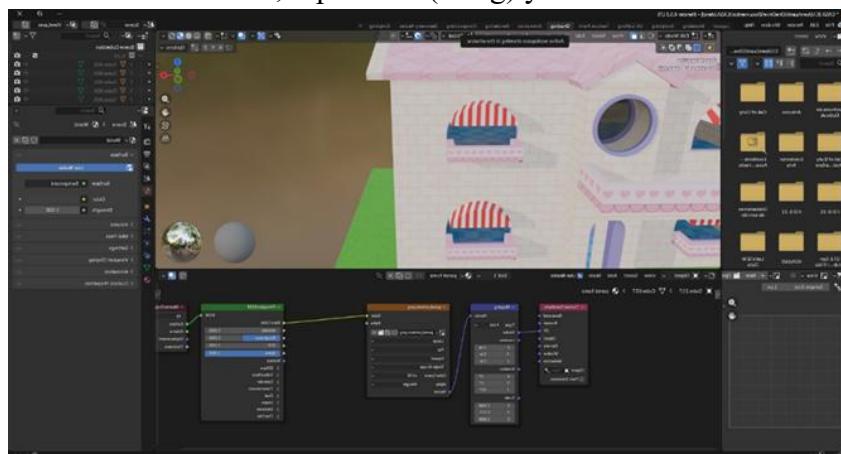


Imagen de Configuración básica de nodos utilizada para manipular las coordenadas de textura.

En este proyecto, fue necesario **rotar 90° en el eje Z** la textura de las paredes para alinear correctamente el patrón, además de ajustar los factores de escala para repetir la imagen la cantidad necesaria.

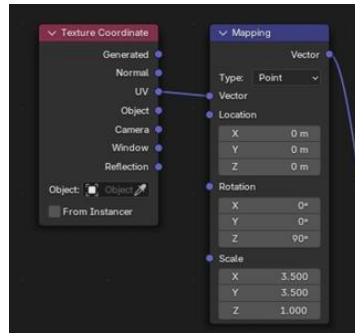


Imagen de Ajuste de rotación y repetición de textura mediante el nodo *Mapping*.

### 2.5.2 Texturizado específico de superficies curvas (toldos).

Los toldos presentan geometría semicurva, por lo que su proyección UV requería un procedimiento específico.

Primero se seleccionaba el objeto, y en modo edición (A para seleccionar toda la malla), se generaba una nueva UV utilizando alguno de estos métodos:

- **Project from View** (opción recomendada para toldos curvos)
- **Cylinder Projection**, alineando la proyección al objeto
- 

Posteriormente, en el editor UV se ajustaba la isla UV:

- se posicionaba sobre la textura de rayas,
- se escalaba hasta enderezar las líneas,
- se rotaba si era necesario,
- y se ajustaban proporciones hasta obtener un patrón limpio.

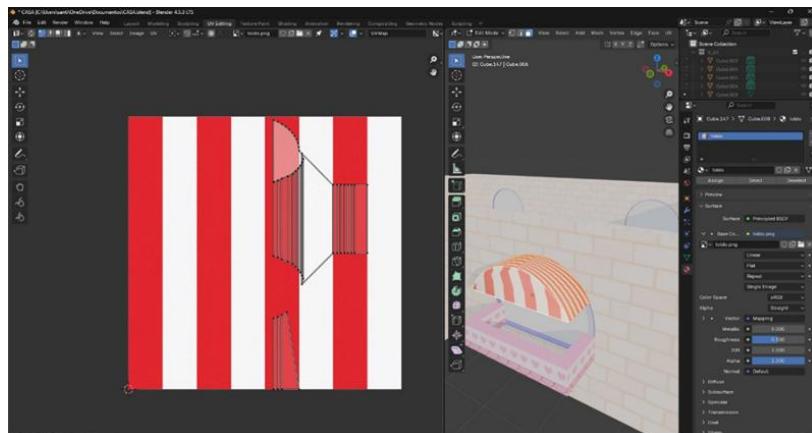


Imagen de Acomodo manual de la isla UV sobre la textura de rayas para corregir distorsiones.

Tras aplicar materiales, controlar tiling y reacomodar UVs en cada elemento, la casa adquirió un estilo visual coherente con la temática seleccionada (Hello Kitty). Las paredes, marcos, toldos y elementos decorativos muestran una alineación limpia y una correcta repetición de textura sin distorsiones.

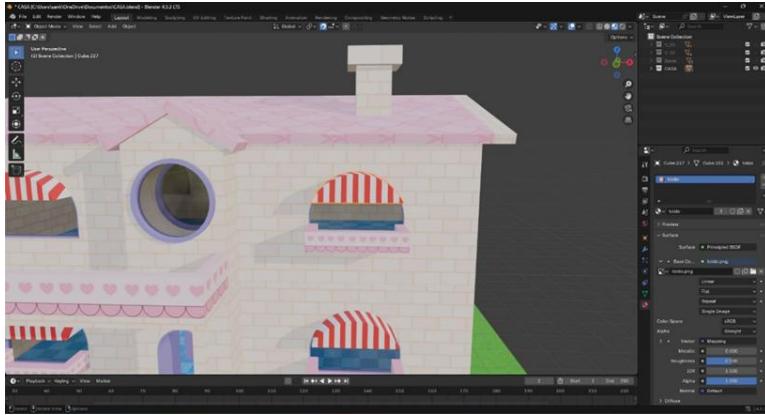
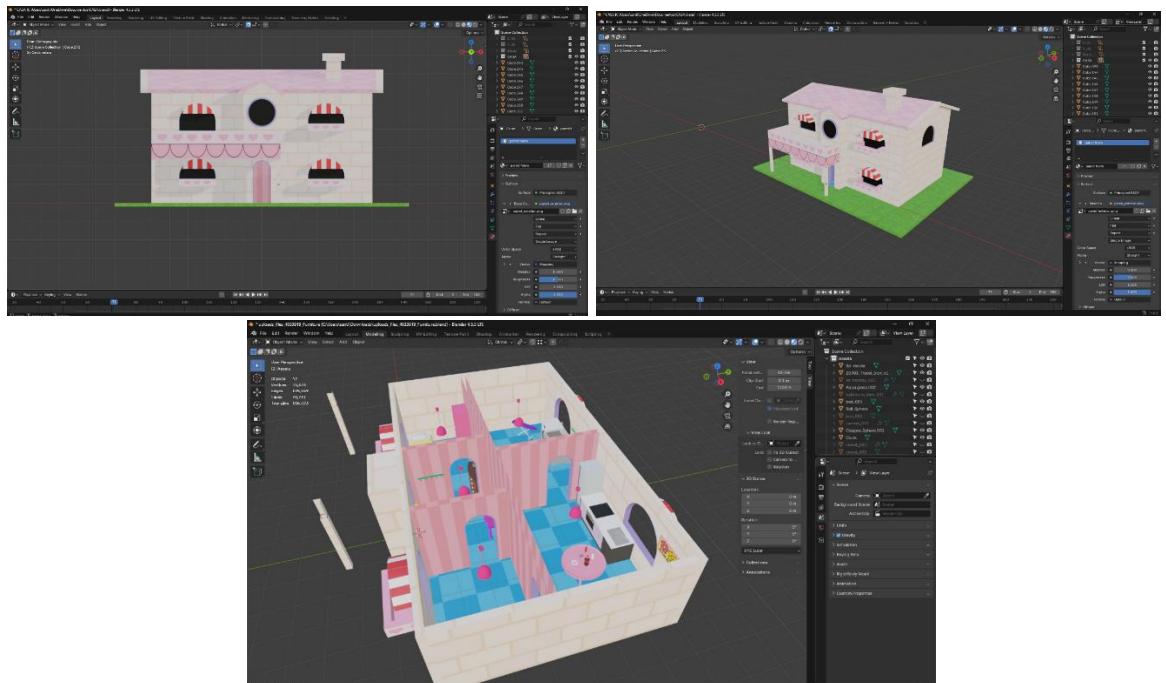


Imagen de Vista final de la fachada completamente texturizada en Blender.

## 2.6 Modelado y Texturizado del Piso.

El piso de la casa se obtuvo a partir de la misma estructura base utilizada para generar las paredes. Como punto de partida se empleó un cubo, cuya geometría fue modificada para conformar los muros del primer y segundo nivel. Durante este proceso, la cara inferior del cubo permaneció como una superficie plana continua, lo que permitió utilizarla directamente como el piso de la vivienda sin necesidad de crear un objeto adicional.

En las siguientes imágenes se aprecia la versión final de la casa y sus interiores, con todos los elementos modelados y texturizados integrados en su entorno.

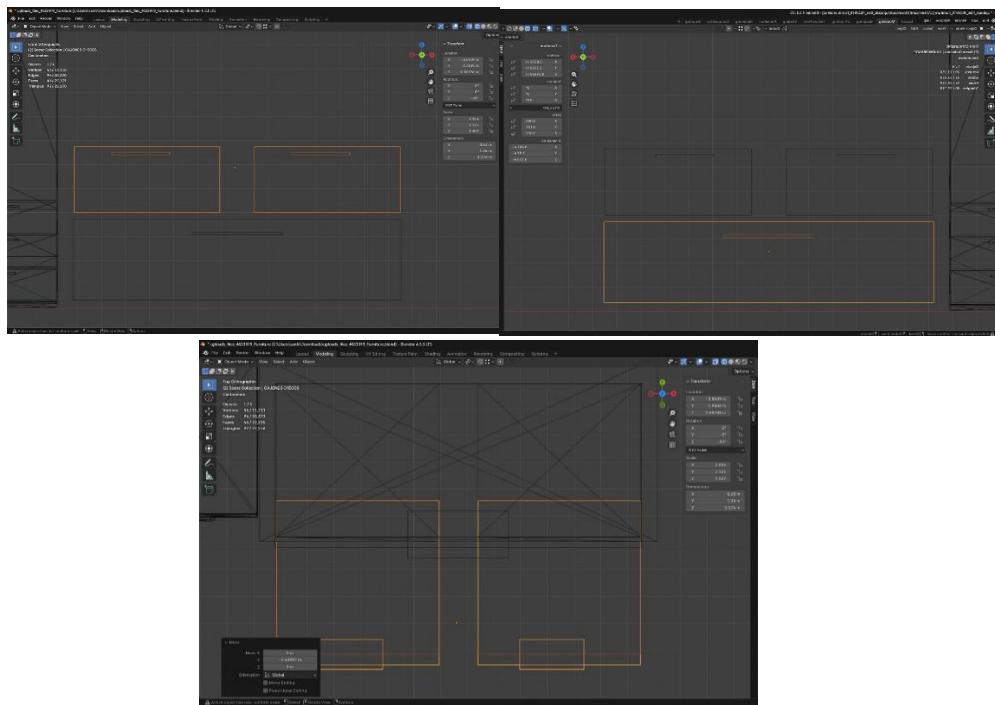


Imágenes finales de la casa.

Los objetos que se crearon para poder ambientar nuestra fachada fueron los siguientes:

El mueble fue construido principalmente a partir de cubos, ajustando sus dimensiones mediante transformaciones básicas para definir el cuerpo principal y la cubierta superior. Los cajones se modelaron como objetos independientes con el fin de permitir su posterior animación en OpenGL; para ello se extrajeron desde el volumen frontal utilizando Inset y Extrude, lo que proporcionó la profundidad necesaria para diferenciar cada sección móvil.

Las manijas se resolvieron como piezas simples basadas en cubos de muy baja altura, colocadas por separado para facilitar el movimiento de los cajones durante la animación. La estructura se organizó cuidadosamente en colecciones y pivotes, de modo que cada cajón pudiera abrirse correctamente hacia adelante sin deformaciones.



### 3. Exportación de Modelos 3D a Formato .OBJ

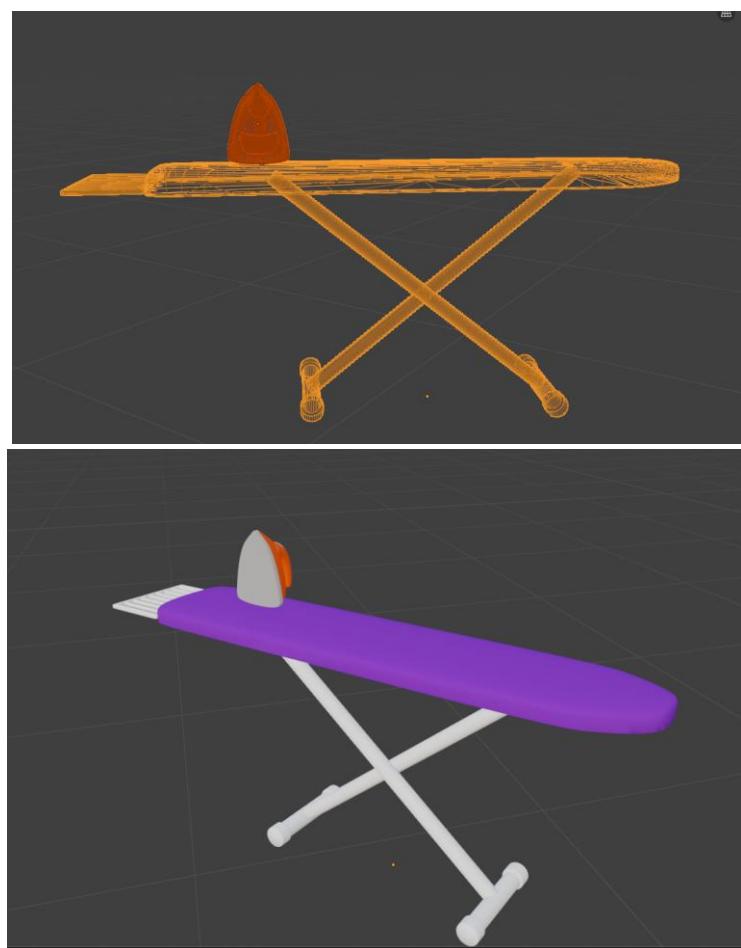


Imagen. Burro de planchar y plancha de ropa.

