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.

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

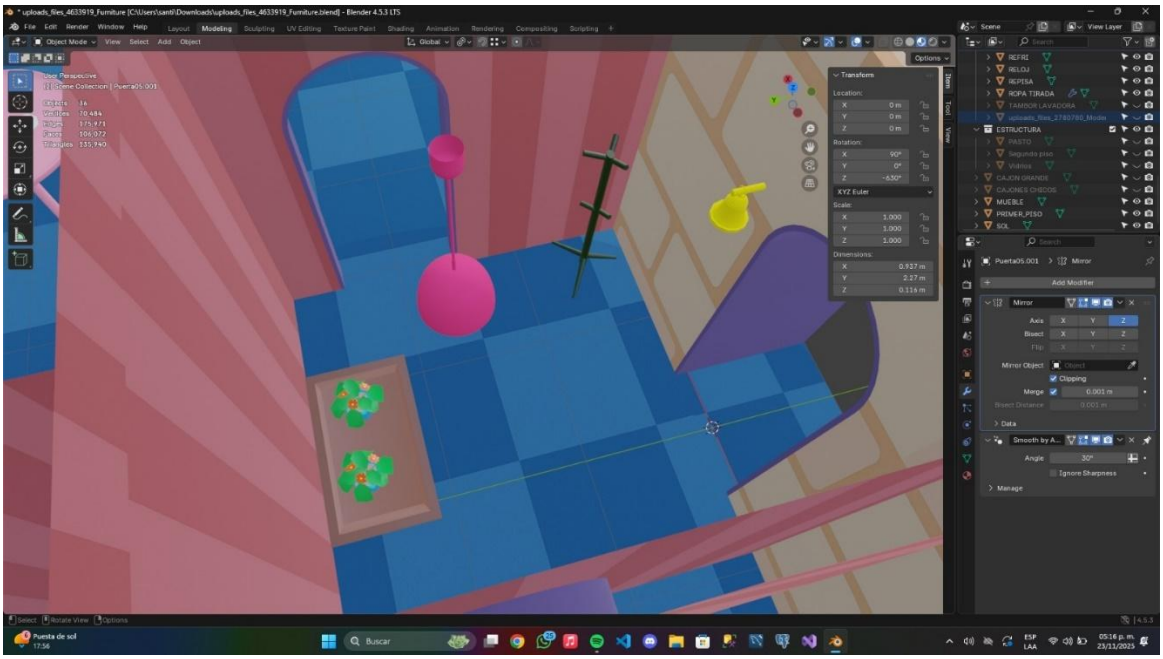
En esta sección se describen los cuartos desarrollados dentro de la escena 3D, así como los objetos modelados, texturizados e integrados en cada espacio. Se incluyen también las animaciones implementadas y los elementos interactivos definidos durante el desarrollo del proyecto.

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



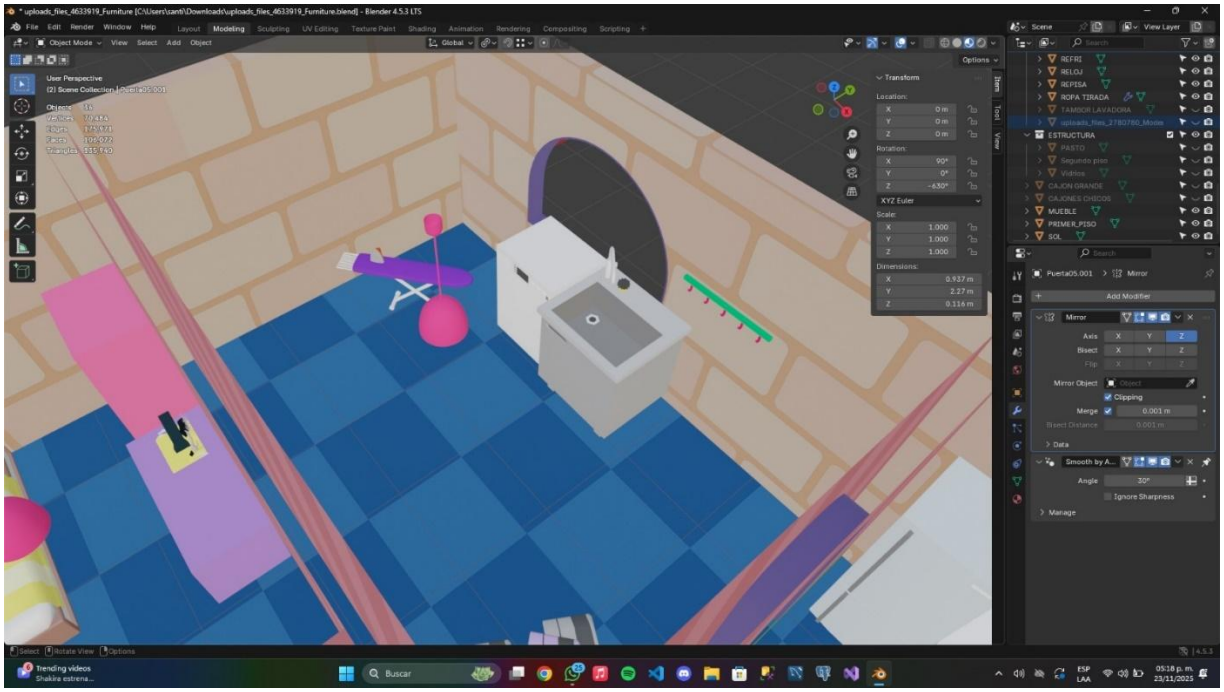
Imágenes Cuarto 01

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.

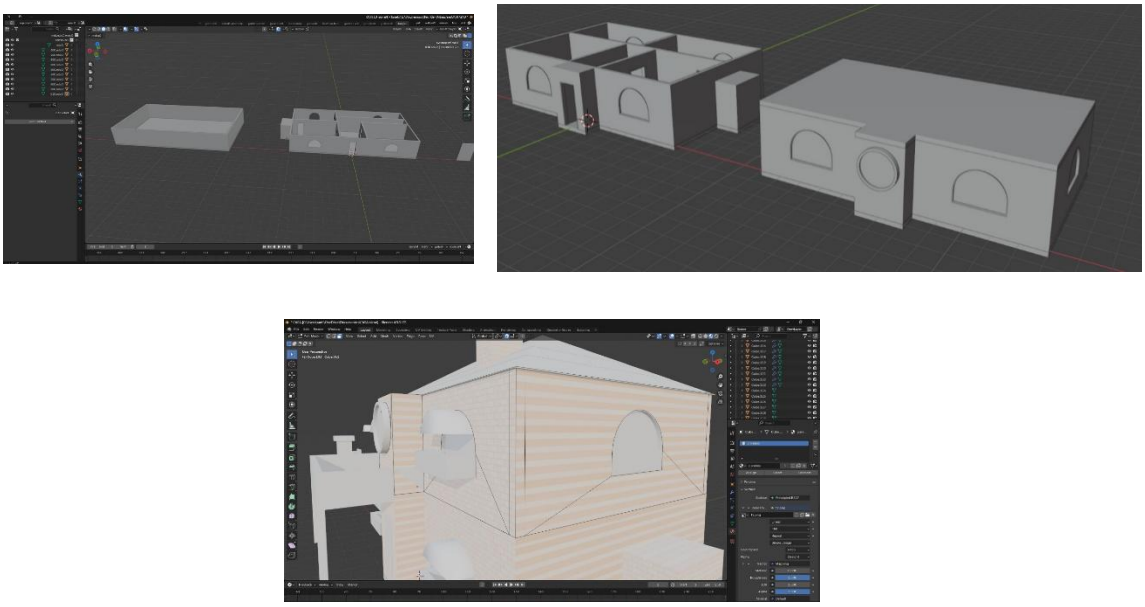
2. Diseño y modelado 3D.

