



**UNIVERSIDAD NACIONAL
AUTÓNOMA DE MÉXICO
FACULTAD DE INGENIERÍA**

**INGENIERÍA EN COMPUTACIÓN
LABORATORIO DE COMPUTACIÓN GRÁFICA E
INTERACCIÓN HUMANO COMPUTADORA**

**PROFESOR: ING. EDÉN ESPINOZA URZÚA
SEMESTRE 2025-2**

**PROYECTO FINAL:
Manual De Usuario y Técnico**

ALUMNOS:

González Frías Ana Paula
Orihuela Arellano Santiago

GRUPO:

04

FECHA DE ENTREGA:

15 de noviembre de 2025

ÍNDICE

MANUAL DE USUARIO - PROYECTO FINAL	2
REQUISITOS PREVIOS.....	2
ESTRUCTURA DE LA CARPETA EJECUTABLE	2
CÓMO EJECUTAR EL PROGRAMA.....	3
CONTROLES DEL PROGRAMA.....	4
CARACTERÍSTICAS DE LA ESCENA	5
SOLUCIÓN DE PROBLEMAS.....	6
CIERRE DEL PROGRAMA	6
NOTAS ADICIONALES	¡Error! Marcador no definido.
MANUAL TÉCNICO - PROYECTO FINAL.....	8
DESCRIPCIÓN DEL PROYECTO.....	8
OBJETIVOS	8
DIAGRAMA DE FLUJO DEL SOFTWARE	8
DIAGRAMA DE GANTT	9
ALCANCE DEL PROYECTO.....	11
LIMITANTES	11
METODOLOGÍA DE SOFTWARE APLICADA	11
DOCUMENTACIÓN DEL CÓDIGO	28
CONCLUSIONES	42
REFERENCIAS	43

MANUAL DE USUARIO - PROYECTO FINAL.

Este manual te ayudará a ejecutar y usar el programa “Proyecto Final”, un entorno 3D hecho con OpenGL que incluye iluminación, vista en primera persona y modelos 3D. Sigue las indicaciones con atención para aprovechar al máximo la experiencia.

REQUISITOS PREVIOS.

El sistema operativo necesario para ejecutar el programa es Windows, ya que trabaja con las bibliotecas incluidas. Además, es importante revisar que cuenten con todos los archivos de la carpeta del ejecutable antes de comenzar.

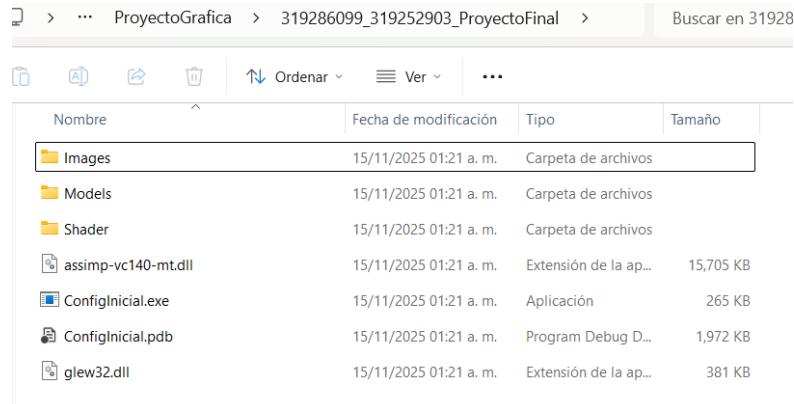


Imagen 1. Archivos necesarios en la carpeta “Ejecutable”

ESTRUCTURA DE LA CARPETA EJECUTABLE.

Esta sección describe el contenido de cada carpeta y archivo incluido en el programa dentro de la carpeta del ejecutable.