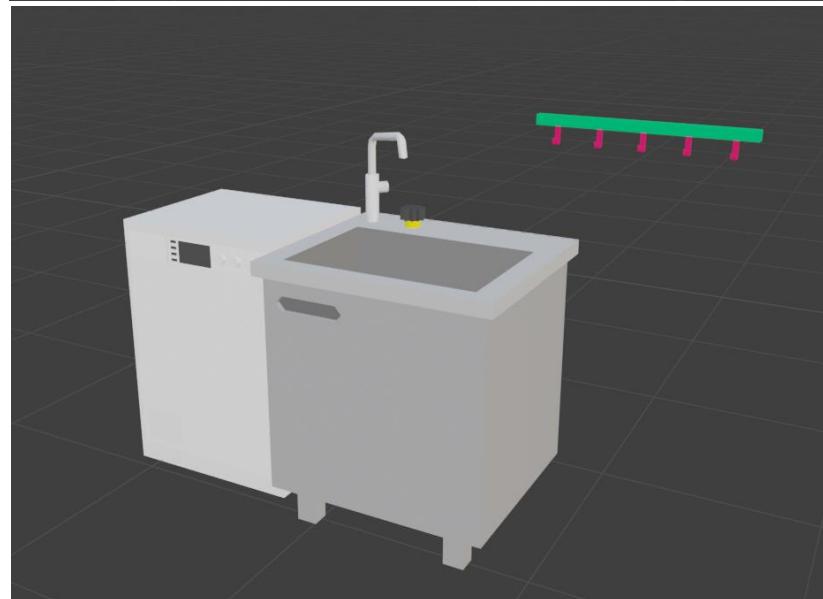
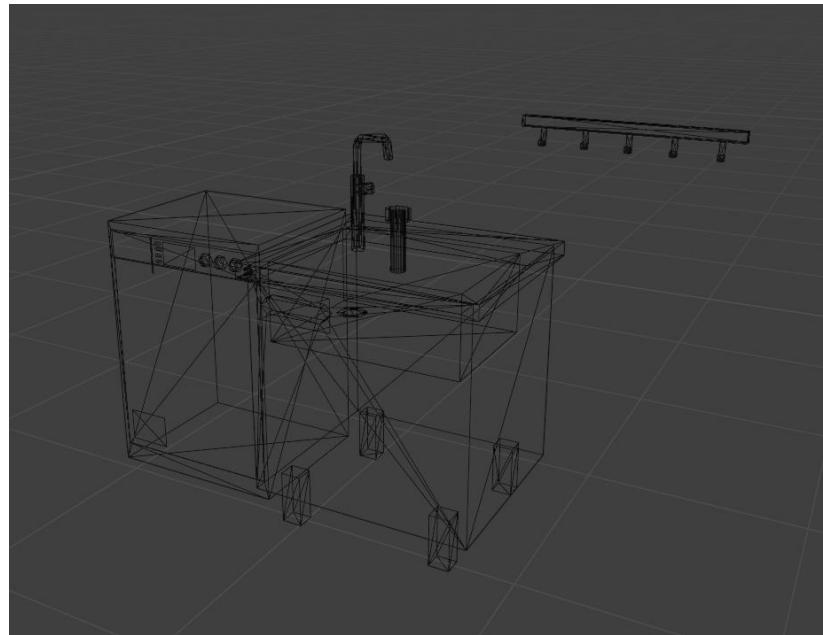


Imagen. Lavadora sin tambor, lavabo del cuarto de lavado y perchero de pared.

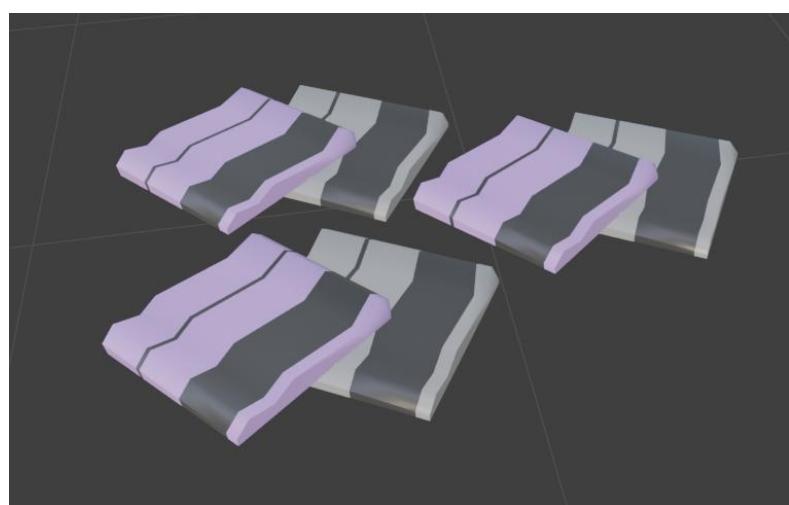


Imagen. Ropa tirada en el cuarto de lavado.

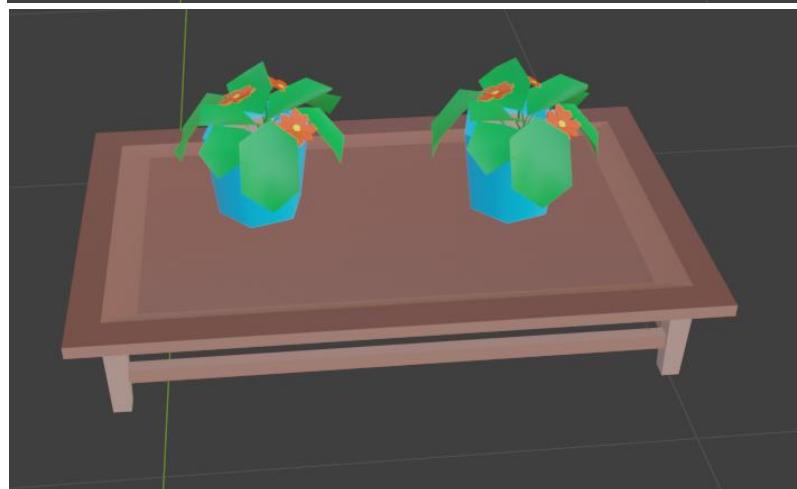
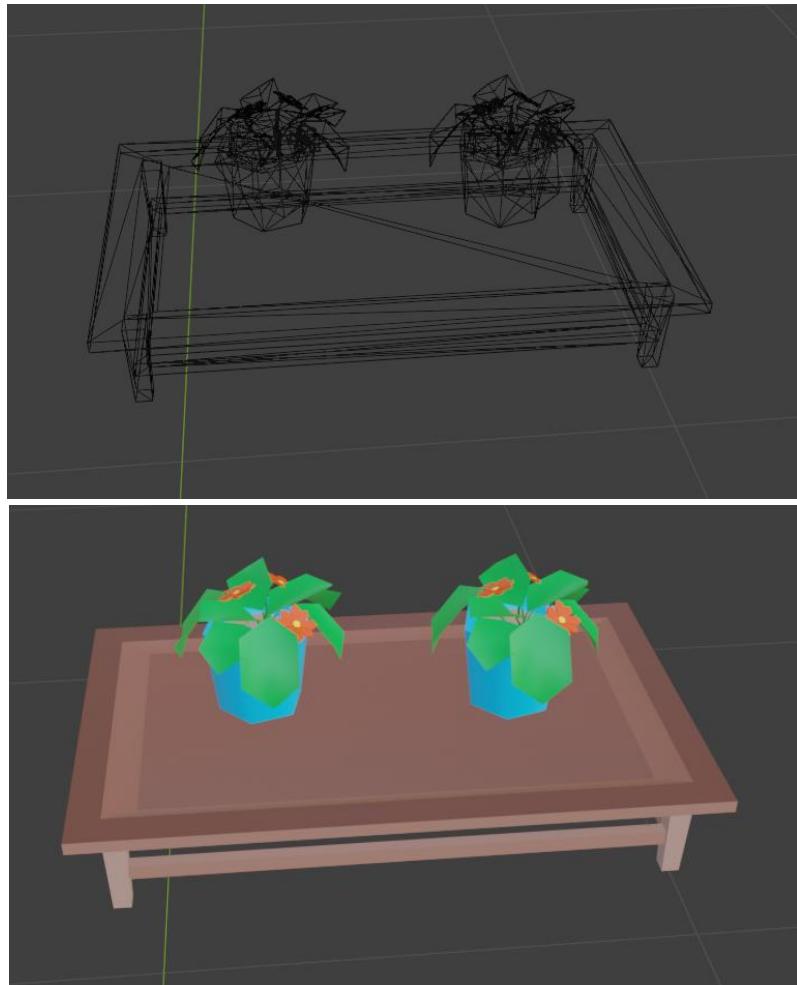
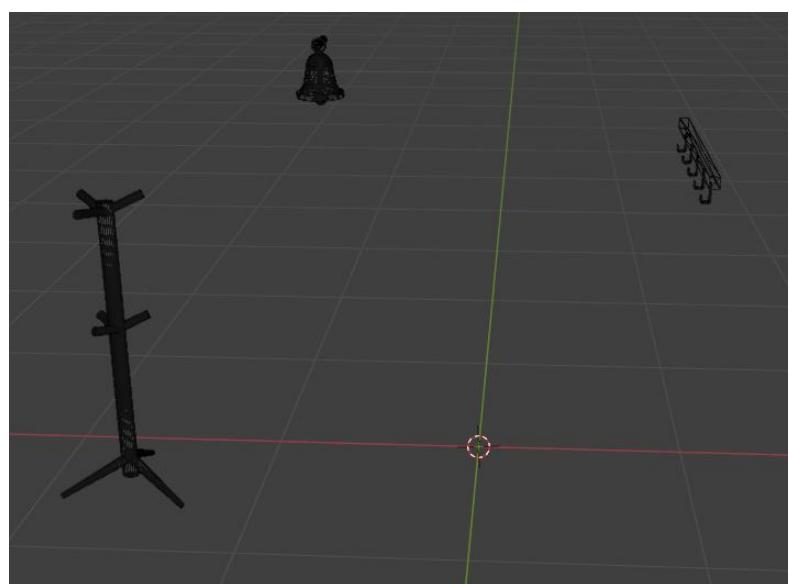


Imagen. Flores sobre mesa.



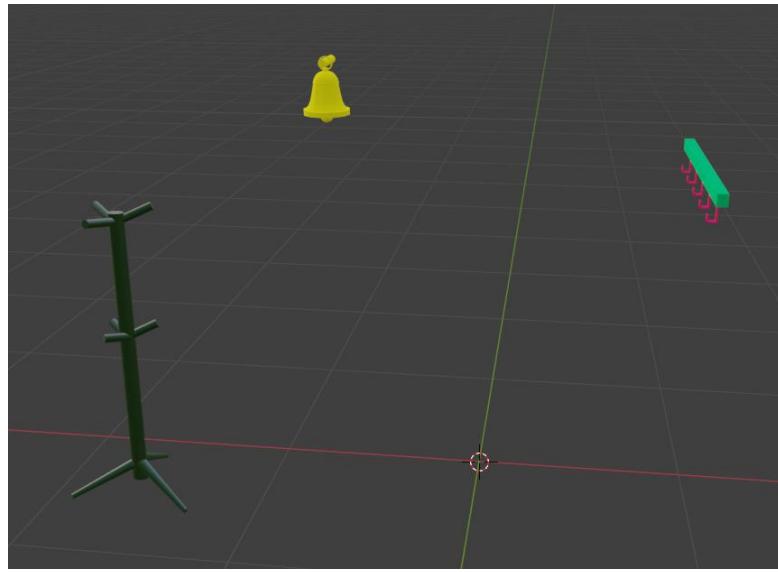


Imagen. Pechero, perchero de pared y campana.

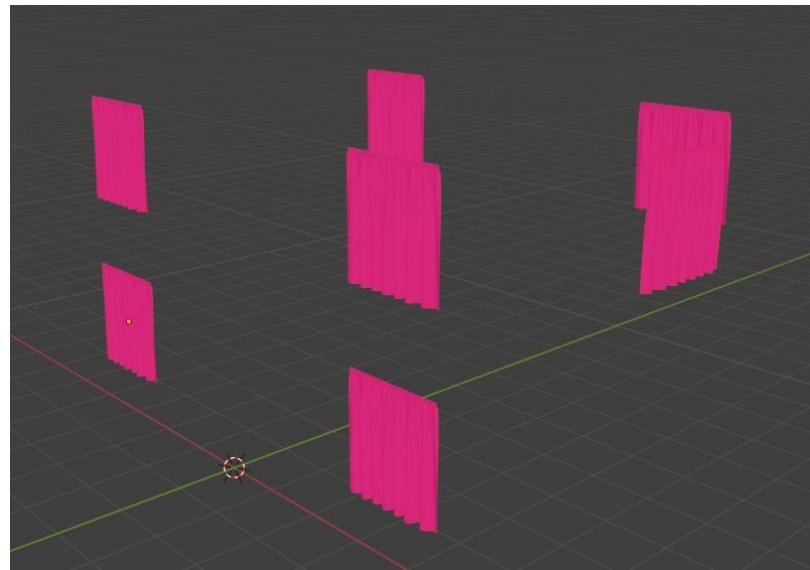
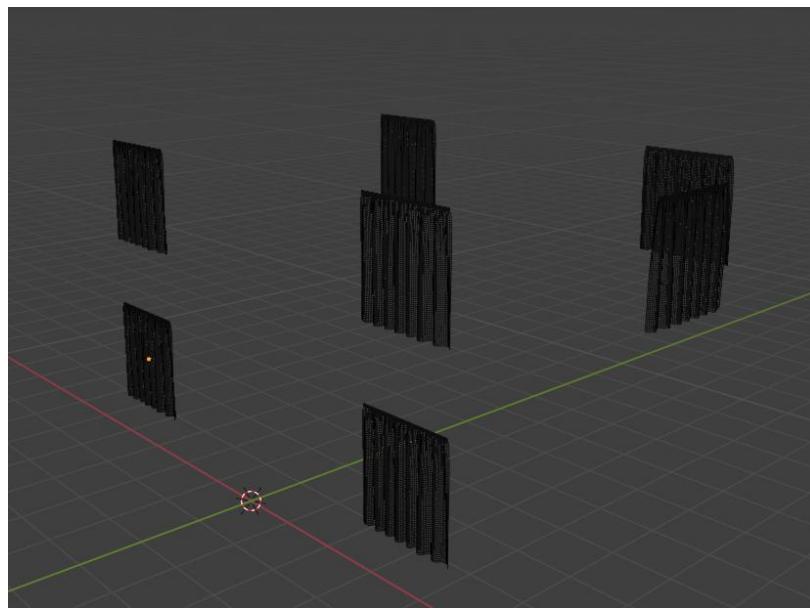


Imagen. Cortinas.