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.

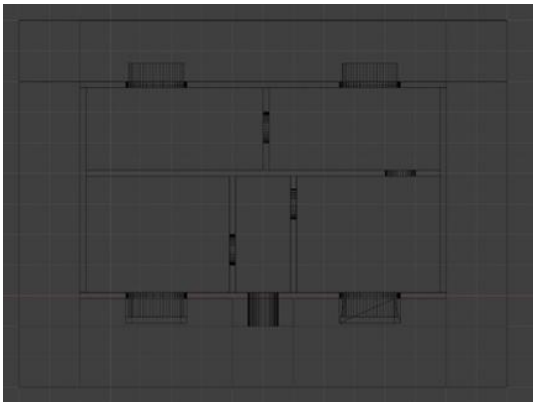
2.1 Construcción Inicial de la Fachada.
La estructura principal se modeló a partir de un cubo inicial, al cual se le aplicaron extrusiones para formar las paredes exteriores y los volúmenes de la entrada. Sin embargo, esta primera versión presentó problemas relacionados con el grosor de las paredes: 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.



Imágenes 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.

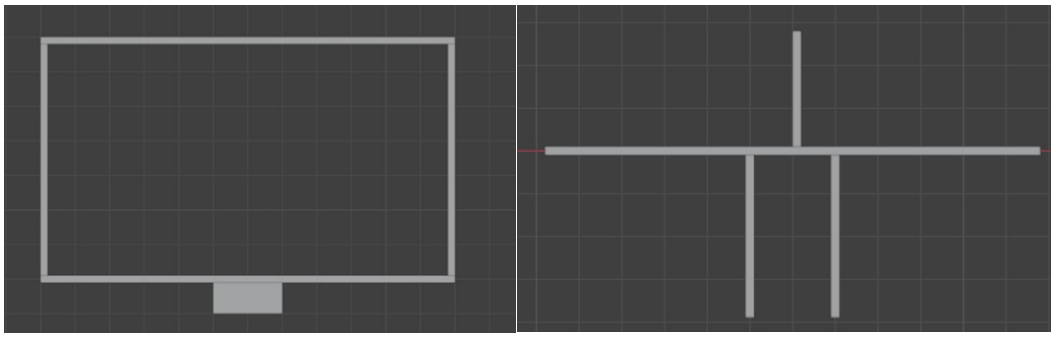
   Este procedimiento permitió construir una estructura sólida, topológicamente coherente y libre de los problemas detectados en las primeras versiones.

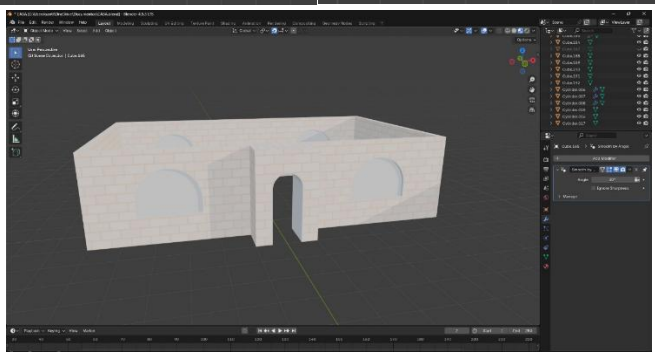
2.3 Desarrollo de la Fachada y Muros Interiores.

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.





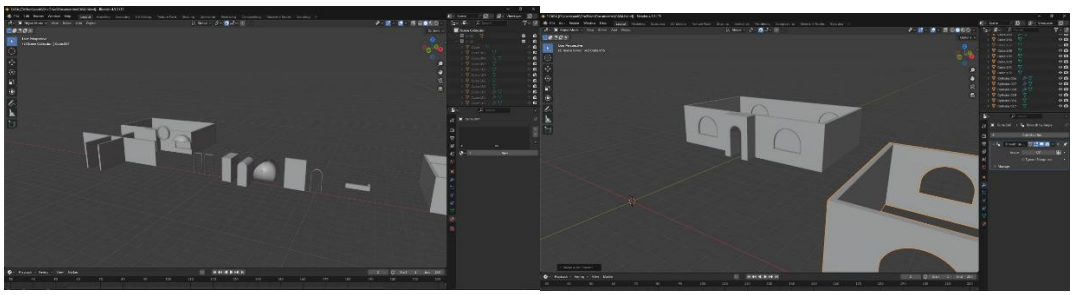
Imágenes Vista de la casa tras su reconstrucción con texturas base y distribución de habitaciones.

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.



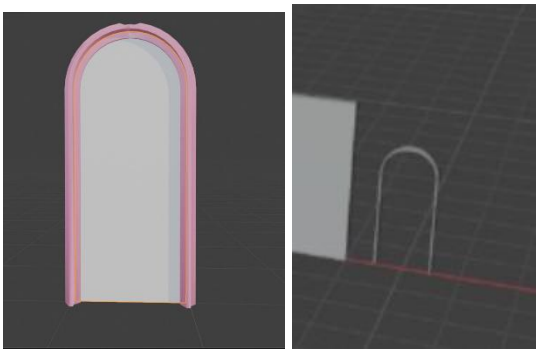
Imágenes Vista de la casa tras su reconstrucción con texturas base y distribución de habitaciones.

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.

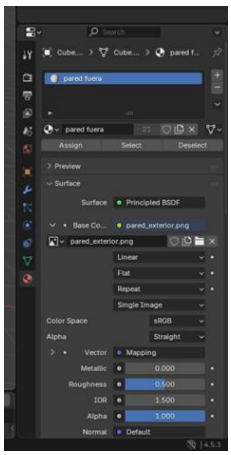


Imágenes Vista de la casa tras su reconstrucción con texturas base y distribución de habitaciones.

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.