- **Images:** Carpeta que contiene las imágenes y texturas utilizadas dentro de la aplicación, como gráficos de gotas y vapor, entre otros elementos visuales.
- **Models:** Carpeta que contiene los archivos de modelos 3D usados para construir la casa y sus componentes. Incluye una amplia colección de archivos .obj y .mtl que representan objetos como muebles, puertas y partes de la estructura necesarios para renderizar la casa.
- **Shader:** Carpeta que contiene los shaders del programa. Estos archivos controlan la iluminación, el color y el estilo visual de los objetos renderizados. Incluye shaders como (lighting.vs, lighting.frag, lamp.vs, lamp.frag, core.vs, core.frag) que controlan la iluminación y el renderizado.
- **assimp-vc140-mt.dll:** Biblioteca utilizada para cargar archivos de modelos 3D en formatos compatibles. Es indispensable para que la aplicación pueda leer los objetos almacenados en la carpeta Models.
- **ConfigInicial.exe:** Archivo ejecutable que contiene configuraciones iniciales o ajustes previos al funcionamiento principal del programa.
- **ConfigInicial.pdb:** Archivo de depuración generado por el entorno de desarrollo. No requiere modificación y solo se utiliza para diagnóstico técnico en caso de errores.
- **glew32.dll:** Biblioteca encargada de habilitar extensiones de OpenGL necesarias para el renderizado 3D. El programa requiere este archivo para funcionar correctamente.

Nota: Es importante mantener esta estructura sin modificaciones, ya que cada elemento cumple una función necesaria para el correcto funcionamiento del entorno 3D.

CÓMO EJECUTAR EL PROGRAMA.

1. Abrir la Carpeta Ejecutable:

Accede, desde el Explorador de archivos de Windows, a la carpeta donde se encuentran todos los archivos del programa.

Verifica que la estructura esté completa y que no falte ninguna carpeta o archivo necesario.

2. Ejecutar el Programa:

Dentro de la carpeta localiza el archivo ejecutable (.exe) dentro de la carpeta.

Para iniciar haz doble clic en el archivo .exe (*ConfigInicial.exe*) para iniciar el programa.

Si Windows muestra una advertencia de seguridad, selecciona “Ejecutar de todos modos” para permitir que el programa se abra.



Imagen 2. Archivo .exe

3. Vista inicial del programa:

Al iniciar el programa, se abre una ventana con el nombre del proyecto en la parte superior. En pantalla aparece la escena 3D donde se muestra la **casa de Hello Kitty**, colocada sobre un pequeño terreno y renderizada con sus colores y detalles característicos.

Esta es la primera vista que aparece al ejecutar la aplicación y corresponde a la posición inicial de la cámara.

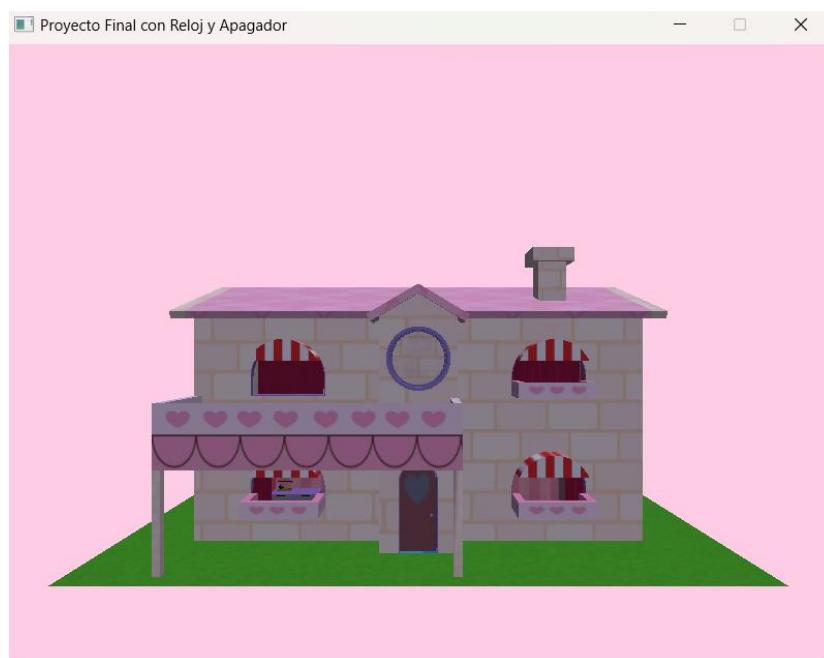


Imagen 3. Escena 3D

CONTROLES DEL PROGRAMA.