Los siguientes objetos se encuentran en el origen de nuestro archivo debido a que se debieron de trasladar a dicho punto para poder trasladarlos a su posición original y animarlos.

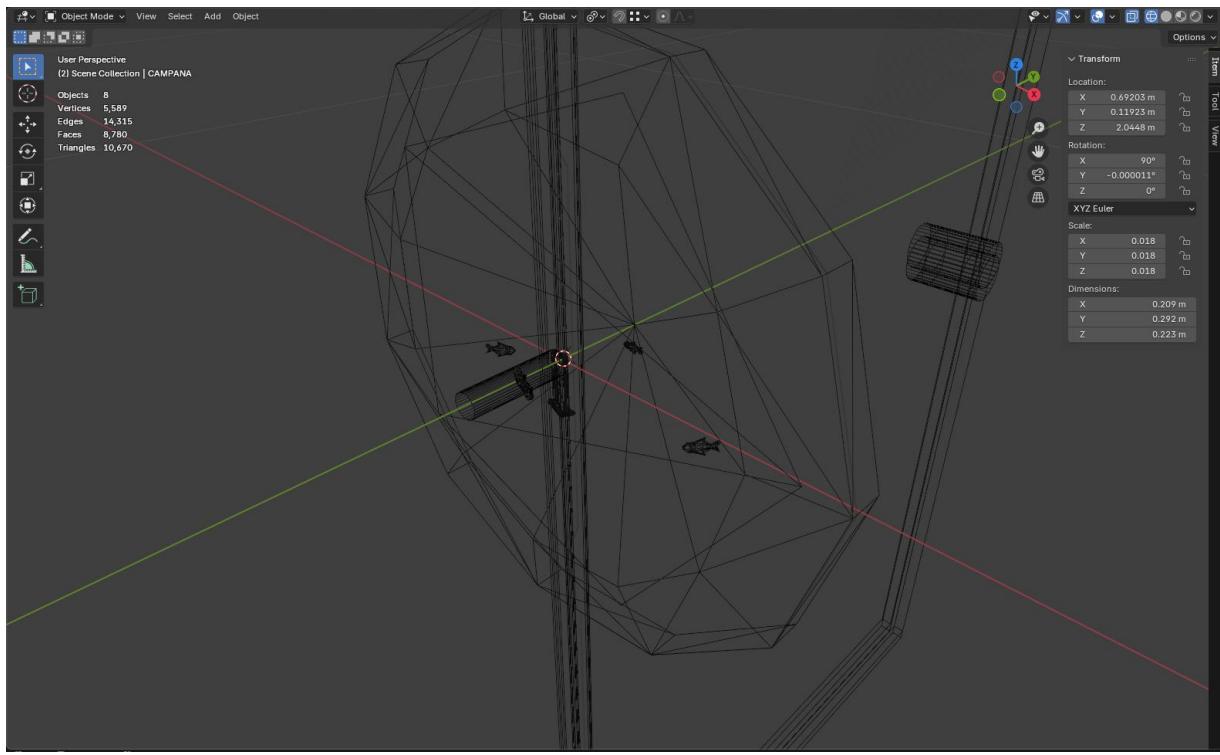


Imagen. Tambor de la lavadora, peces de la pecera, manecilla del reloj

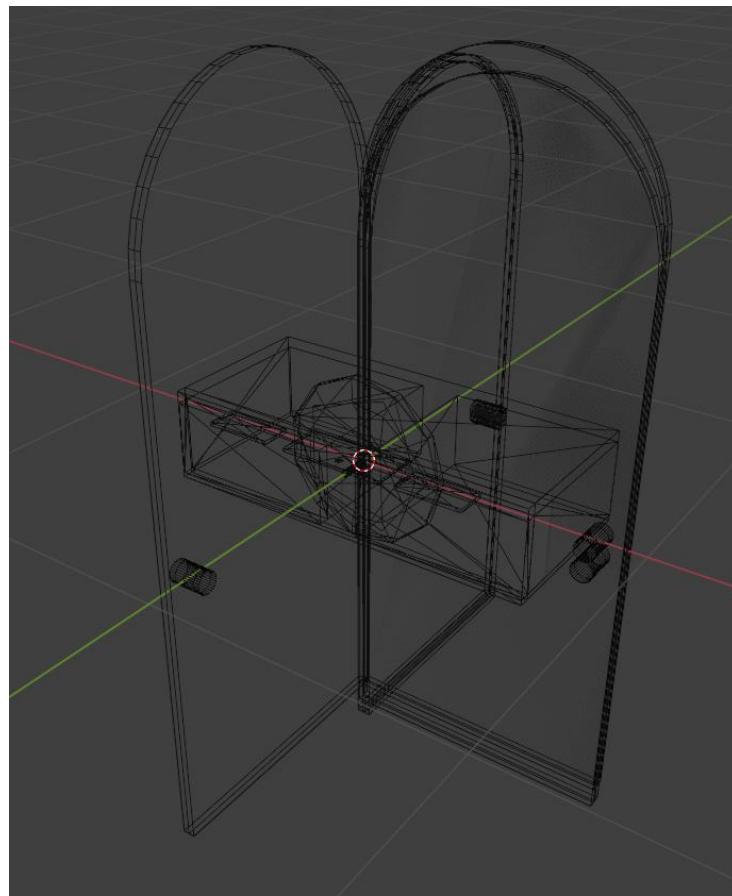


Imagen. Puertas de la casa y cajones.

Una vez finalizado el modelado de los objetos en Blender, se procedió a su exportación en formato .obj, elegido por su amplia compatibilidad con librerías de carga como Assimp y con la tubería de renderizado en OpenGL. Para asegurar una importación correcta durante la etapa de programación, se consideraron los siguientes puntos técnicos:

### **3.2 Geometría depurada:**

Antes de exportar, se revisaron todas las mallas para garantizar que no presentaran caras invertidas, normales inconsistentes o geometría no manifold. También se redujeron polígonos innecesarios y se consolidaron objetos cuando era útil para simplificar la estructura final.

### **3.3 Escala coherente:**

Cada modelo fue uniformado en escala para mantener proporciones correctas al integrarse en la escena general. Esto evitó transformaciones adicionales en OpenGL y aseguró que todos los objetos conservaran su tamaño esperado una vez cargados.

### **3.4 Pivotes y orientación correcta:**

Los pivotes fueron colocados en posiciones estratégicas, especialmente en objetos animables ya que en estos el pivote es necesario y se alinearon con los ejes globales para evitar desplazamientos inesperados durante las transformaciones en código.

### **3.5 Preparación para OpenGL:**

Previo a la exportación, se aplicaron transformaciones y se limpiaron historiales para garantizar archivos ligeros y consistentes.

## **4. Desarrollo incremental del software.**

### **Base del entorno gráfico.**

La construcción del entorno gráfico comenzó utilizando C++ junto con OpenGL 3.3 Core Profile, integrando bibliotecas esenciales como GLFW3, GLEW, GLM, STB\_IMAGE y un sistema de carga de modelos basado en Assimp. Una vez definidos los 44 modelos exportados desde Blender, se desarrolló la estructura base del motor gráfico encargado del renderizado en tiempo real.

Para la gestión de la ventana principal, el contexto de OpenGL y la captura de entradas del usuario, se utilizó GLFW3. Gracias a esto fue posible establecer una resolución fija de  $800 \times 600$  píxeles, activar el modo de cursor deshabilitado para navegación tipo FPS y registrar múltiples teclas destinadas al control de animaciones, iluminación y movimiento de cámara.

El acceso a las extensiones modernas del Core Profile se logró mediante GLEW, configurado en modo experimental para asegurar compatibilidad con shaders personalizados y con estructuras avanzadas como VBOs, VAOs y EBOs. Paralelamente, la biblioteca GLM proporcionó todas las funciones matemáticas necesarias para gestionar transformaciones espaciales, matrices de vista y proyección, rotaciones, escalados y operaciones trigonométricas utilizadas tanto en el sistema de cámara como en animaciones y partículas.

Para la carga de texturas se incorporó STB\_IMAGE, responsable de procesar imágenes PNG con soporte de canal alfa. Esta biblioteca permitió voltear las texturas verticalmente para adaptarlas al sistema de coordenadas de OpenGL y configurarlas adecuadamente para su uso en los sistemas de partículas y otros elementos que requerían transparencia.

Finalmente, el sistema de carga de modelos basado en Assimp, implementado mediante la clase Model.h, se encargó de importar los modelos .obj, leyendo sus geometrías, materiales y jerarquías internas. Esta infraestructura transformó los modelos en estructuras compatibles con OpenGL, incluso

aquellos compuestos por múltiples meshes o materiales, asegurando que sus propiedades de iluminación fueran procesadas correctamente.

## Importación de Objetos.

A partir de la base gráfica ya funcional en OpenGL, se comenzó la integración de los modelos 3D previamente creados y exportados desde Blender en formato .obj. Esta fase se realizó de manera progresiva mediante la clase Model, agregando los elementos uno por uno dentro del game loop para mantener el control sobre su posicionamiento, rotación y escalado a través de matrices de transformación glm::mat4. Cada modelo se configuró con su sistema de iluminación correspondiente utilizando shaders (lighting.vs/frag y lamp.vs/frag), aplicando transformaciones específicas mediante funciones glm::translate(), glm::rotate() y glm::scale(), y enviando estas matrices al pipeline gráfico con glUniformMatrix4fv () antes de renderizarse con el método Draw(). Los elementos animados, como puertas, cajones, cortinas y el tambor de la lavadora se implementaron mediante variables de control que modifican sus transformaciones en tiempo real dentro de la función Animation(), mientras que los objetos estáticos simplemente se renderizan con su matriz de transformación base, manteniendo así una estructura modular y organizada que facilitó el debugging y ajustes visuales durante el desarrollo.

Para este proceso se asociaron texturas y materiales a cada modelo mediante archivos .obj que incluyen referencias a sus materiales y mapas UV previamente configurados en Blender, mientras que para efectos especiales se cargaron texturas externas (GOTA.png y VAPOR.png en formato .png) mediante la librería stb\_image, aplicando filtros de textura con glTexParameter() y generando mipmaps con glGenerateMipmap() para optimizar el renderizado a diferentes distancias.

Se utilizaron tres tipos de shaders en GLSL para manejar distintos comportamientos visuales: el lightingShader implementa el modelo de iluminación Phong que calcula componentes ambiental, difuso y especular en cada fragmento con propiedades de material configurables (material.ambient, material.diffuse, material.specular, material.shininess); el lampShader renderiza objetos emisores de luz como los focos y el sol sin cálculos de iluminación; y el billboardShader maneja las partículas de agua y vapor con soporte para transparencia mediante blending alpha.

La iluminación se configuró mediante un sistema híbrido que incluye una luz direccional simulando el sol (controlable con tecla U) con propiedades ajustables según su estado de encendido/apagado, cinco luces puntuales posicionadas estratégicamente en diferentes habitaciones (cuarto, pasillo, cocina, lavandería) con atenuación cuadrática calculada mediante constantes constant, linear y quadratic, y un spotlight desactivado que simula una linterna desde la posición de la cámara, logrando así una iluminación dinámica y realista controlable en tiempo real.

## Verificación del funcionamiento.

Durante el desarrollo se realizaron revisiones periódicas para asegurar que los modelos se cargaran correctamente y que la escena se visualizara de forma adecuada. También se corrigieron de inmediato fallos relacionados con iluminación, interacción o animaciones.

## **Gestión del proyecto:**

En la parte final del proyecto se mejoró el rendimiento de los modelos y de toda la escena para que todo funcionara de manera más ligera sin perder calidad. También se preparó una documentación clara donde se explica cómo se hizo el proyecto y cómo usarlo, además de un video que muestra su funcionamiento.

Para organizarnos mejor usamos Git y GitHub, lo que nos permitió respaldar el código, llevar un control de los cambios y trabajar de forma coordinada.

Gracias a estas mejoras y a la organización del trabajo, el proyecto terminó siendo más estable, fácil de entender y con un mejor desempeño.

## **DOCUMENTACIÓN DEL CÓDIGO.**

A continuación, se presentan los archivos que conforman nuestro proyecto, junto con una breve explicación de la función que desempeña cada uno dentro del sistema.

### **Main.cpp:**

Este archivo es el núcleo principal del programa, responsable de inicializar el entorno gráfico OpenGL, cargar shaders y modelos 3D, gestionar las animaciones interactivas, implementar sistemas de partículas para efectos visuales, y renderizar la escena completa de una casa doméstica con múltiples habitaciones y objetos animados. Se usa C++ junto con OpenGL, GLFW, GLEW, GLM y la clase Model para carga de archivos .obj.

### *Bibliotecas utilizadas:*

- GLFW: Crea la ventana de la aplicación y gestiona la entrada del teclado y mouse.
- GLEW: Permite el uso de funciones modernas de OpenGL, permitiendo acceso a características como VAOs, VBOs, shaders programables y texturas avanzadas.
- GLM: Proporciona operaciones matemáticas para vectores y matrices.
- SOIL2 y stb\_image.h: Cargan texturas externas en formato PNG para efectos
- Shader.h, Camera.h, Model.h: Archivos personalizados que encapsulan funciones de shader, cámara y carga de modelos .obj.

### **Prototipos de Funciones.**

Las funciones del programa se agrupan en cinco categorías principales. Primero, están los callbacks de GLFW, encargados de gestionar la entrada del usuario mediante teclado y mouse. Después, las funciones de animación, que actualizan el movimiento y comportamiento de los objetos dinámicos en cada frame. También se incluye el sistema de partículas, responsable de los efectos visuales como el agua y el vapor. Además, se integran funciones de renderizado para objetos creados mediante geometría procedural, como el reloj y el apagador. Finalmente, se cuenta con un conjunto de funciones auxiliares que manejan buffers y el dibujo de primitivas.