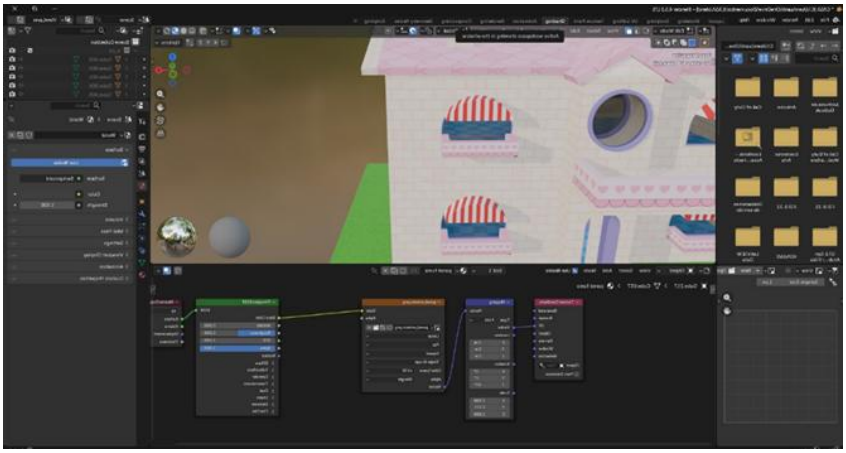


Imagen de Configuración básica de nodos utilizada para manipular las coordenadas de textura.

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.

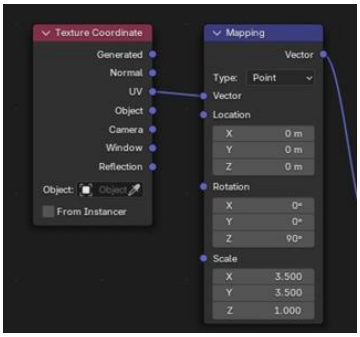


Imagen de Ajuste de rotación y repetición de textura mediante el nodo *Mapping*.

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.

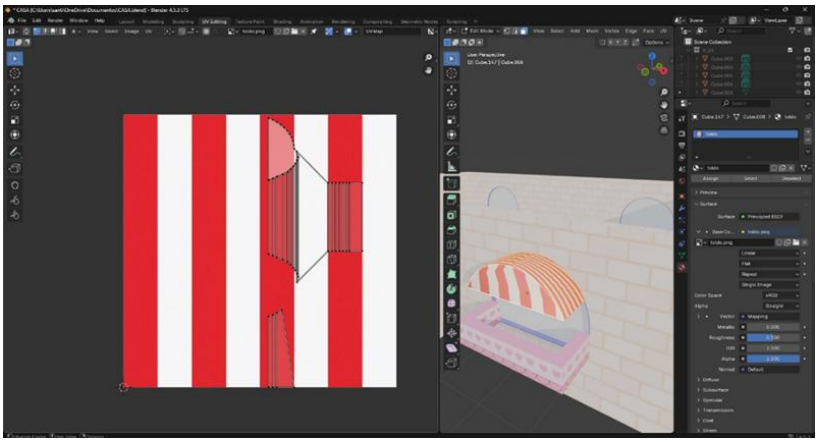


Imagen de Acomodo manual de la isla UV sobre la textura de rayas para corregir distorsiones.

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.

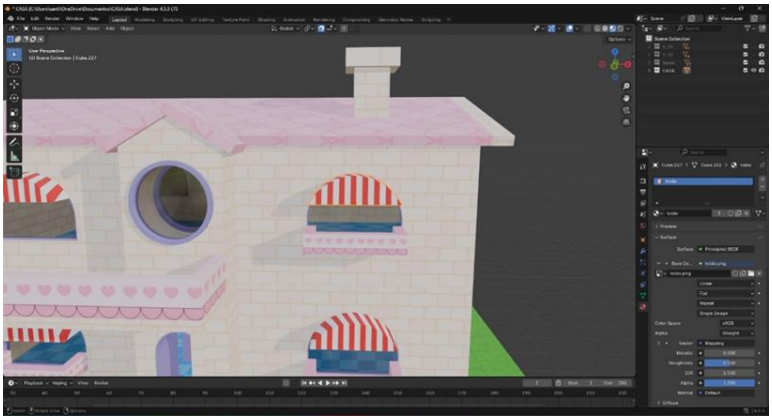
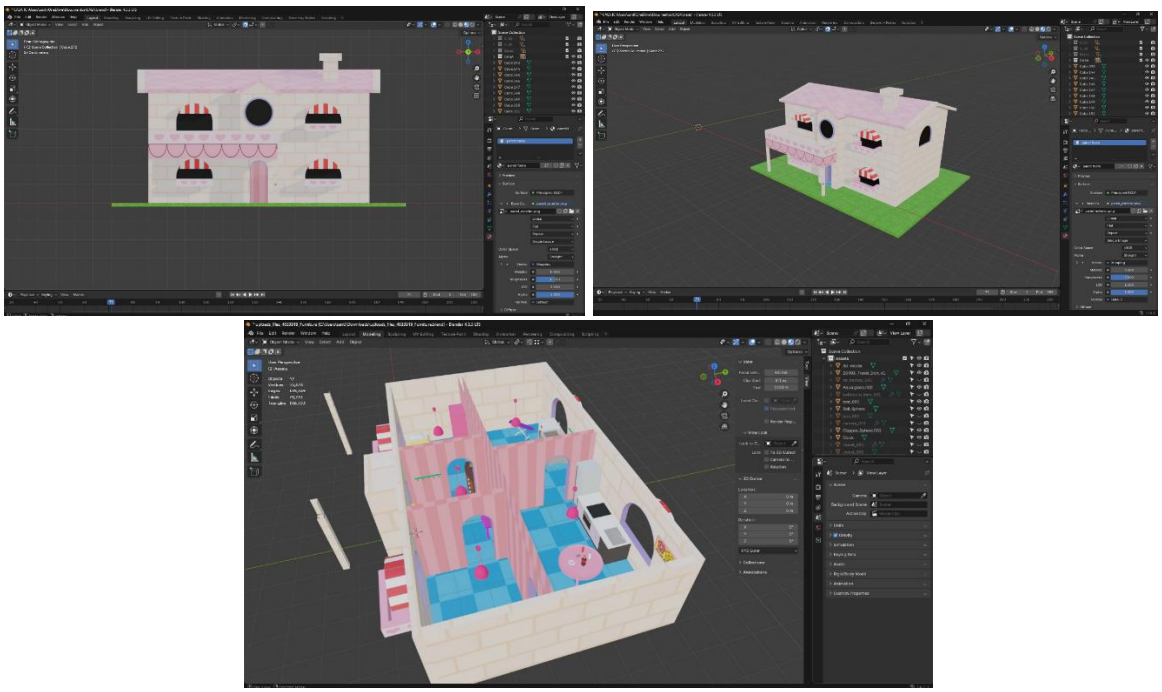


Imagen de Vista final de la fachada completamente texturizada en Blender.

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.