El programa permite mover la cámara, interactuar con el personaje y controlar algunas luces mediante el teclado y el ratón. A continuación se describen los controles disponibles:

Control de la Cámara (Tercera Persona)

- **W o Flecha Arriba:** desplaza la cámara hacia la parte frontal de la escena.
- **S o Flecha Abajo:** desplaza la cámara hacia la parte posterior.
- **A o Flecha Izquierda:** mueve la cámara lateralmente hacia la izquierda.
- **D o Flecha Derecha:** mueve la cámara lateralmente hacia la derecha.
- **Ratón:** permite rotar la vista en cualquier dirección; el cursor permanece desactivado para facilitar el movimiento.

Otras Funciones

- **ESC:** finaliza la ejecución del programa y cierra la ventana.

Control de la Luz Puntual

- **L:** Activa/desactiva el sistema de iluminación artificial interior compuesto por 5 fuentes de luz puntual.
- **U:** Activa/desactiva la fuente de luz direccional que simula iluminación solar exterior.

NOTA: En este código solo hay 2 controles de iluminación. Las luces están en posiciones fijas definidas dentro de nuestro arreglo de Focos y nuestro Sol.

Animación de Puertas y Cortinas

- **P:** controla la apertura y cierre de la *puerta principal* de la casa.
- **O:** activa o desactiva simultáneamente las *cuatro puertas interiores* del modelo.
- **C:** desplaza horizontalmente las *cortinas* de la ventana principal.

Animaciones del Primer Cuarto



Imagen 4. Cuarto 1

Cajonera:

- **J:** mueve el *cajón grande* del mueble.
- **K:** mueve los *cajones pequeños* de la cajonera.

Animaciones del Cuarto de Lavado

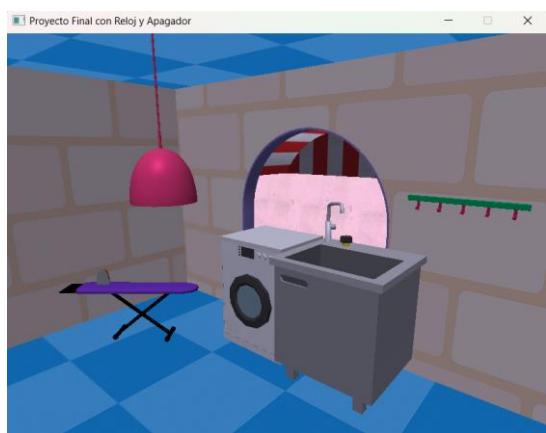


Imagen 5. Cuarto de Lavado

Tambor de lavadora:

- **T:** activa o detiene la *rotación continua* del tambor.

Sistema de partículas (agua y vapor):

- **G:** activa el *flujo de agua* que sale desde la boquilla del lavabo.
- **V:** activa o desactiva el *efecto de vapor ascendente*.