Todas estas funciones trabajan de manera conjunta y organizada, permitiendo un sistema modular donde cada componente cumple una tarea específica, lo que facilita la lectura, mantenimiento y ampliación del código.

```

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);

```

## Dimensiones de Ventana, Cámara y Movimiento.

El sistema gráfico inicia creando una ventana fija de 800×600 píxeles, la cual no puede redimensionarse para mantener una relación de aspecto constante y evitar problemas en los cálculos de proyección. Esta ventana trabaja con OpenGL 3.3 Core Profile, con la opción de forward compatibility activada para asegurar funcionamiento en macOS. Además, el cursor se configura en modo deshabilitado (oculto y capturado), lo que permite mover la cámara en primera persona de forma continua sin verse limitado por los bordes de la pantalla.

```

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);

```

## Configuración de la cámara.

La cámara utiliza un modelo de movimiento en primera persona (FPS), iniciando desde una posición elevada ( $y = 5$ ) y ligeramente alejada ( $z = 15$ ) para obtener una vista general de toda la escena.

```

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

```

El proyecto trabaja con un sistema de coordenadas de mano derecha: Y apunta hacia arriba, X hacia la derecha y Z hacia el frente (con valores negativos alejándose de la cámara).

El giro de la cámara se controla con el movimiento del mouse, calculando la diferencia entre su posición actual y la anterior. Para evitar saltos bruscos en el primer movimiento, se usa la variable *firstMouse*, que estabiliza la lectura inicial del cursor.

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

## Sistema de movimiento y uso de DeltaTime.

El movimiento se gestiona mediante un sistema de doble buffer de estados de teclas que permite detectar múltiples teclas presionadas simultáneamente, esencial para movimiento diagonal y combinaciones de

controles. El sistema DeltaTime asegura que el movimiento sea independiente de la tasa de frames (FPS), multiplicando las velocidades por el tiempo transcurrido desde el último frame, garantizando así que un objeto se mueva la misma distancia en 1 segundo sin importar si el juego corre a 30 FPS o 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);
        }
    
```

## Sistema de Iluminación.

El sistema implementa iluminación Phong con tres tipos de fuentes: una luz direccional simulando el sol (controlable con tecla U), cinco luces puntuales en el techo de cada habitación con atenuación cuadrática (controlables con tecla L), y un spotlight desactivado reservado para linterna de cámara. Cada luz aporta componentes ambientales, difuso y especular que se suman para calcular el color final por fragmento.

```

// ===== 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);
}

```

El siguiente bloque de código corresponde al inicio de la función main() en un proyecto de C++ con OpenGL. Su propósito es configurar el entorno gráfico y preparar la ventana de renderizado.

- Inicializa GLFW y GLEW.
- Crea una ventana OpenGL con un contexto activo.
- Asigna funciones de entrada (teclado y mouse).
- Define el área de dibujo (viewport).
- Carga los shaders a usar para iluminación (lightingShader) y lámpara (lampShader).

Este bloque de código se encarga de inicializar la biblioteca **GLEW**, que permite acceder a las funciones modernas de OpenGL.

```

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

```

En esta parte del código se crean tres objetos Shader, cada uno asociado a un par de archivos GLSL.

- El lightingShader gestiona toda la iluminación de la escena (luces direccionales, puntuales y materiales).
- El lampShader se utiliza para dibujar objetos emisores de luz sin cálculos de iluminación.
- Finalmente, el billboardShader controla el renderizado de partículas mediante la técnica de billboarding, asegurando que siempre miren hacia la cámara.

```

// 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");

```

## Carga de modelos 3D.

En este bloque se inicializan todos los modelos utilizados en la escena. Cada línea crea un objeto Model y carga un archivo .obj desde la carpeta Models. Al hacerlo, Assimp importa automáticamente la geometría, materiales y jerarquía del modelo para que luego pueda renderizarse con OpenGL.

Los modelos incluyen la estructura principal de la casa (primer piso, segundo piso, ventanas, pasto), así como los elementos interactivos y animados (puertas, cortinas y focos). Cada objeto queda listo para ser dibujado dentro del ciclo de renderizado y manipulado mediante transformaciones, iluminación y animaciones.

```

// ===== 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_IQUIERDA.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");

```

### Animación del Vapor:

La animación del vapor se implementó mediante un sistema de partículas diseñado específicamente para simular el efecto visual generado por la plancha dentro del cuarto de lavado. Para ello, se inicializa un arreglo de partículas, donde cada una contiene datos esenciales para su comportamiento: posición, velocidad, tamaño, tiempo de vida y estado activo. Al iniciar el programa, todas las partículas permanecen inactivas y con tiempo de vida igual a cero, quedando listas para activarse cuando el usuario inicie la animación correspondiente.

Una vez que el vapor se activa, las partículas comienzan a generarse de forma progresiva, desplazándose hacia arriba con una velocidad suave y reduciendo gradualmente su tamaño hasta desaparecer, replicando el comportamiento natural del vapor liviano. Los parámetros de color, transparencia y atenuación se controlan desde el shader para lograr un efecto visual suave y coherente con el entorno.

El vapor utiliza una textura específica (VAPOR.png) aplicada mediante un sistema de billboards, que consiste en un plano que siempre mira hacia la cámara sin importar su orientación. Esto permite obtener un efecto volumétrico convincente sin necesidad de emplear geometrías complejas.

```

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

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, &nChannels, 0);

if (data)
{
    GLenum format = GL_RGB;
    if (nChannels == 3)
        format = GL_RGB;
    else if (nChannels == 3)
        format = GL_RGBA;
    else if (nChannels == 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 << "(" << nChannels << " canales)" << std::endl;
}
else
{
    std::cout << "Error al cargar textura: Images/VAPOR.png" << std::endl;
}
stbi_image_free(data);

```

La geometría del billboard está conformada por un cuadrado plano de vértices simples, acompañado de un conjunto de coordenadas de textura que definen la forma en que la imagen del vapor se mapea sobre cada partícula. Para su representación se utilizan un VAO, VBO y EBO, los cuales almacenan la estructura y el orden de los triángulos necesarios para renderizar cada partícula en pantalla.

```

// ===== 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
};

```

El billboard se dibuja usando dos triángulos definidos por un arreglo de índices:

```

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);

```

Para renderizarlo, se configuran un **VAO**, un **VBO** y un **EBO**, que almacenan la información de vértices y la estructura de los triángulos. Además, se habilitan dos atributos:

- **Atributo 0:** posición del vértice
- **Atributo 1:** coordenadas de textura

Este proceso permite que cada partícula se dibuje como un pequeño plano con una textura aplicada, manteniendo un rendimiento alto y un sistema visualmente consistente.

## Renderizado de la Escena (Dibujado de Objetos)

Objetos estáticos (sin animación)

Los objetos que no se mueven ni se rotan en tiempo real, como el primer piso, el pasto o el segundo piso, se dibujan con una matriz model identidad:

```

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

```

## Objetos animados por rotación: puertas y tambor.

En el caso de las puertas, además de posicionarlas, se aplica una rotación que depende de la variable de animación (por ejemplo 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 coloca la puerta en su posición dentro de la casa.
- rotate aplica la apertura/cierre alrededor del eje Y, usando rotPuerta01 que se va actualizando en la función de animación.

Algo similar pasa con el tambor de la lavadora, solo que gira en el eje Z.

## Objetos animados por translación: cortinas

Las cortinas forman parte de los objetos animados mediante translación dentro de la escena. Su movimiento se realiza exclusivamente sobre el eje X, utilizando dos variables independientes: posCortinaDerecha y posCortinaIzquierda, las cuales controlan la posición horizontal de cada cortina por separado.

Para gestionar este comportamiento, se utiliza una bandera que determina si la animación está activa (animCortinas) y otra que define si el movimiento corresponde a abrir o cerrar (abrirCortinas). Mientras la animación esté habilitada, cada cortina se desplaza a una velocidad constante definida por velocidadCortina, moviéndose en direcciones opuestas para simular la apertura o el cierre.

```

// ===== 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;
        }
    }
}

```

El sistema verifica continuamente si las cortinas han llegado a su posición límite. En el caso de apertura, ambas se desplazan hacia afuera hasta alcanzar POS\_FINAL\_CORTINA\_DERECHA y POS\_FINAL\_CORTINA\_IZQUIERDA. Para el cierre, se mueven hacia adentro hasta llegar a POS\_INICIAL\_CORTINA\_DERECHA y POS\_INICIAL\_CORTINA\_IZQUIERDA. Una vez alcanzado el punto objetivo, la animación se detiene automáticamente,

### Resto del mobiliario y objetos decorativos.

El resto de objetos (cama, roperos, mesa, repisa, pecera, libros, refri, etc.) se dibujan casi siempre con:

```

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

```

### Función Animation().

La función Animation() se encarga de actualizar todas las animaciones de la escena en cada frame, utilizando deltaTime para que los movimientos sean suaves e independientes de los FPS. Dentro de esta función se controlan tres tipos de elementos: animaciones de transformación (rotaciones y traslaciones).

Al inicio se calculan las velocidades base según el tiempo transcurrido entre frames:

```

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

```

Con esto, la apertura de puertas, el desplazamiento de cortinas se adapta automáticamente al rendimiento de la máquina.

Después se actualizan las animaciones continuas:

- El tambor de la lavadora (rotTambor) aumenta su ángulo de rotación mientras sus banderas (animTambor) este activo.
- Cuando el valor llega a 360°, se “reinicia” restando 360 para evitar que el ángulo crezca indefinidamente.

Las siguientes secciones controlan las animaciones bidireccionales (abrir/cerrar) de puertas y cortinas. Cada grupo tiene una bandera que indica si la animación está activa y otra que indica si el movimiento es de apertura o cierre:

- Puerta principal (rotPuerta01, animPuerta01, abrirPuerta01)
- Puertas internas (02 a 05) sincronizadas con animPuertasInternas y abrirPuertasInternas
- Cortinas (posCortinaDerecha, posCortinaIzquierda, animCortinas, abrirCortinas)

En cada caso, se aumenta o disminuye el valor (rotación o posición) hasta llegar a un límite definido, por ejemplo:

```
// ====== 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;
        }
    }
}
```

### Manejo de movimiento: DoMovement()

La función DoMovement() se encarga de mover la cámara según las teclas que el usuario mantenga presionadas. No revisa directamente los eventos de teclado, sino el arreglo global keys[], que se va actualizando en KeyCallback.

```
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);
    }
}
```

De esta forma, la cámara se puede mover hacia adelante, atrás y lateralmente usando WASD o las flechas, y el uso de deltaTime hace que la velocidad sea consistente sin importar los FPS.

### Manejo de teclado y toggles: KeyCallback()

KeyCallback() es la función registrada en GLFW para procesar cada evento de teclado. Aquí se resuelven dos cosas:

1. Acciones inmediatas (por ejemplo, cerrar la ventana con ESC).
2. Cambio de banderas booleanas que activan o desactivan animaciones o efectos, que luego se procesan en Animation().

```
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);
    }
}
```

Para las animaciones, el patrón general es:

- Detectar la tecla y el evento GLFW\_PRESS.
- Cambiar una variable booleana o indicar si se va a “abrir” o “cerrar” algo.

Lo que cambia es que el tambor de la lavadora usa un toggle simple, mientras que Las puertas y cortinas usan un toggle inteligente, que decide si se abren o se cierran según su posición actual.

Las teclas L, U, V activan o desactivan sistemas completos (agua, luces, sol, vapor) simplemente invirtiendo un booleano:

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

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

// ===== 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;
    }
}
```

## Control de la vista con el mouse: MouseCallback()

MouseCallback() controla la rotación de la cámara tipo FPS usando la posición del mouse:

```
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 evita un “salto” brusco en el primer movimiento.
- Se calculan los offsets en X y Y con respecto a la posición anterior.
- Estos offsets se envían a camera.ProcessMouseMovement(), que actualiza yaw y pitch.

## Shader.h

La clase Shader se encarga de cargar, compilar y activar los programas de sombreado necesarios para el renderizado en OpenGL. Cuando se crea un objeto Shader, el sistema lee desde archivo el código del vertex shader y del fragment shader, los compila verificando posibles errores y luego los enlaza dentro de un único programa ejecutable por la GPU. Una vez construido, el shader proporciona funciones para activarlo durante el dibujado (Use()) y para obtener ubicaciones de variables uniformes, como el parámetro color, que permite

enviar valores desde C++ hacia el shader. En resumen, esta clase automatiza todo el proceso técnico de gestión de shaders, facilitando su uso dentro del motor gráfico.