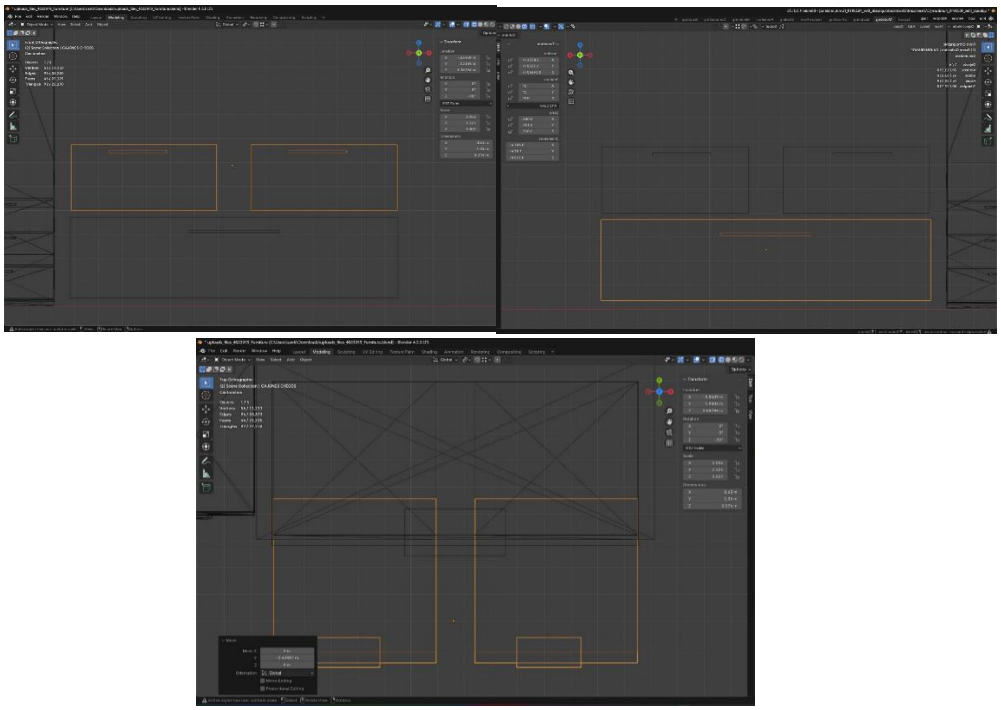


Imágenes finales de la casa.

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

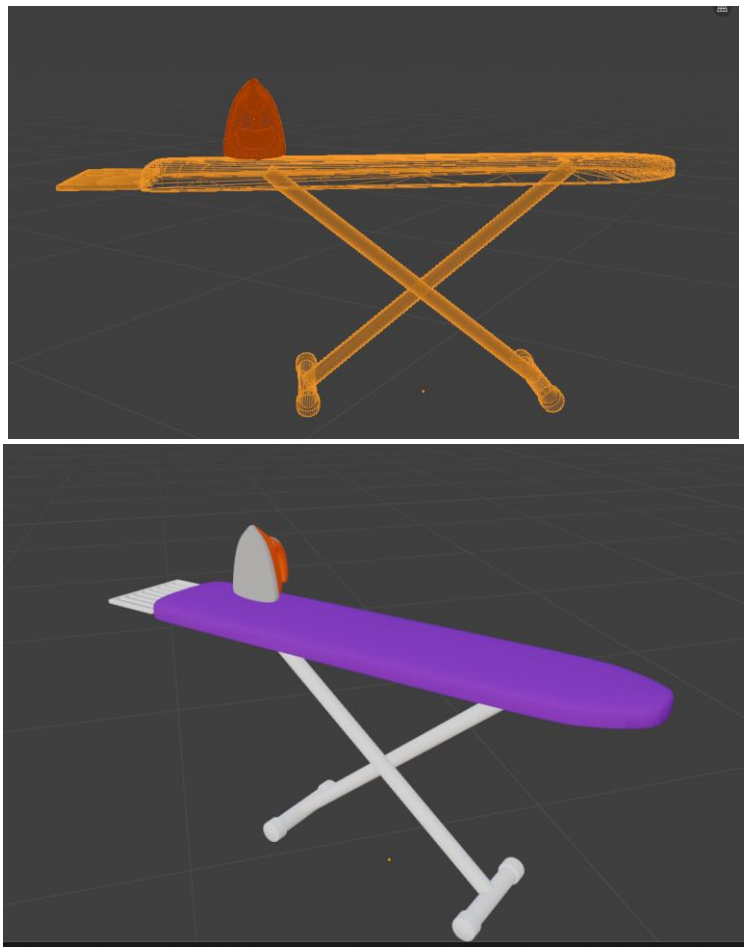
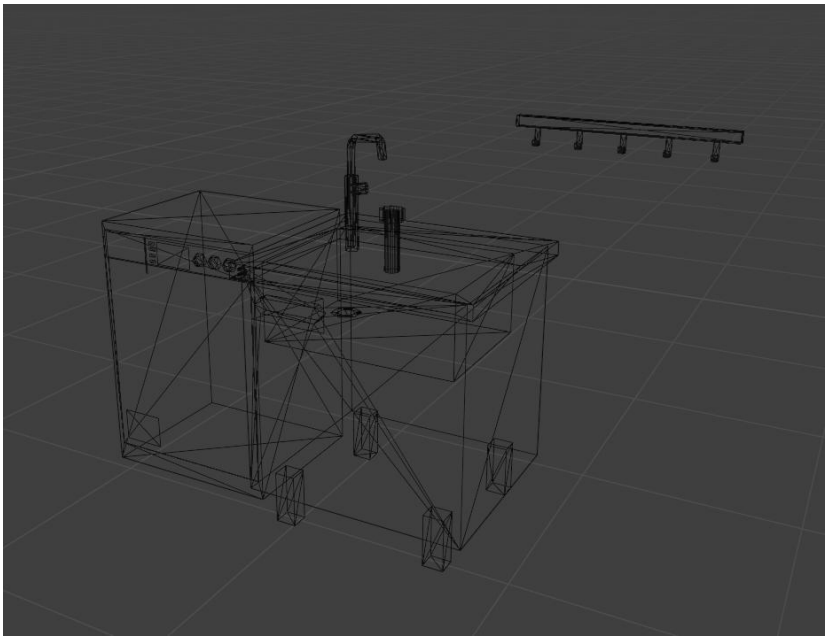


Imagen. Burro de planchar y plancha de ropa.