Animación de Cocina

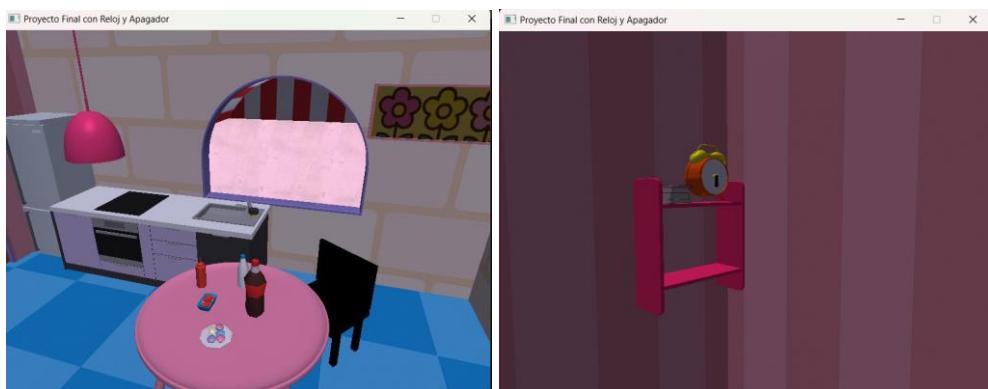


Imagen 6. Cocina

- **M:** activa o detiene el movimiento de las *manecillas* del área de cocina.

CARACTERÍSTICAS DE LA ESCENA.

Modelos 3D

- La escena está conformada por una casa temática de *Hello Kitty*, acompañada de 44 modelos 3D adicionales que representan puertas, muebles, decoración, electrodomésticos, cortinas y objetos animados. Todos los modelos son cargados desde archivos .obj y renderizados con iluminación Phong.

Iluminación

- Luz direccional: Simula la iluminación exterior con un efecto similar a la luz solar. Su dirección base es (-0.3, -1.0, -0.5) y puede encenderse o apagarse mediante el teclado. El modelo 3D del sol aparece solo cuando la luz está activa.

- o Luces puntuales interiores: La casa utiliza **cinco fuentes de luz puntual**, distribuidas en distintas habitaciones (cuarto, pasillo, cuarto vacío, cocina y lavandería). Todas pueden activarse o desactivarse simultáneamente con una sola tecla y tienen una intensidad reducida para crear un ambiente interior tenue.
- o Transparencias: Se utilizan efectos de *blending alfa* para objetos con vidrio o materiales semi-transparentes, como las ventanas, la pecera y las partículas de agua y vapor.
- o Cámara en primera persona: El recorrido se realiza mediante una cámara tipo FPS (First-Person Shooter). El usuario puede avanzar, retroceder, desplazarse lateralmente y girar la vista con el mouse para explorar libremente el interior y exterior de la casa.

SOLUCIÓN DE PROBLEMAS.

El programa no inicia

- Verifique que los archivos de dependencias (glew32.dll, assimp-vc140-mt.dll, se encuentren en el mismo directorio que el archivo ejecutable (.exe).
- Asegúrese de que el usuario tenga permisos para ejecutar aplicaciones. En caso de duda, intente abrir el programa con “Ejecutar como administrador”.
- Si el problema continúa, vuelva a copiar todos los archivos originales de la carpeta del proyecto o recompile la aplicación desde el entorno de desarrollo (por ejemplo, Visual Studio).

La escena no se renderiza correctamente

- Compruebe que las carpetas Images, Models y Shader estén completas y que no falte ningún archivo .obj, .mtl, .png o shader (.vs, .frag).
- Si algún archivo fue movido, renombrado o eliminado, restáurelo a su ubicación original dentro de la carpeta del ejecutable.
- Verifique que las rutas a modelos, texturas y shaders definidas en el código sean válidas.
- Asegúrese de que los controladores de la tarjeta gráfica estén actualizados a la versión más reciente.

Movimiento lento o comportamiento irregular

- Cierre otros programas o procesos que consuman recursos del sistema antes de ejecutar la aplicación.
- En caso de bajo rendimiento, reduzca la calidad de renderizado o desactive efectos visuales avanzados.
- Verifique el uso de la GPU y CPU para asegurarse de que la aplicación esté utilizando el hardware de manera óptima.