```

1  #ifndef SHADER_H
2  #define SHADER_H
3
4  #include <string>
5  #include <vector>
6  #include <iostream>
7  #include <fstream>
8
9  #include <GL/glew.h>
10
11 class Shader
12 {
13 public:
14     GLuint uniformColor;
15
16     // Constructor generates the shader on the fly
17     Shader(const std::string &vertexPath, const std::string &fragmentPath)
18     {
19         // 1. Retrieve the vertex/fragment source code from filePath
20         std::string vertexCode;
21         std::string fragmentCode;
22         std::ifstream vShaderFile;
23         std::ifstream fShaderFile;
24
25         // ensures ifstream objects can throw exceptions:
26         vShaderFile.exceptions(std::ifstream::badbit);
27         fShaderFile.exceptions(std::ifstream::badbit);
28         try
29         {
30             // Open files
31             vShaderFile.open(vertexPath);
32             fShaderFile.open(fragmentPath);
33             std::stringstream vShaderStream, fShaderStream;
34             // Read files' buffer content into streams
35             vShaderStream << vShaderFile.rdbuf();
36             fShaderStream << fShaderFile.rdbuf();
37             // close file handlers
38             vShaderFile.close();
39             fShaderFile.close();
40             // Convert stream into string
41             vertexCode = vShaderStream.str();
42             fragmentCode = fShaderStream.str();
43         }
44         catch (std::ifstream::failure e)
45         {
46             std::cout << "ERROR::SHADER::FILE_NOT_SUCCESSFULLY_READ" << std::endl;
47         }
48
49         //le damos la localidad de color
50         uniformColor = glGetUniformLocation(this->Program, "color");
51
52         // Delete the shaders as they're linked into our program now and no longer necessary
53         glDeleteShader(vertex);
54         glDeleteShader(fragment);
55
56         // Uses the current shader
57         void Use()
58         {
59             glUseProgram(this->Program);
60
61             GLuint getColorLocation()
62             {
63                 return uniformColor;
64             }
65
66         #endif
67
68         // Create the shaders
69         const GLchar *vertexCode = vertexCode.c_str();
70         const GLchar *fShaderCode = fragmentCode.c_str();
71
72         // 2. Compile shaders
73         GLint vertex, fragment;
74         GLint success;
75         GLint infoLogLength;
76
77         // Vertex Shader
78         vertex = glCreateShader(GL_VERTEX_SHADER);
79         glShaderSource(vertex, 1, &vertexCode, NULL);
80         glCompileShader(vertex);
81         // Print compilation errors if any
82         glGetShaderiv(vertex, GL_COMPILE_STATUS, &success);
83         if (!success)
84         {
85             glGetShaderInfoLog(vertex, 512, NULL, infoLog);
86             std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED" << infoLog << std::endl;
87         }
88
89         // Fragment Shader
90         fragment = glCreateShader(GL_FRAGMENT_SHADER);
91         glShaderSource(fragment, 1, &fShaderCode, NULL);
92         glCompileShader(fragment);
93         // Print compilation errors if any
94         glGetShaderiv(fragment, GL_COMPILE_STATUS, &success);
95         if (!success)
96         {
97             glGetShaderInfoLog(fragment, 512, NULL, infoLog);
98             std::cout << "ERROR::SHADER::FRAGMENT::COMPILATION_FAILED" << infoLog << std::endl;
99         }
100
101         // Shader Program
102         this->Program = glCreateProgram();
103         glAttachShader(this->Program, vertex);
104         glAttachShader(this->Program, fragment);
105         glBindProgram(this->Program);
106         // Print linking errors if any
107         glGetProgramiv(this->Program, GL_LINK_STATUS, &success);
108         if (!success)
109         {
110             glGetProgramInfoLog(this->Program, 512, NULL, infoLog);
111             std::cout << "ERROR::SHADER::PROGRAM::LINKING_FAILED" << infoLog << std::endl;
112         }
113     }
114
115     // Define several possible options for camera movement. Used as abstraction to stay away from window-system specific input methods
116     enum Camera_Movement
117     {
118         FORWARD,
119         BACKWARD,
120         LEFT,
121         RIGHT
122     };
123
124     // Default camera values
125     const GLfloat YAW = -90.0f;
126     const GLfloat PITCH = 0.0f;
127     const GLfloat SPEED = 2.5f;
128     const GLfloat SENSITIVITY = 0.25f;
129     const GLfloat ZOOM = 45.0f;
130
131     // An abstract camera class that processes input and calculates the corresponding Eular Angles, Vectors and Matrices for use in OpenGL
132     class Camera
133     {
134     public:
135         // Constructor with vectors
136         Camera(GLfloat position_x, GLfloat position_y, GLfloat position_z, GLfloat up_x, GLfloat up_y, GLfloat up_z, GLfloat yaw, GLfloat pitch) : front(glm::vec3(0.0f, 0.0f, 0.0f)), movementSpeed(SPEED), mouseSensitivity(SENSITIVITY), zoom(ZOOM)
137         {
138             this->position = position;
139             this->up = up;
140             this->front = front;
141             this->pitch = pitch;
142             this->updateCameraVectors();
143         }
144
145         // Constructor with scalar values
146         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(SENSITIVITY), zoom(ZOOM)
147         {
148     }
149
150     };
151
152     Camera camera;
153
154     // Set the camera's position
155     void setPos(GLfloat x, GLfloat y, GLfloat z)
156     {
157         position.x = x;
158         position.y = y;
159         position.z = z;
160
161         camera.setPos(x, y, z);
162     }
163
164     // Set the camera's up vector (IMPORTANT: Needs to be unit vector)
165     void setUp(GLfloat x, GLfloat y, GLfloat z)
166     {
167         up.x = x;
168         up.y = y;
169         up.z = z;
170
171         camera.setUp(x, y, z);
172     }
173
174     // Set the camera's yaw and pitch
175     void setYawPitch(GLfloat yaw, GLfloat pitch)
176     {
177         yaw = yaw;
178         pitch = pitch;
179
180         camera.setYawPitch(yaw, pitch);
181     }
182
183     // Set the camera's field of view
184     void setZoom(GLfloat zoom)
185     {
186         zoom = zoom;
187
188         camera.setZoom(zoom);
189     }
190
191     // Processes input received from any keyboard like GLFW or similar
192     void processInput()
193     {
194         if (glfwGetKey(GLFW_KEY_W) == GLFW_PRESS)
195             camera.ProcessKeyboard(Camera_Movement::FORWARD, deltaTime);
196         if (glfwGetKey(GLFW_KEY_S) == GLFW_PRESS)
197             camera.ProcessKeyboard(Camera_Movement::BACKWARD, deltaTime);
198         if (glfwGetKey(GLFW_KEY_A) == GLFW_PRESS)
199             camera.ProcessKeyboard(Camera_Movement::LEFT, deltaTime);
200         if (glfwGetKey(GLFW_KEY_D) == GLFW_PRESS)
201             camera.ProcessKeyboard(Camera_Movement::RIGHT, deltaTime);
202
203         if (glfwGetKey(GLFW_KEY_Q) == GLFW_PRESS)
204             camera.ProcessMouseMovement(-mouseSensitivity);
205         if (glfwGetKey(GLFW_KEY_E) == GLFW_PRESS)
206             camera.ProcessMouseMovement(mouseSensitivity);
207
208         if (glfwGetKey(GLFW_KEY_Z) == GLFW_PRESS)
209             camera.ProcessZoom(0.5f);
210         if (glfwGetKey(GLFW_KEY_X) == GLFW_PRESS)
211             camera.ProcessZoom(-0.5f);
212     }
213
214     // Returns the camera's View Matrix
215     glm::mat4 GetViewMatrix()
216     {
217         return camera.GetViewMatrix();
218     }
219
220     // Returns the camera's Projection Matrix
221     glm::mat4 GetProjectionMatrix()
222     {
223         return camera.GetProjectionMatrix();
224     }
225
226     // Returns the camera's Model View Projection Matrix
227     glm::mat4 GetModelViewProjectionMatrix()
228     {
229         return camera.GetModelViewProjectionMatrix();
230     }
231
232     // Returns the camera's Model View Matrix
233     glm::mat4 GetModelViewMatrix()
234     {
235         return camera.GetModelViewMatrix();
236     }
237
238     // Returns the camera's View Matrix
239     glm::mat4 GetViewMatrix()
240     {
241         return camera.GetViewMatrix();
242     }
243
244     // Returns the camera's Projection Matrix
245     glm::mat4 GetProjectionMatrix()
246     {
247         return camera.GetProjectionMatrix();
248     }
249
250     // Returns the camera's Model View Projection Matrix
251     glm::mat4 GetModelViewProjectionMatrix()
252     {
253         return camera.GetModelViewProjectionMatrix();
254     }
255
256     // Returns the camera's Model View Matrix
257     glm::mat4 GetModelViewMatrix()
258     {
259         return camera.GetModelViewMatrix();
260     }
261
262     // Returns the camera's View Matrix
263     glm::mat4 GetViewMatrix()
264     {
265         return camera.GetViewMatrix();
266     }
267
268     // Returns the camera's Projection Matrix
269     glm::mat4 GetProjectionMatrix()
270     {
271         return camera.GetProjectionMatrix();
272     }
273
274     // Returns the camera's Model View Projection Matrix
275     glm::mat4 GetModelViewProjectionMatrix()
276     {
277         return camera.GetModelViewProjectionMatrix();
278     }
279
280     // Returns the camera's Model View Matrix
281     glm::mat4 GetModelViewMatrix()
282     {
283         return camera.GetModelViewMatrix();
284     }
285
286     // Returns the camera's View Matrix
287     glm::mat4 GetViewMatrix()
288     {
289         return camera.GetViewMatrix();
290     }
291
292     // Returns the camera's Projection Matrix
293     glm::mat4 GetProjectionMatrix()
294     {
295         return camera.GetProjectionMatrix();
296     }
297
298     // Returns the camera's Model View Projection Matrix
299     glm::mat4 GetModelViewProjectionMatrix()
300     {
301         return camera.GetModelViewProjectionMatrix();
302     }
303
304     // Returns the camera's Model View Matrix
305     glm::mat4 GetModelViewMatrix()
306     {
307         return camera.GetModelViewMatrix();
308     }
309
310     // Returns the camera's View Matrix
311     glm::mat4 GetViewMatrix()
312     {
313         return camera.GetViewMatrix();
314     }
315
316     // Returns the camera's Projection Matrix
317     glm::mat4 GetProjectionMatrix()
318     {
319         return camera.GetProjectionMatrix();
320     }
321
322     // Returns the camera's Model View Projection Matrix
323     glm::mat4 GetModelViewProjectionMatrix()
324     {
325         return camera.GetModelViewProjectionMatrix();
326     }
327
328     // Returns the camera's Model View Matrix
329     glm::mat4 GetModelViewMatrix()
330     {
331         return camera.GetModelViewMatrix();
332     }
333
334     // Returns the camera's View Matrix
335     glm::mat4 GetViewMatrix()
336     {
337         return camera.GetViewMatrix();
338     }
339
340     // Returns the camera's Projection Matrix
341     glm::mat4 GetProjectionMatrix()
342     {
343         return camera.GetProjectionMatrix();
344     }
345
346     // Returns the camera's Model View Projection Matrix
347     glm::mat4 GetModelViewProjectionMatrix()
348     {
349         return camera.GetModelViewProjectionMatrix();
350     }
351
352     // Returns the camera's Model View Matrix
353     glm::mat4 GetModelViewMatrix()
354     {
355         return camera.GetModelViewMatrix();
356     }
357
358     // Returns the camera's View Matrix
359     glm::mat4 GetViewMatrix()
360     {
361         return camera.GetViewMatrix();
362     }
363
364     // Returns the camera's Projection Matrix
365     glm::mat4 GetProjectionMatrix()
366     {
367         return camera.GetProjectionMatrix();
368     }
369
370     // Returns the camera's Model View Projection Matrix
371     glm::mat4 GetModelViewProjectionMatrix()
372     {
373         return camera.GetModelViewProjectionMatrix();
374     }
375
376     // Returns the camera's Model View Matrix
377     glm::mat4 GetModelViewMatrix()
378     {
379         return camera.GetModelViewMatrix();
380     }
381
382     // Returns the camera's View Matrix
383     glm::mat4 GetViewMatrix()
384     {
385         return camera.GetViewMatrix();
386     }
387
388     // Returns the camera's Projection Matrix
389     glm::mat4 GetProjectionMatrix()
390     {
391         return camera.GetProjectionMatrix();
392     }
393
394     // Returns the camera's Model View Projection Matrix
395     glm::mat4 GetModelViewProjectionMatrix()
396     {
397         return camera.GetModelViewProjectionMatrix();
398     }
399
400     // Returns the camera's Model View Matrix
401     glm::mat4 GetModelViewMatrix()
402     {
403         return camera.GetModelViewMatrix();
404     }
405
406     // Returns the camera's View Matrix
407     glm::mat4 GetViewMatrix()
408     {
409         return camera.GetViewMatrix();
410     }
411
412     // Returns the camera's Projection Matrix
413     glm::mat4 GetProjectionMatrix()
414     {
415         return camera.GetProjectionMatrix();
416     }
417
418     // Returns the camera's Model View Projection Matrix
419     glm::mat4 GetModelViewProjectionMatrix()
420     {
421         return camera.GetModelViewProjectionMatrix();
422     }
423
424     // Returns the camera's Model View Matrix
425     glm::mat4 GetModelViewMatrix()
426     {
427         return camera.GetModelViewMatrix();
428     }
429
430     // Returns the camera's View Matrix
431     glm::mat4 GetViewMatrix()
432     {
433         return camera.GetViewMatrix();
434     }
435
436     // Returns the camera's Projection Matrix
437     glm::mat4 GetProjectionMatrix()
438     {
439         return camera.GetProjectionMatrix();
440     }
441
442     // Returns the camera's Model View Projection Matrix
443     glm::mat4 GetModelViewProjectionMatrix()
444     {
445         return camera.GetModelViewProjectionMatrix();
446     }
447
448     // Returns the camera's Model View Matrix
449     glm::mat4 GetModelViewMatrix()
450     {
451         return camera.GetModelViewMatrix();
452     }
453
454     // Returns the camera's View Matrix
455     glm::mat4 GetViewMatrix()
456     {
457         return camera.GetViewMatrix();
458     }
459
460     // Returns the camera's Projection Matrix
461     glm::mat4 GetProjectionMatrix()
462     {
463         return camera.GetProjectionMatrix();
464     }
465
466     // Returns the camera's Model View Projection Matrix
467     glm::mat4 GetModelViewProjectionMatrix()
468     {
469         return camera.GetModelViewProjectionMatrix();
470     }
471
472     // Returns the camera's Model View Matrix
473     glm::mat4 GetModelViewMatrix()
474     {
475         return camera.GetModelViewMatrix();
476     }
477
478     // Returns the camera's View Matrix
479     glm::mat4 GetViewMatrix()
480     {
481         return camera.GetViewMatrix();
482     }
483
484     // Returns the camera's Projection Matrix
485     glm::mat4 GetProjectionMatrix()
486     {
487         return camera.GetProjectionMatrix();
488     }
489
490     // Returns the camera's Model View Projection Matrix
491     glm::mat4 GetModelViewProjectionMatrix()
492     {
493         return camera.GetModelViewProjectionMatrix();
494     }
495
496     // Returns the camera's Model View Matrix
497     glm::mat4 GetModelViewMatrix()
498     {
499         return camera.GetModelViewMatrix();
500     }
501
502     // Returns the camera's View Matrix
503     glm::mat4 GetViewMatrix()
504     {
505         return camera.GetViewMatrix();
506     }
507
508     // Returns the camera's Projection Matrix
509     glm::mat4 GetProjectionMatrix()
510     {
511         return camera.GetProjectionMatrix();
512     }
513
514     // Returns the camera's Model View Projection Matrix
515     glm::mat4 GetModelViewProjectionMatrix()
516     {
517         return camera.GetModelViewProjectionMatrix();
518     }
519
520     // Returns the camera's Model View Matrix
521     glm::mat4 GetModelViewMatrix()
522     {
523         return camera.GetModelViewMatrix();
524     }
525
526     // Returns the camera's View Matrix
527     glm::mat4 GetViewMatrix()
528     {
529         return camera.GetViewMatrix();
530     }
531
532     // Returns the camera's Projection Matrix
533     glm::mat4 GetProjectionMatrix()
534     {
535         return camera.GetProjectionMatrix();
536     }
537
538     // Returns the camera's Model View Projection Matrix
539     glm::mat4 GetModelViewProjectionMatrix()
540     {
541         return camera.GetModelViewProjectionMatrix();
542     }
543
544     // Returns the camera's Model View Matrix
545     glm::mat4 GetModelViewMatrix()
546     {
547         return camera.GetModelViewMatrix();
548     }
549
550     // Returns the camera's View Matrix
551     glm::mat4 GetViewMatrix()
552     {
553         return camera.GetViewMatrix();
554     }
555
556     // Returns the camera's Projection Matrix
557     glm::mat4 GetProjectionMatrix()
558     {
559         return camera.GetProjectionMatrix();
560     }
561
562     // Returns the camera's Model View Projection Matrix
563     glm::mat4 GetModelViewProjectionMatrix()
564     {
565         return camera.GetModelViewProjectionMatrix();
566     }
567
568     // Returns the camera's Model View Matrix
569     glm::mat4 GetModelViewMatrix()
570     {
571         return camera.GetModelViewMatrix();
572     }
573
574     // Returns the camera's View Matrix
575     glm::mat4 GetViewMatrix()
576     {
577         return camera.GetViewMatrix();
578     }
579
580     // Returns the camera's Projection Matrix
581     glm::mat4 GetProjectionMatrix()
582     {
583         return camera.GetProjectionMatrix();
584     }
585
586     // Returns the camera's Model View Projection Matrix
587     glm::mat4 GetModelViewProjectionMatrix()
588     {
589         return camera.GetModelViewProjectionMatrix();
590     }
591
592     // Returns the camera's Model View Matrix