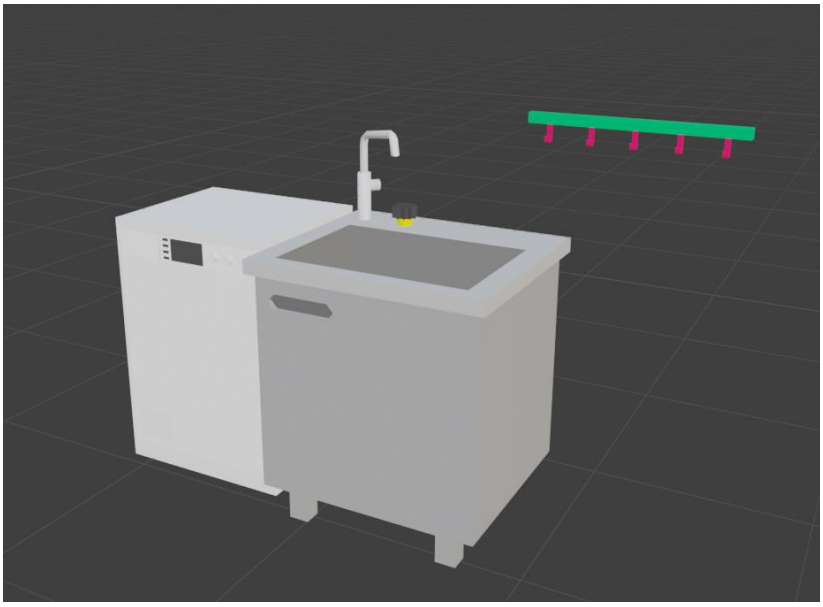


Imagen. Lavadora sin tambor, lavabo del cuarto de lavado y perchero de pared.

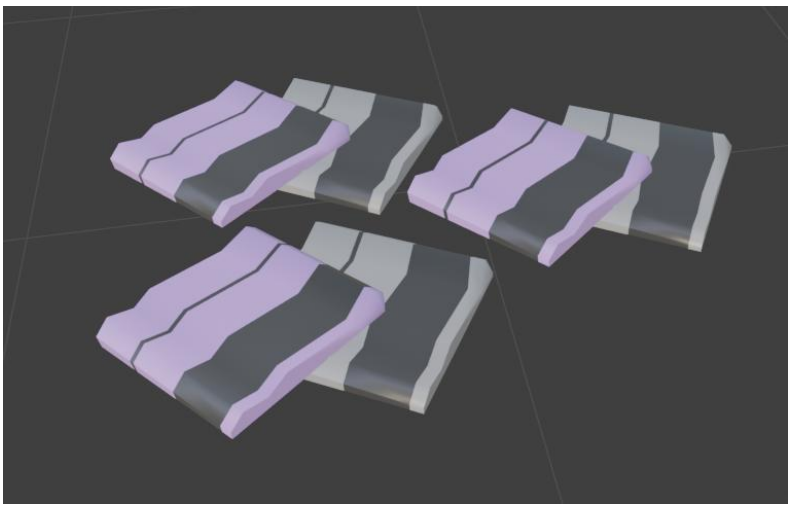


Imagen. Ropa tirada en el cuarto de lavado.

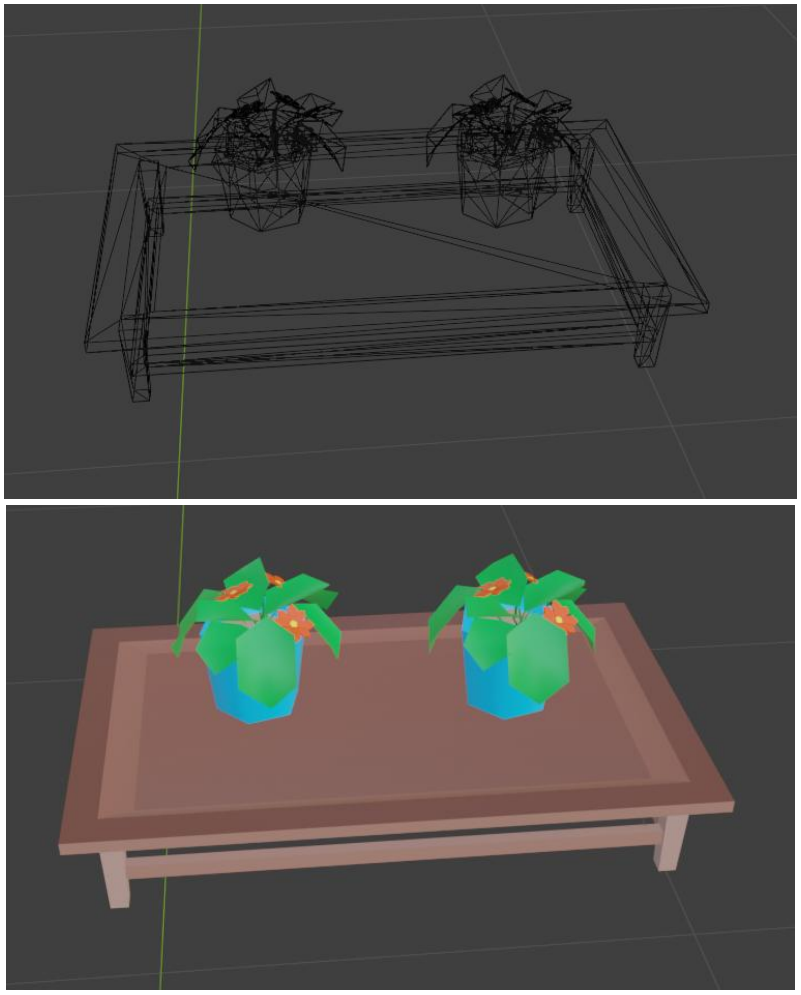
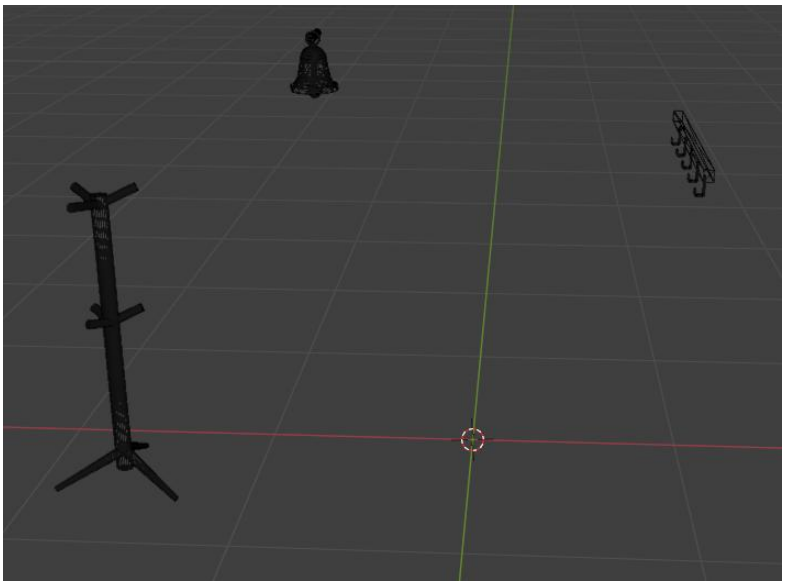


Imagen. Flores sobre mesa.