Las animaciones o teclas no responden

- Verifique que la ventana del programa esté **activa** (en primer plano) al presionar las teclas.
- Compruebe que el teclado esté configurado correctamente (idioma y distribución) y que no haya teclas bloqueadas.
- Confirme que las teclas de control (P, O, C, J, K, T, G, V, M, R, etc.) no hayan sido modificadas en el código o reasignadas a otras funciones.

CIERRE DEL PROGRAMA

- Para finalizar la ejecución de forma segura, presione la tecla Esc.

- También es posible cerrar la aplicación haciendo clic en el botón “Cerrar” (X) ubicado en la esquina superior derecha de la ventana.
- Se recomienda evitar el cierre forzoso del programa para prevenir posibles errores en la liberación de recursos.

CONSIDERACIONES FINALES.

- Este software ha sido desarrollado con fines **educativos y de demostración**. Se recomienda **no modificar los archivos del proyecto** a menos que se cuente con conocimientos avanzados en **OpenGL y C++**.
- Para realizar personalizaciones en la escena (como cambiar modelos, texturas o shaders), es necesario **recompilar el código fuente** utilizando un entorno de desarrollo compatible, por ejemplo, **Visual Studio**.
- Una vez configurado correctamente, el usuario puede **explorar la escena 3D y experimentar con los controles**, disfrutando del entorno visual inspirado en **Hello Kitty**.

“Cada detalle cuenta cuando el código y la creatividad se unen para dar vida al mundo de Hello Kitty.”

MANUAL TÉCNICO - PROYECTO FINAL.

DESCRIPCIÓN DEL PROYECTO.

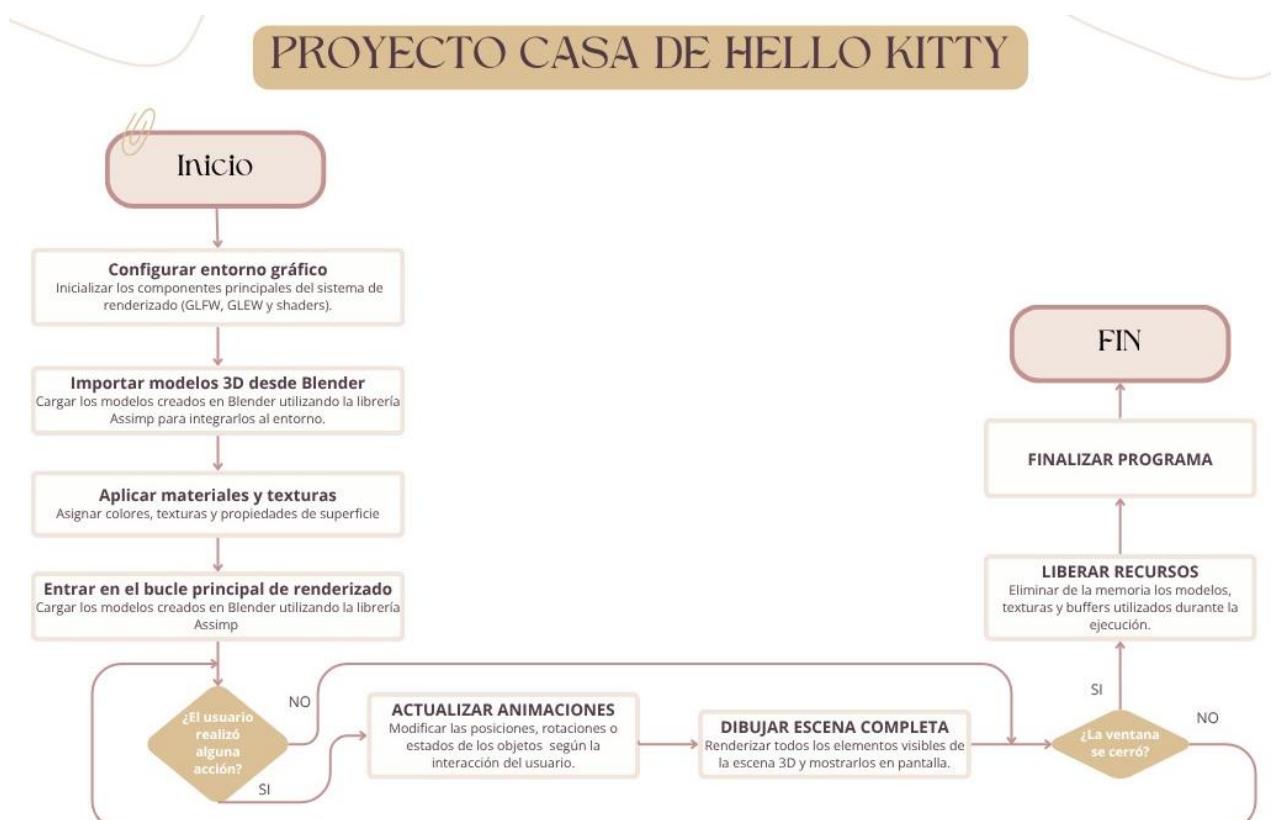
Este proyecto consiste en la **recreación tridimensional de una fachada ficticia inspirada en el universo de Hello Kitty**, diseñada para representar un entorno colorido y amigable característico de su estética. El desarrollo se llevó a cabo utilizando **OpenGL** junto con el lenguaje de programación **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. 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.