593     glm::mat4 GetModelViewMatrix()
594     {
595         return camera.GetModelViewMatrix();
596     }
597
598     // Returns the camera's View Matrix
599     glm::mat4 GetViewMatrix()
600     {
601         return camera.GetViewMatrix();
602     }
603
604     // Returns the camera's Projection Matrix
605     glm::mat4 GetProjectionMatrix()
606     {
607         return camera.GetProjectionMatrix();
608     }
609
610     // Returns the camera's Model View Projection Matrix
611     glm::mat4 GetModelViewProjectionMatrix()
612     {
613         return camera.GetModelViewProjectionMatrix();
614     }
615
616     // Returns the camera's Model View Matrix
617     glm::mat4 GetModelViewMatrix()
618     {
619         return camera.GetModelViewMatrix();
620     }
621
622     // Returns the camera's View Matrix
623     glm::mat4 GetViewMatrix()
624     {
625         return camera.GetViewMatrix();
626     }
627
628     // Returns the camera's Projection Matrix
629     glm::mat4 GetProjectionMatrix()
630     {
631         return camera.GetProjectionMatrix();
632     }
633
634     // Returns the camera's Model View Projection Matrix
635     glm::mat4 GetModelViewProjectionMatrix()
636     {
637         return camera.GetModelViewProjectionMatrix();
638     }
639
640     // Returns the camera's Model View Matrix
641     glm::mat4 GetModelViewMatrix()
642     {
643         return camera.GetModelViewMatrix();
644     }
645
646     // Returns the camera's View Matrix
647     glm::mat4 GetViewMatrix()
648     {
649         return camera.GetViewMatrix();
650     }
651
652     // Returns the camera's Projection Matrix
653     glm::mat4 GetProjectionMatrix()
654     {
655         return camera.GetProjectionMatrix();
656     }
657
658     // Returns the camera's Model View Projection Matrix
659     glm::mat4 GetModelViewProjectionMatrix()
660     {
661         return camera.GetModelViewProjectionMatrix();
662     }
663
664     // Returns the camera's Model View Matrix
665     glm::mat4 GetModelViewMatrix()
666     {
667         return camera.GetModelViewMatrix();
668     }
669
670     // Returns the camera's View Matrix
671     glm::mat4 GetViewMatrix()
672     {
673         return camera.GetViewMatrix();
674     }
675
676     // Returns the camera's Projection Matrix
677     glm::mat4 GetProjectionMatrix()
678     {
679         return camera.GetProjectionMatrix();
680     }
681
682     // Returns the camera's Model View Projection Matrix
683     glm::mat4 GetModelViewProjectionMatrix()
684     {
685         return camera.GetModelViewProjectionMatrix();
686     }
687
688     // Returns the camera's Model View Matrix
689     glm::mat4 GetModelViewMatrix()
690     {
691         return camera.GetModelViewMatrix();
692     }
693
694     // Returns the camera's View Matrix
695     glm::mat4 GetViewMatrix()
696     {
697         return camera.GetViewMatrix();
698     }
699
700     // Returns the camera's Projection Matrix
701     glm::mat4 GetProjectionMatrix()
702     {
703         return camera.GetProjectionMatrix();
704     }
705
706     // Returns the camera's Model View Projection Matrix
707     glm::mat4 GetModelViewProjectionMatrix()
708     {
709         return camera.GetModelViewProjectionMatrix();
710     }
711
712     // Returns the camera's Model View Matrix
713     glm::mat4 GetModelViewMatrix()
714     {
715         return camera.GetModelViewMatrix();
716     }
717
718     // Returns the camera's View Matrix
719     glm::mat4 GetViewMatrix()
720     {
721         return camera.GetViewMatrix();
722     }
723
724     // Returns the camera's Projection Matrix
725     glm::mat4 GetProjectionMatrix()
726     {
727         return camera.GetProjectionMatrix();
728     }
729
730     // Returns the camera's Model View Projection Matrix
731     glm::mat4 GetModelViewProjectionMatrix()
732     {
733         return camera.GetModelViewProjectionMatrix();
734     }
735
736     // Returns the camera's Model View Matrix
737     glm::mat4 GetModelViewMatrix()
738     {
739         return camera.GetModelViewMatrix();
740     }
741
742     // Returns the camera's View Matrix
743     glm::mat4 GetViewMatrix()
744     {
745         return camera.GetViewMatrix();
746     }
747
748     // Returns the camera's Projection Matrix
749     glm::mat4 GetProjectionMatrix()
750     {
751         return camera.GetProjectionMatrix();
752     }
753
754     // Returns the camera's Model View Projection Matrix
755     glm::mat4 GetModelViewProjectionMatrix()
756     {
757         return camera.GetModelViewProjectionMatrix();
758     }
759
760     // Returns the camera's Model View Matrix
761     glm::mat4 GetModelViewMatrix()
762     {
763         return camera.GetModelViewMatrix();
764     }
765
766     // Returns the camera's View Matrix
767     glm::mat4 GetViewMatrix()
768     {
769         return camera.GetViewMatrix();
770     }
771
772     // Returns the camera's Projection Matrix
773     glm::mat4 GetProjectionMatrix()
774     {
775         return camera.GetProjectionMatrix();
776     }
777
778     // Returns the camera's Model View Projection Matrix
779     glm::mat4 GetModelViewProjectionMatrix()
780     {
781         return camera.GetModelViewProjectionMatrix();
782     }
783
784     // Returns the camera's Model View Matrix
785     glm::mat4 GetModelViewMatrix()
786     {
787         return camera.GetModelViewMatrix();
788     }
789
790     // Returns the camera's View Matrix
791     glm::mat4 GetViewMatrix()
792     {
793         return camera.GetViewMatrix();
794     }
795
796     // Returns the camera's Projection Matrix
797     glm::mat4 GetProjectionMatrix()
798     {
799         return camera.GetProjectionMatrix();
800     }
801
802     // Returns the camera's Model View Projection Matrix
803     glm::mat4 GetModelViewProjectionMatrix()
804     {
805         return camera.GetModelViewProjectionMatrix();
806     }
807
808     // Returns the camera's Model View Matrix
809     glm::mat4 GetModelViewMatrix()
810     {
811         return camera.GetModelViewMatrix();
812     }
813
814     // Returns the camera's View Matrix
815     glm::mat4 GetViewMatrix()
816     {
817         return camera.GetViewMatrix();
818     }
819
820     // Returns the camera's Projection Matrix
821     glm::mat4 GetProjectionMatrix()
822     {
823         return camera.GetProjectionMatrix();
824     }
825
826     // Returns the camera's Model View Projection Matrix
827     glm::mat4 GetModelViewProjectionMatrix()
828     {
829         return camera.GetModelViewProjectionMatrix();
830     }
831
832     // Returns the camera's Model View Matrix
833     glm::mat4 GetModelViewMatrix()
834     {
835         return camera.GetModelViewMatrix();
836     }
837
838     // Returns the camera's View Matrix
839     glm::mat4 GetViewMatrix()
840     {
841         return camera.GetViewMatrix();
842     }
843
844     // Returns the camera's Projection Matrix
845     glm::mat4 GetProjectionMatrix()
846     {
847         return camera.GetProjectionMatrix();
848     }
849
850     // Returns the camera's Model View Projection Matrix
851     glm::mat4 GetModelViewProjectionMatrix()
852     {
853         return camera.GetModelViewProjectionMatrix();
854     }
855
856     // Returns the camera's Model View Matrix
857     glm::mat4 GetModelViewMatrix()
858     {
859         return camera.GetModelViewMatrix();
860     }
861
862     // Returns the camera's View Matrix
863     glm::mat4 GetViewMatrix()
864     {
865         return camera.GetViewMatrix();
866     }
867
868     // Returns the camera's Projection Matrix
869     glm::mat4 GetProjectionMatrix()
870     {
871         return camera.GetProjectionMatrix();
872     }
873
874     // Returns the camera's Model View Projection Matrix
875     glm::mat4 GetModelViewProjectionMatrix()
876     {
877         return camera.GetModelViewProjectionMatrix();
878     }
879
880     // Returns the camera's Model View Matrix
881     glm::mat4 GetModelViewMatrix()
882     {
883         return camera.GetModelViewMatrix();
884     }
885
886     // Returns the camera's View Matrix
887     glm::mat4 GetViewMatrix()
888     {
889         return camera.GetViewMatrix();
890     }
891
892     // Returns the camera's Projection Matrix
893     glm::mat4 GetProjectionMatrix()
894     {
895         return camera.GetProjectionMatrix();
896     }
897
898     // Returns the camera's Model View Projection Matrix
899     glm::mat4 GetModelViewProjectionMatrix()
900     {
901         return camera.GetModelViewProjectionMatrix();
902     }
903
904     // Returns the camera's Model View Matrix
905     glm::mat4 GetModelViewMatrix()
906     {
907         return camera.GetModelViewMatrix();
908     }
909
910     // Returns the camera's View Matrix
911     glm::mat4 GetViewMatrix()
912     {
913         return camera.GetViewMatrix();
914     }
915
916     // Returns the camera's Projection Matrix
917     glm::mat4 GetProjectionMatrix()
918     {
919         return camera.GetProjectionMatrix();
920     }
921
922     // Returns the camera's Model View Projection Matrix
923     glm::mat4 GetModelViewProjectionMatrix()
924     {
925         return camera.GetModelViewProjectionMatrix();
926     }
927
928     // Returns the camera's Model View Matrix
929     glm::mat4 GetModelViewMatrix()
930     {
931         return camera.GetModelViewMatrix();
932     }
933
934     // Returns the camera's View Matrix
935     glm::mat4 GetViewMatrix()
936     {
937         return camera.GetViewMatrix();
938     }
939
940     // Returns the camera's Projection Matrix
941     glm::mat4 GetProjectionMatrix()
942     {
943         return camera.GetProjectionMatrix();
944     }
945
946     // Returns the camera's Model View Projection Matrix
947     glm::mat4 GetModelViewProjectionMatrix()
948     {
949         return camera.GetModelViewProjectionMatrix();
950     }
951
952     // Returns the camera's Model View Matrix
953     glm::mat4 GetModelViewMatrix()
954     {
955         return camera.GetModelViewMatrix();
956     }
957
958     // Returns the camera's View Matrix
959     glm::mat4 GetViewMatrix()
960     {
961         return camera.GetViewMatrix();
962     }
963
964     // Returns the camera's Projection Matrix
965     glm::mat4 GetProjectionMatrix()
966     {
967         return camera.GetProjectionMatrix();
968     }
969
970     // Returns the camera's Model View Projection Matrix
971     glm::mat4 GetModelViewProjectionMatrix()
972     {
973         return camera.GetModelViewProjectionMatrix();
974     }
975
976     // Returns the camera's Model View Matrix
977     glm::mat4 GetModelViewMatrix()
978     {
979         return camera.GetModelViewMatrix();
980     }
981
982     // Returns the camera's View Matrix
983     glm::mat4 GetViewMatrix()
984     {
985         return camera.GetViewMatrix();
986     }
987
988     // Returns the camera's Projection Matrix
989     glm::mat4 GetProjectionMatrix()
990     {
991         return camera.GetProjectionMatrix();
992     }
993
994     // Returns the camera's Model View Projection Matrix
995     glm::mat4 GetModelViewProjectionMatrix()
996     {
997         return camera.GetModelViewProjectionMatrix();
998     }
999
1000    // Returns the camera's Model View Matrix
1001    glm::mat4 GetModelViewMatrix()
1002    {
1003        return camera.GetModelViewMatrix();
1004    }
1005
1006    // Returns the camera's View Matrix
1007    glm::mat4 GetViewMatrix()
1008    {
1009        return camera.GetViewMatrix();
1010    }
1011
1012    // Returns the camera's Projection Matrix
1013    glm::mat4 GetProjectionMatrix()
1014    {
1015        return camera.GetProjectionMatrix();
1016    }
1017
1018    // Returns the camera's Model View Projection Matrix
1019    glm::mat4 GetModelViewProjectionMatrix()
1020    {
1021        return camera.GetModelViewProjectionMatrix();
1022    }
1023
1024    // Returns the camera's Model View Matrix
1025    glm::mat4 GetModelViewMatrix()
1026    {
1027        return camera.GetModelViewMatrix();
1028    }
1029
1030    // Returns the camera's View Matrix
1031    glm::mat4 GetViewMatrix()
1032    {
1033        return camera.GetViewMatrix();
1034    }
1035
1036    // Returns the camera's Projection Matrix
1037    glm::mat4 GetProjectionMatrix()
1038    {
1039        return camera.GetProjectionMatrix();
1040    }
1041
1042    // Returns the camera's Model View Projection Matrix
1043    glm::mat4 GetModelViewProjectionMatrix()
1044    {
1045        return camera.GetModelViewProjectionMatrix();
1046    }
1047
1048    // Returns the camera's Model View Matrix
1049    glm::mat4 GetModelViewMatrix()
1050    {
1051        return camera.GetModelViewMatrix();
1052    }
1053
1054    // Returns the camera's View Matrix
1055    glm::mat4 GetViewMatrix()
1056    {
1057        return camera.GetViewMatrix();
1058    }
1059
1060    // Returns the camera's Projection Matrix
1061    glm::mat4 GetProjectionMatrix()
1062    {
1063        return camera.GetProjectionMatrix();
1064    }
1065
1066    // Returns the camera's Model View Projection Matrix
1067    glm::mat4 GetModelViewProjectionMatrix()
1068    {
1069        return camera.GetModelViewProjectionMatrix();
1070    }
1071
1072    // Returns the camera's Model View Matrix
1073    glm::mat4 GetModelViewMatrix()
1074    {
1075        return camera.GetModelViewMatrix();
1076    }
1077
1078    // Returns the camera's View Matrix
1079    glm::mat4 GetViewMatrix()
1080    {
1081        return camera.GetViewMatrix();
1082    }
1083
1084    // Returns the camera's Projection Matrix
1085    glm::mat4 GetProjectionMatrix()
1086    {
1087        return camera.GetProjectionMatrix();
1088    }
1089
1090    // Returns the camera's Model View Projection Matrix
1091    glm::mat4 GetModelViewProjectionMatrix()
1092    {
1093        return camera.GetModelViewProjectionMatrix();
1094    }
1095
1096    // Returns the camera's Model View Matrix
1097    glm::mat4 GetModelViewMatrix()
1098    {
1099        return camera.GetModelViewMatrix();
1100    }
1101
1102    // Returns the camera's View Matrix
1103    glm::mat4 GetViewMatrix()
1104    {
1105        return camera.GetViewMatrix();
1106    }
1107
1108    // Returns the camera's Projection Matrix
1109    glm::mat4 GetProjectionMatrix()
1110    {
1111        return camera.GetProjectionMatrix();
1112    }
1113
1114    // Returns the camera's Model View Projection Matrix
1115    glm::mat4 GetModelViewProjectionMatrix()
1116    {
1117        return camera.GetModelViewProjectionMatrix();
1118    }
1119
1120    // Returns the camera's Model View Matrix
1121    glm::mat4 GetModelViewMatrix()
1122    {
1123        return camera.GetModelViewMatrix();
1124    }
1125
1126    // Returns the camera's View Matrix
1127    glm::mat4 GetViewMatrix()
1128    {
1129        return camera.GetViewMatrix();
1130    }
1131
1132    // Returns the camera's Projection Matrix
1133    glm::mat4 GetProjectionMatrix()
1134    {
1135        return camera.GetProjectionMatrix();
1136    }
1137
1138    // Returns the camera's Model View Projection Matrix
1139    glm::mat4 GetModelViewProjectionMatrix()
1140    {
1141        return camera.GetModelViewProjectionMatrix();
1142    }
1143
1144    // Returns the camera's Model View Matrix
1145    glm::mat4 GetModelViewMatrix()
1146    {
1147        return camera.GetModelViewMatrix();
1148    }
1149
1150    // Returns the camera's View Matrix
1151    glm::mat4 GetViewMatrix()
1152    {
1153        return camera.GetViewMatrix();
1154    }
1155
1156    // Returns the camera's Projection Matrix
1157    glm::mat4 GetProjectionMatrix()
1158    {
1159        return camera.GetProjectionMatrix();
1160    }
1161
1162    // Returns the camera's Model View Projection Matrix
1163    glm::mat4 GetModelViewProjectionMatrix()
1164    {
1165        return camera.GetModelViewProjectionMatrix();
1166    }
1167
1168    // Returns the camera's Model View Matrix
1169    glm::mat4 GetModelViewMatrix()
1170    {
1171        return camera.GetModelViewMatrix();
1172    }
1173
1174    // Returns the camera's View Matrix
1175    glm::mat4 GetViewMatrix()
1176    {
1177        return camera.GetViewMatrix();
1178    }
1179
1180    // Returns the camera's Projection Matrix
1181    glm::mat4 GetProjectionMatrix()
1182    {
1183        return camera.GetProjectionMatrix();
1184    }
1185
1186    // Returns the camera's Model View Projection Matrix
1187    glm::mat4 GetModelViewProjectionMatrix()
1188    {
1189        return camera.GetModelViewProjectionMatrix();
1190    }
1191
1192    // Returns the camera's Model View Matrix
1193    glm::mat4 GetModelViewMatrix()
1194    {
1195        return camera.GetModelViewMatrix();
1196    }
1197
1198    // Returns the camera's View Matrix
1199    glm::mat4 GetViewMatrix()
1200    {
1201        return camera.GetViewMatrix();
1202    }
1203
1204    // Returns the camera's Projection Matrix
1205    glm::mat4 GetProjectionMatrix()
1206    {
1207        return camera.GetProjectionMatrix();
1208    }
1209
1210    // Returns the camera's Model View Projection Matrix
1211    glm::mat4 GetModelViewProjectionMatrix()
1212    {
1213        return camera.GetModelViewProjectionMatrix();
1214    }
1215
1216    // Returns the camera's Model View Matrix
1217    glm::mat4 GetModelViewMatrix()
1218    {
1219        return camera.GetModelViewMatrix();
1220    }
1221
1222    // Returns the camera's View Matrix
1223    glm::mat4 GetViewMatrix()
1224    {
1225        return camera.GetViewMatrix();
1226    }
1227
1228    // Returns the camera's Projection Matrix
1229    glm::mat4 GetProjectionMatrix()
1230    {
1231        return camera.GetProjectionMatrix();
1232    }
1233
1234    // Returns the camera's Model View Projection Matrix
1235    glm::mat4 GetModelViewProjectionMatrix()
1236    {
1237        return camera.GetModelViewProjectionMatrix();
1238    }
12
```