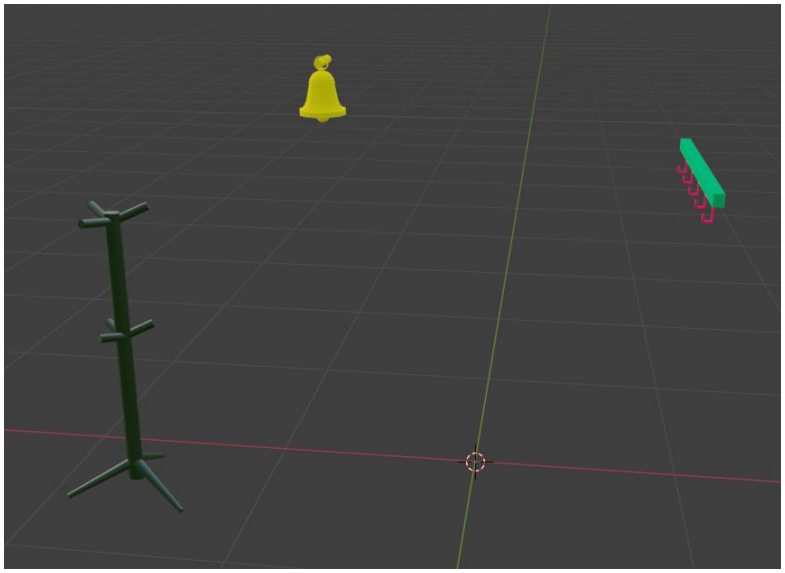


Imagen. Pechero, perchero de pared y campana.

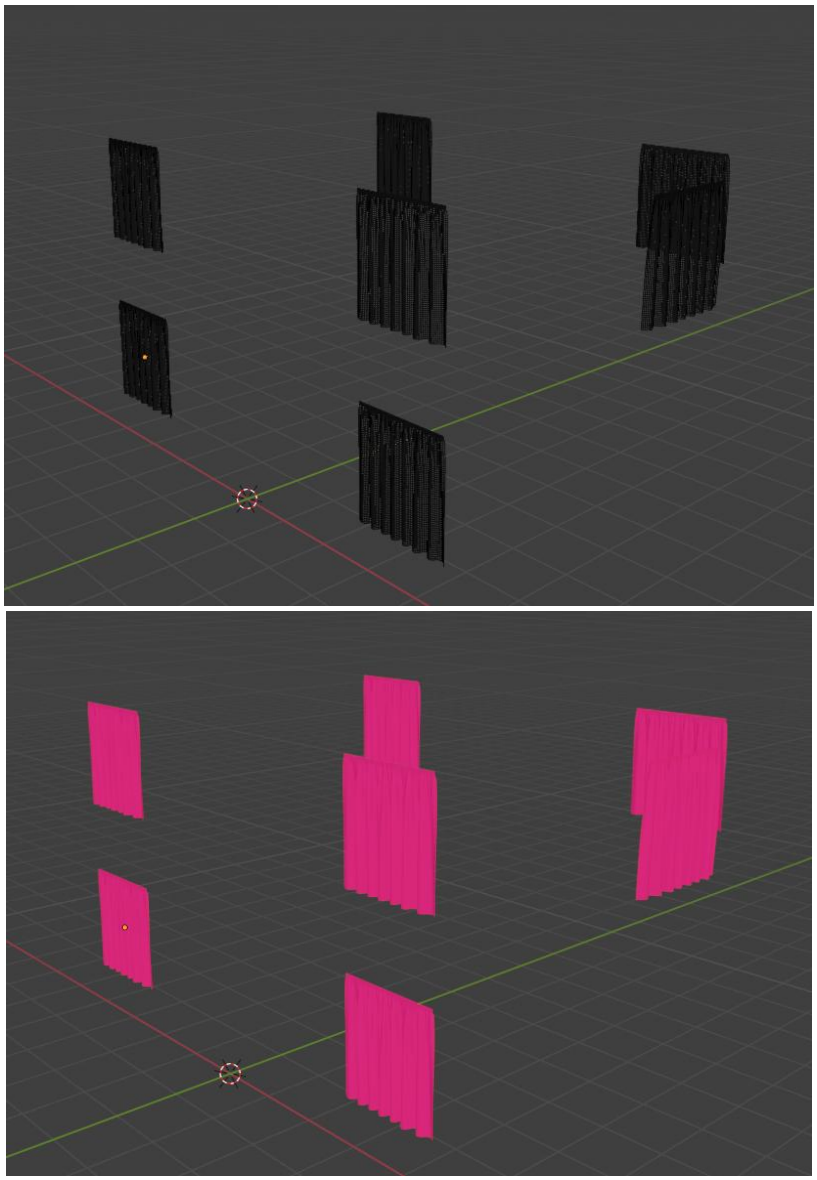


Imagen. Cortinas.

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.