```

46     this->position = glm::vec3(posX, posY, posZ);
47     this->worldUp = glm::vec3(upX, upY, upZ);
48     this->yaw = yaw;
49     this->pitch = pitch;
50     this->updateCameraVectors();
51 }
52
53 // Returns the view matrix calculated using Eular Angles and the LookAt Matrix
54 glm::mat4 GetViewMatrix()
55 {
56     return glm::lookAt(this->position, this->position + this->front, this->up);
57 }
58
59 // 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)
60 void ProcessKeyboard(Camera_Movement direction, GLfloat deltaTime)
61 {
62     GLfloat velocity = this->movementSpeed * deltaTime;
63
64     if (direction == FORWARD)
65     {
66         this->position += this->front * velocity;
67     }
68
69     if (direction == BACKWARD)
70     {
71         this->position -= this->front * velocity;
72     }
73
74     if (direction == LEFT)
75     {
76         this->position -= this->right * velocity;
77     }
78
79     if (direction == RIGHT)
80     {
81         this->position += this->right * velocity;
82     }
83
84
85 // Processes input received from a mouse input system. Expects the offset value in both the x and y direction.
86 void ProcessMouseMovement(GLfloat xOffset, GLfloat yOffset, GLboolean constrainPitch = true)
87 {
88     xOffset *= this->mouseSensitivity;
89     yOffset *= this->mouseSensitivity;
90
91     this->yaw += xOffset;
92     this->pitch += yOffset;
93
94     // Make sure that when pitch is out of bounds, screen doesn't get flipped
95     if (constrainPitch)
96     {
97         if (this->pitch > 89.0f)
98         {
99             this->pitch = 89.0f;
100         }
101
102         if (this->pitch < -89.0f)
103         {
104             this->pitch = -89.0f;
105         }
106     }
107
108     // Update Front, Right and Up Vectors using the updated Eular angles
109     this->updateCameraVectors();
110 }
111
112 // Processes input received from a mouse scroll-wheel event. Only requires input on the vertical wheel-axis
113 void ProcessMouseScroll(GLfloat yOffset)
114 {
115 }
116
117
118 GLfloat GetZoom()
119 {
120     return this->zoom;
121 }
122
123 glm::vec3 GetPosition()
124 {
125     return this->position;
126 }
127
128 glm::vec3 GetFront()
129 {
130     return this->front;
131 }
132
133

```

## Billboard.frag

Este fragment shader está diseñado para renderizar texturas con transparencia, muy útil para efectos como gotas de agua, vapor o cualquier billboard. Primero recibe las coordenadas de textura (TexCoords) generadas en el vertex shader y utiliza el sampler texture1 para obtener el color del píxel desde la imagen. Luego aplica una condición clave: si la componente alfa del píxel (texColor.a) es menor a 0.1, el shader descarta ese fragmento usando discard, evitando dibujar partes completamente transparentes y logrando bordes suaves sin cuadros indeseados. Finalmente, si el píxel es válido, lo coloca en pantalla asignándolo a FragColor. En resumen, este shader permite dibujar partículas y objetos semitransparentes de forma limpia, respetando correctamente la transparencia de su textura.

```

1      #version 330 core
2      out vec4 FragColor;
3
4      in vec2 TexCoords;
5
6      uniform sampler2D texture1;
7
8      void main()
9      {
10         vec4 texColor = texture(texture1, TexCoords);
11         if(texColor.a < 0.1)
12             discard;
13         FragColor = texColor;
14     }
15

```

## Billboard.vs

Este vertex shader se encarga de preparar cada vértice de un billboard o partícula para ser dibujado en la escena 3D. Recibe como entrada la posición del vértice (aPos) y sus coordenadas de textura (aTexCoords) y envía

estas últimas al fragment shader mediante la variable TexCoords. Luego toma la posición del vértice y la transforma mediante las matrices model, view y projection, que ubican, orientan y proyectan el objeto correctamente en el mundo 3D y en la cámara. El resultado final se asigna a gl\_Position, lo que determina el lugar exacto donde aparecerá la partícula. En resumen, este shader posiciona la geometría del billboard en el espacio 3D y pasa las coordenadas de textura necesarias para el renderizado final.

```

1  #version 330 core
2  out vec4 FragColor;
3
4  in vec2 TexCoords;
5
6  uniform sampler2D texture1;
7
8  void main()
9  {
10     vec4 texColor = texture(texture1, TexCoords);
11     if(texColor.a < 0.1)
12         discard;
13     FragColor = texColor;
14 }
```

### Lamp.vs:

Este vertex shader se encarga de transformar los cubos o esferas que representan las fuentes de luz en el mundo 3D. Aunque no afectan la iluminación directamente, su correcta visualización facilita la orientación espacial, ajuste de posicionamiento de luces puntuales y la depuración visual del sistema de iluminación implementado.

```

1  #version 330 core
2  layout (location = 0) in vec3 position;
3
4
5
6  uniform mat4 model;
7  uniform mat4 view;
8  uniform mat4 projection;
9
10 void main()
11 {
12     gl_Position = projection * view * model * vec4(position, 1.0f)
13 }
14 }
```

### lighting.frag

Este código corresponde a un fragment shader en GLSL encargado de calcular el color final de cada fragmento (píxel) de los modelos renderizados en la escena mediante el modelo de iluminación Phong. Su objetivo principal es simular un comportamiento de luz realista combinando tres tipos de fuentes de iluminación: una luz direccional, varias luces puntuales y un spotlight. Además, incorpora soporte para texturas difusas, texturas especulares y transparencia, permitiendo efectos visuales dinámicos dentro del entorno 3D.

El shader inicia declarando varias **estructuras (struct)** que agrupan los datos necesarios para representar materiales y fuentes de luz.

La estructura Material contiene:

- una textura difusa (diffuse) que define el color base del objeto,
- una textura especular (specular) que controla en qué zonas el objeto refleja más luz,
- y un parámetro shininess que determina qué tan concentrado o disperso es el brillo especular.

Después se definen tres tipos de luces:

1. **DirLight** (luz **direccional**):  
Representa una luz global similar al sol. No tiene posición, solo una **dirección fija**, y aporta

iluminación en toda la escena. Sus componentes ambiental, difusa y especular se aplican de manera uniforme a los fragmentos.

2. **PointLight** **(luz puntual):** Simula focos o lámparas colocadas dentro del escenario. Cada luz puntual tiene una **posición en el espacio 3D**, además de parámetros de **atenuación** (constant, linear y quadratic) que permiten que la intensidad disminuya conforme aumenta la distancia entre la luz y el fragmento. Esto evita que los objetos lejanos se iluminen con la misma fuerza que los cercanos, aportando realismo.
3. **SpotLight** **(luz tipo linterna):** Es similar a una luz puntual pero con dirección y forma cónica. Tiene un ángulo interno (cutOff) donde la luz es fuerte, y un ángulo externo (outerCutOff) donde la luz se desvanece gradualmente. Esto permite simular una linterna o foco dirigido que ilumina solo un área específica.

El shader recibe desde el vertex shader tres valores interpolados:

- FragPos: posición del fragmento en el mundo,
- Normal: normal del fragmento (para saber hacia dónde mira la superficie),
- y TexCoords: coordenadas UV para leer las texturas.

En el main(), el shader normaliza la normal y calcula la dirección de vista (viewDir) desde el fragmento hacia la cámara. Con estos datos empieza el cálculo de luz:

- Primero aplica la **luz direccional** usando CalcDirLight, obteniendo el primer aporte de iluminación.
- Luego recorre un arreglo de **luces puntuales** (pointLights[]) y suma el resultado de cada una usando CalcPointLight.
- Finalmente agrega el aporte del **spotlight** mediante CalcSpotLight.

Cada función de luz aplica el modelo Phong clásico:

1. **Componente ambiental (ambient):** iluminación base constante que evita zonas completamente negras.
2. **Componente difusa (diffuse):** depende del ángulo entre la normal del objeto y la dirección de luz; simula cómo la luz “rebota” suavemente en superficies mates.
3. **Componente especular (specular):** simula reflejos brillantes; depende del ángulo entre el reflejo de luz y la dirección hacia la cámara, y se controla con shininess y la textura especular.

En los cálculos de luces puntuales y spotlight se incluye la **atenuación por distancia**, lo cual hace que la luz pierda fuerza gradualmente según el fragmento esté más lejos del foco. En el spotlight, además de eso, se calcula una **intensidad por cono**, que determina si el fragmento está dentro del área iluminada y qué tan fuerte debe verse la luz en ese punto.

Al final, el resultado total (result) representa la suma de todas las aportaciones de luz. Ese color se envía como salida del shader en color.

Además, el shader incluye un control de transparencia: si el alfa del fragmento es bajo y la variable transparency está activada, el fragmento se descarta (discard). Esto se usa para objetos con textura transparente como efectos de vapor, partículas o elementos con canal alfa.

```

1. #version 330 core
2. #define NUMBER_OF_POINT_LIGHTS 4
3.
4. struct Material
5. {
6.     sampler2D diffuse;
7.     sampler2D specular;
8.     float shininess;
9. };
10.
11. struct DirLight
12. {
13.     vec3 direction;
14.
15.     vec3 ambient;
16.     vec3 diffuse;
17.     vec3 specular;
18.     float linear;
19.     float quadratic;
20.
21.     vec3 ambient;
22.     vec3 position;
23.     vec3 viewDir;
24.     float constant;
25.     float linear;
26.     float quadratic;
27.
28.     vec3 ambient;
29.     vec3 diffuse;
30.     vec3 specular;
31.     float outerCutoff;
32. };
33.
34. struct SpotLight
35. {
36.     vec3 position;
37.     vec3 direction;
38.     float cutoff;
39.     float outerCutoff;
40.     float linear;
41.     float quadratic;
42. };
43.
44. vec3 FragPos;
45. in vec3 Normal;
46. in vec3 TexCoords;
47.
48. struct Material
49. {
50.     vec3 Ambient;
51.     vec3 Diffuse;
52.     vec3 Specular;
53.     float Shininess;
54.     float Reflective;
55.     float Transparency;
56.     vec3 LightDir;
57.     vec3 LightColor;
58.     uniform PointLight pointLights[NUMBER_OF_POINT_LIGHTS];
59.     uniform SpotLight spotLight;
60.     uniform int Transparency;
61.     uniform int Specular;
62. };
63.
64. // Function prototypes
65. vec3 CalcDirLight(DirLight light, vec3 normal, vec3 viewDir);
66. vec3 CalcPointLight(PointLight light, vec3 normal, vec3 fragPos, vec3 viewDir);
67. vec3 CalcSpotLight(SpotLight light, vec3 normal, vec3 fragPos, vec3 viewDir);
68.
69. void main()
70. {
71.     vec3 norm = normalize(Normal);
72.     vec3 viewDir = normalize(viewPos - FragPos);
73.
74.     // Directional Lighting
75.     vec3 result = CalcDirLight(direction, norm, viewDir);
76.
77.     // Point Lights
78.     for (int i = 0; i < NUMBER_OF_POINT_LIGHTS; i++)
79.     {
80.         result += CalcPointLight(pointLights[i], norm, FragPos, viewDir);
81.     }
82.
83.     // Spot Light
84.     result += CalcSpotLight(spotLight, norm, FragPos, viewDir);
85.
86.     color = result;
87.     if (color.a < 0.1 || transparency == 1)
88.         discard;
89.
90. }
91.
92. // Calculates the color when using a directional light.
93. vec3 CalcDirLight(DirLight light, vec3 normal, vec3 viewDir)
94. {
95.     vec3 lightDir = normalize(-light.direction);
96.
97.     // Diffuse shading
98.     float diff = max(dot(normal, lightDir), 0.0);
99.
100.    // Specular shading
101.    vec3 reflectDir = reflect(-lightDir, normal);
102.    float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
103.
104.    // Combine results
105.    vec3 ambient = light.ambient * vec3(texture(material.diffuse, TexCoords));
106.    vec3 diffuse = light.diffuse * diff * vec3(texture(material.diffuse, TexCoords));
107.    vec3 specular = light.specular * spec * vec3(texture(material.specular, TexCoords));
108.
109.    return (ambient + diffuse + specular);
110.
111. }
112.
113. // Calculates the color when using a point light.
114. vec3 CalcPointLight(PointLight light, vec3 normal, vec3 fragPos, vec3 viewDir)
115. {
116.     vec3 lightDir = normalize(light.position - fragPos);
117.
118.     // Diffuse shading
119.     float diff = max(dot(normal, lightDir), 0.0);
120.
121.     // Specular shading
122.     vec3 reflectDir = reflect(-lightDir, normal);
123.     float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
124.
125.     // Attenuation
126.     float distance = length(light.position - fragPos);
127.     float attenuation = 1.0f / (light.constant + light.linear * distance + light.quadratic * (distance * distance));
128.
129.     // Combine results
130.     vec3 ambient = light.ambient * vec3(texture(material.diffuse, TexCoords));
131.     vec3 diffuse = light.diffuse * diff * vec3(texture(material.diffuse, TexCoords));
132.     vec3 specular = light.specular * spec * vec3(texture(material.specular, TexCoords));
133.
134.     ambient *= attenuation;
135.     diffuse *= attenuation;
136.     specular *= attenuation;
137.
138.     return (ambient + diffuse + specular);
139.
140. }
141.
142. // Calculates the color when using a spot light.
143. vec3 CalcSpotLight(SpotLight light, vec3 normal, vec3 fragPos, vec3 viewDir)
144. {
145.     vec3 lightDir = normalize(light.position - fragPos);
146.
147.     // Diffuse shading
148.     float diff = max(dot(normal, lightDir), 0.0);
149.
150.     // Specular shading
151.     vec3 reflectDir = reflect(-lightDir, normal);
152.     float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
153.
154.     // Attenuation
155.     float distance = length(light.position - fragPos);
156.     float attenuation = 1.0f / (light.constant + light.linear * distance + light.quadratic * (distance * distance));
157.
158.     // Spotlight intensity
159.     float theta = dot(lightDir, normalize(-light.direction));
160.     float epsilon = light.cutOff - light.outerCutOff;
161.     float intensity = clamp((theta - light.outerCutOff) / epsilon, 0.0, 1.0);
162.
163.     // Combine results
164.     vec3 ambient = light.ambient * vec3(texture(material.diffuse, TexCoords));
165.     vec3 diffuse = light.diffuse * diff * vec3(texture(material.diffuse, TexCoords));
166.     vec3 specular = light.specular * spec * vec3(texture(material.specular, TexCoords));
167.
168.     ambient *= attenuation * intensity;
169.     diffuse *= attenuation * intensity;
170.     specular *= attenuation * intensity;
171.
172.     return (ambient + diffuse + specular);
173.
174. }

```

## lighting.vs

El archivo lighting.vs corresponde al vertex shader encargado de procesar cada uno de los vértices utilizados en la escena 3D. Su función principal es transformar la posición original de los vértices, preparar la información necesaria para los cálculos de iluminación y asegurar que las texturas se apliquen correctamente en el fragment shader. Este shader constituye la primera etapa del pipeline gráfico programable de OpenGL.

El shader recibe tres atributos fundamentales:

- **position:** representa la posición del vértice en el espacio local del modelo.
- **normal:** vector normal asociado al vértice, necesario para determinar la interacción de la luz con las superficies.
- **texCoords:** coordenadas UV que indican qué parte de la textura corresponde a cada vértice.

De igual forma, el shader utiliza tres matrices enviadas desde la aplicación principal:

- **model**, que transforma el objeto desde su espacio local al espacio global;
- **view**, que corresponde a la transformación de la cámara dentro de la escena;
- **projection**, que aplica la perspectiva necesaria para generar la profundidad visual de la escena.

Dentro de la función main(), el vertex shader ejecuta las operaciones esenciales para preparar el vértice:

1. Transformación de posición:

$$gl\_Position = projection * view * model * vec4(position, 1.0f);$$

Con esta instrucción, el vértice pasa del espacio local al espacio de clip, permitiendo su correcta representación en pantalla.

2. Cálculo de la posición global del vértice:

$$FragPos = vec3(model * vec4(position, 1.0f));$$

Esta posición es indispensable para los cálculos de iluminación en el fragment shader.

3. Transformación de la normal:

$$Normal = mat3(transpose(inverse(model))) * normal;$$

Las normales requieren una transformación especial mediante la matriz normal para preservar su orientación incluso cuando el modelo ha sido escalado o rotado.

4. Asignación de coordenadas de textura:

5. TexCoords = texCoords;

Las coordenadas UV se envían sin modificación para asegurar un texturizado preciso en las siguientes etapas.

```
1      #version 330 core
2      layout (location = 0) in vec3 position;
3      layout (location = 1) in vec3 normal;
4      layout (location = 2) in vec2 texCoords;
5
6      out vec3 Normal;
7      out vec3 FragPos;
8      out vec2 TexCoords;
9
10     uniform mat4 model;
11     uniform mat4 view;
12     uniform mat4 projection;
13
14     void main()
15     {
16         gl_Position = projection * view * model * vec4(position, 1.0f);
17         FragPos = vec3(model * vec4(position, 1.0f));
18         Normal = mat3(transpose(inverse(model))) * normal;
19         TexCoords = texCoords;
20     }
```

## modelLoading.frag

El archivo **modelLoading.frag** corresponde al fragment shader utilizado durante la carga y renderización de modelos importados mediante la clase *Model*. Su función principal es obtener el color final de cada fragmento a partir de una textura difusa, aplicando además un control de transparencia para descartar los fragmentos que no deben renderizarse. Este shader se emplea en objetos que no requieren cálculos de iluminación complejos y cuyo aspecto depende directamente de la textura aplicada.

El shader recibe como entrada las coordenadas de textura (**TexCoords**) generadas y enviadas previamente desde el vertex shader. A partir de estas coordenadas, accede a la textura asociada mediante el uniform:

```
uniform sampler2D texture_diffuse1;
```

El proceso principal se realiza dentro de la función main(). Primero, el shader obtiene el color del fragmento consultando la textura:

```
vec4 texColor = texture(texture_diffuse1, TexCoords);
```

Este valor contiene los componentes **RGB** y el canal **alfa**, el cual es utilizado para determinar si el fragmento debe mostrarse o descartarse. El shader aplica una verificación de transparencia:

```
if(texColor.a < 0.1)
    discard;
```

Si el valor alfa es muy bajo (menor a 0.1), el fragmento se descarta, lo que permite que zonas transparentes de la textura no aparezcan en pantalla. Esto es fundamental para modelos que contienen objetos parcialmente transparentes, recortes o contornos definidos mediante alfa.

Finalmente, si el fragmento no es descartado, el shader asigna el color obtenido directamente al pixel de salida:

```
color = texColor;
```

De esta manera, el color final dependerá únicamente de la textura, sin modificaciones adicionales ni cálculos de luz.

```
1 #version 330 core
2 in vec2 TexCoords;
3
4 out vec4 color;
5
6 uniform sampler2D texture_diffuse1;
7
8 void main()
9 {
10     vec4 texColor = texture(texture_diffuse1, TexCoords);
11     if(texColor.a < 0.1)
12         discard;
13     color = texColor;
14 }
```

## modelLoading.frag

Este vertex shader tiene como propósito transformar las posiciones de los vértices del modelo para su correcta proyección en pantalla, además de enviar las coordenadas de textura al fragment shader. El shader recibe la posición del vértice, su normal y las coordenadas UV, aunque en este caso únicamente utiliza las coordenadas de textura.

Dentro de la función main(), el shader asigna directamente las coordenadas UV:

*TexCoords = aTexCoords;*

Posteriormente, calcula la posición final del vértice multiplicándolo por las matrices model, view y projection, lo cual coloca el objeto en su posición dentro de la escena y aplica la perspectiva necesaria:  
`gl_Position = projection * view * model * vec4(aPos, 1.0);`

Este shader es sencillo y está diseñado para modelos que únicamente requieren transformación espacial y aplicación de texturas, sin necesidad de cálculos de iluminación en esta etapa.

```
1 #version 330 core
2 layout (location = 0) in vec3 aPos;
3 layout (location = 1) in vec3 aNormal;
4 layout (location = 2) in vec2 aTexCoords;
5
6 out vec2 TexCoords;
7
8 uniform mat4 model;
9 uniform mat4 view;
10 uniform mat4 projection;
11
12 void main()
13 {
14     TexCoords = aTexCoords;
15     gl_Position = projection * view * model * vec4(aPos, 1.0);
16 }
```

## CONCLUSIONES

El desarrollo de este proyecto permitió integrar de manera práctica los conocimientos adquiridos en modelado 3D y programación gráfica, abarcando todo el proceso desde la creación de objetos en Blender hasta su implementación e interacción dentro de un entorno renderizado con OpenGL. La construcción de modelos, su texturizado, la aplicación de animaciones, el uso de shaders personalizados y la configuración de distintos tipos de iluminación proporcionaron una comprensión completa del flujo de trabajo en aplicaciones gráficas. Este proyecto consolidó habilidades técnicas y reforzó la importancia de la planificación, la organización por etapas y la coherencia visual, permitiendo crear una escena funcional, dinámica y estéticamente integrada.

## REFERENCIAS

- Ing. Espinoza Urzúa, E. Curso de Computación Gráfica Semestre 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). Recuperado de <https://www.assimp.org/>
- LearnOpenGL. (n.d.). LearnOpenGL. Recuperado de <https://learnopengl.com/>
- GLFW. (n.d.). Graphics Library Framework. Recuperado de <https://www.glfw.org/>
- GLEW. (n.d.). The OpenGL Extension Wrangler Library. Recuperado de <http://glew.sourceforge.net/>
- GitHub Docs. (n.d.). Managing repositories. Recuperado de <https://docs.github.com/en/repositories>