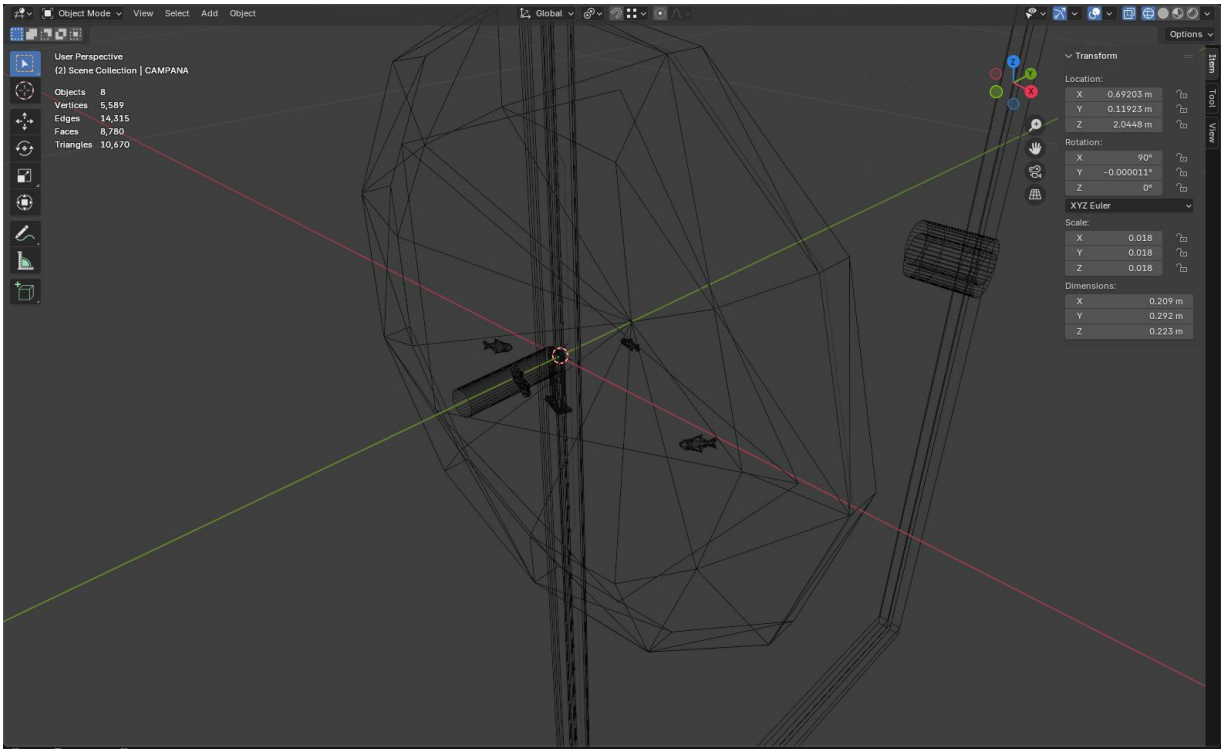


Imagen. Tambor de la lavadora, peces de la pecera, manecilla del reloj

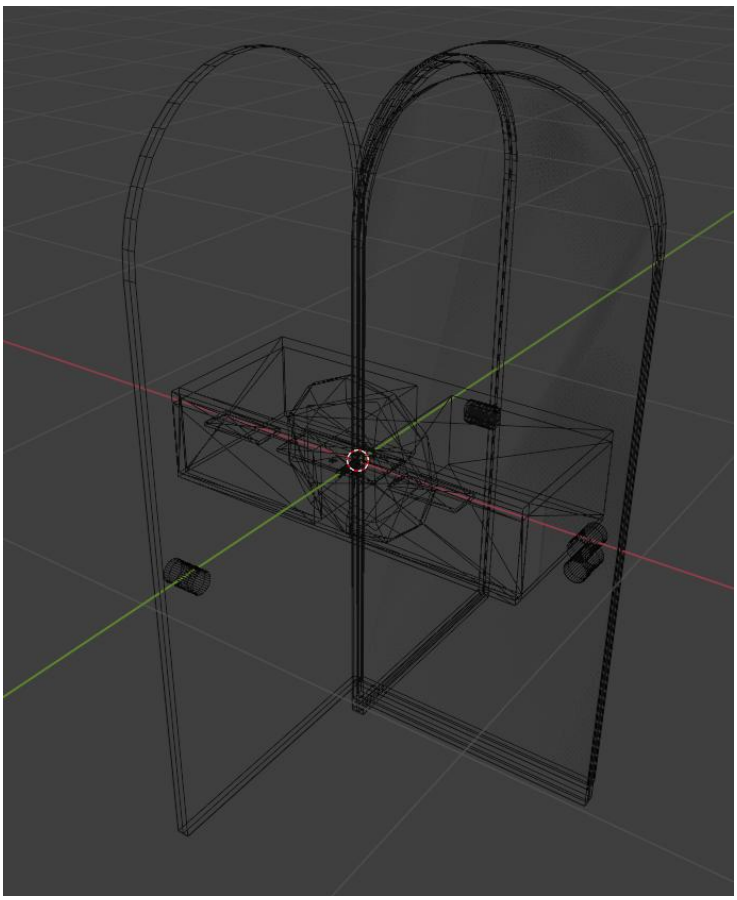


Imagen. Puertas de la casa y cajones.

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 800×600 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 glTexParameteri() 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_IZQUIERDA.obj")
Model cortinas((char*)"Models/CORTINAS_2DO_PISO.obj");
Model lamparas((char*)"Models/LAMPARAS.obj");
Model foco01((char*)"Models/FOCO_CUARTO.obj");
Model foco02((char*)"Models/FOCO_PASILLO.obj");
Model foco03((char*)"Models/FOCO_VACIO.obj");
Model foco04((char*)"Models/FOCO_COCINA.obj");
Model foco05((char*)"Models/FOCO_LAV.obj");
Model sol((char*)"Models/SOL.obj");
```

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 texturaVapor;
glGenTextures(1, &texturaVapor);
glBindTexture(GL_TEXTURE_2D, texturaVapor);

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

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

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

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

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 traslación: cortinas

Las cortinas forman parte de los objetos animados mediante traslación 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 = !aguaEncendida;
    std::cout << "Agua " << (aguaEncendida ? "ENCENDIDA" : "APAGADA") << std::endl;
}

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

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

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

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

Control de la vista con el mouse: MouseCallback()

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

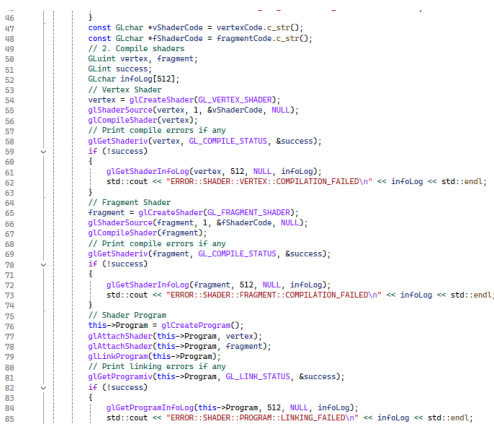
```
void MouseCallback(GLFWwindow* window, double xPos, double yPos)
{
    if (firstMouse)
    {
        lastX = xPos;
        lastY = yPos;
        firstMouse = false;
    }

    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

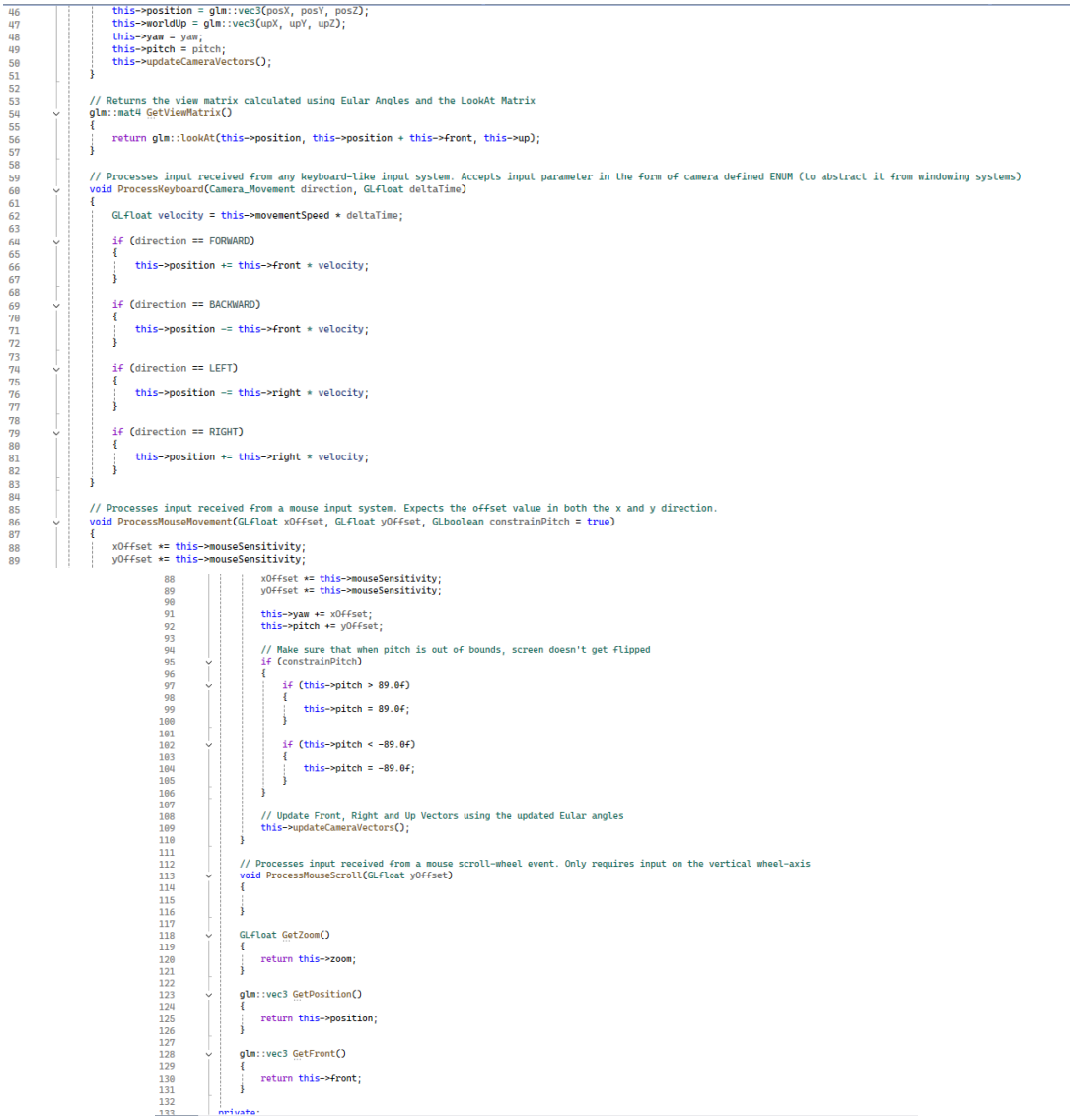
enviar valores desde C++ hacia el shader. En resumen, esta clase automatiza todo el proceso técnico de gestión de shaders, facilitando su uso dentro del motor gráfico.



```
86          }
87          //le damos la localidad de color
88          uniformColor = glGetUniformLocation(this->Program, "color");
89          // Delete the shaders as they're linked into our program now and no longer necessery
90          glDeleteShader(vertex);
91          glDeleteShader(fragment);
92
93      }
94      // Uses the current shader
95      void Use()
96      {
97          glUseProgram(this->Program);
98      }
99
100     GLuint getColorLocation()
101     {
102         return uniformColor;
103     }
104 };
105
106 #endif
```

Camara.h

- La clase Camera implementa la cámara en primera persona que usamos en la escena. Esta clase guarda la posición de la cámara, hacia dónde está mirando (vector front), así como los vectores up y right que definen su orientación en el espacio. A partir de estos datos, la función GetViewMatrix() genera la matriz de vista usando glm::lookAt, que es la que se envía al shader para renderizar todo desde la perspectiva del usuario. La cámara se actualiza con dos tipos de entrada: con ProcessKeyboard() se procesa el movimiento en las direcciones FORWARD, BACKWARD, LEFT, RIGHT, moviendo la posición según la velocidad y el deltaTime; y con ProcessMouseMovement() se actualizan los ángulos de yaw y pitch a partir del movimiento del mouse, limitando el pitch para evitar que la cámara se “voltee” por completo. Cada vez que cambian estos ángulos, la función privada updateCameraVectors() recalcula los vectores front, right y up, manteniendo coherente la orientación de la cámara durante la navegación.



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

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

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

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

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

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

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

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

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

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

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

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

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

    }

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

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

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

private:
```

Billboard.frag

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.

```
#version 330 core
out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D texture1;

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

Billboard.vs

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.

```
#version 330 core
out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D texture1;

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

Lamp.vs:

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.

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



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

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

}
```

lighting.frag

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:

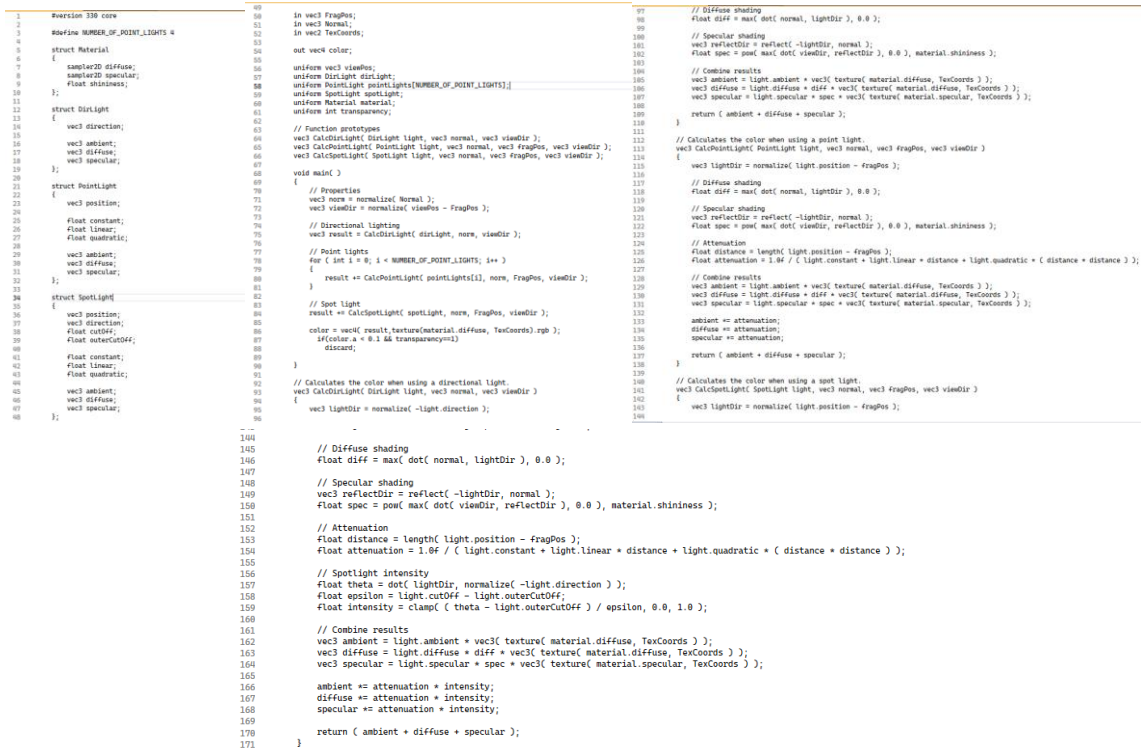
- Primero aplica la **luz direccional** usando CalcDirLight, obteniendo el primer aporte de iluminación.
- Luego recorre un arreglo de **luces puntuales** (pointLights[]) y suma el resultado de cada una usando CalcPointLight.
- Finalmente agrega el aporte del **spotlight** mediante CalcSpotLight.

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.



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.

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

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

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

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

modelLoading.frag

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.

```
#version 330 core
in vec2 TexCoords;

out vec4 color;

uniform sampler2D texture_diffuse1;

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

modelLoading.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.

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

out vec2 TexCoords;

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

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

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