OBJETIVOS.

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 y se mostrará el video final, programada para el **15 de noviembre**, 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).

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.

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).

- Sistemas de partículas representando agua y vapor.

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

1.3 Distribución de roles y responsabilidades.

Con base en las habilidades y preferencias de los miembros del equipo, se asignaron responsabilidades específicas para asegurar una ejecución eficiente. El trabajo se dividió en cuatro grandes áreas: modelado, programación, integración y documentación.

Modelado: El modelado de la escena fue realizado por ambos integrantes, aunque Santiago tuvo una participación mayor al encargarse de la mayoría de los objetos desarrollados en Blender. Su trabajo incluyó la construcción de la estructura general de la casa, el diseño del mobiliario y elementos decorativos, así como la elaboración de los objetos destinados a recibir animación. Además, se ocupó de la optimización de las mallas y la exportación final de cada modelo en formato .obj. Ana Paula también colaboró en esta etapa, participando en la definición de referencias visuales y en el planteamiento de la organización de los espacios interiores.

Programación: La programación del entorno en OpenGL fue desarrollada de manera conjunta por Santiago y Ana Paula. Ambos trabajaron en la implementación del sistema de carga de modelos mediante Assimp, así como en la configuración del pipeline de renderizado, los shaders y los materiales. También participaron en el desarrollo del sistema de cámara en primera persona, la interacción mediante teclado y mouse, y en la programación de animaciones, iluminación dinámica y efectos de partículas. Cada integrante se hizo responsable de determinadas animaciones específicas, coordinando su comportamiento para mantener coherencia visual dentro de la escena.

Integración: La integración de los modelos dentro del entorno interactivo fue un proceso colaborativo. Ambos integrantes ajustaron la escala, orientación y posición de cada objeto en la escena, asegurando que la distribución de la casa fuera coherente y funcional. Asimismo, coordinaron la colocación de luces direccionales y puntuales, adaptando sus valores para resaltar la estética de la casa de Hello Kitty. Durante esta etapa también se integraron las animaciones programadas, verificando su correcto comportamiento dentro del entorno 3D.

Documentación: La documentación del proyecto fue elaborada conjuntamente por ambos integrantes. Esto incluyó la redacción del manual técnico, la estructura general de la documentación del software, el registro del proceso de desarrollo y la preparación del material final de entrega. Además, ambos trabajaron en la organización del repositorio en GitHub, la revisión final del proyecto y la presentación de los avances requeridos durante el proceso académico.

2. Diseño y modelado 3D.

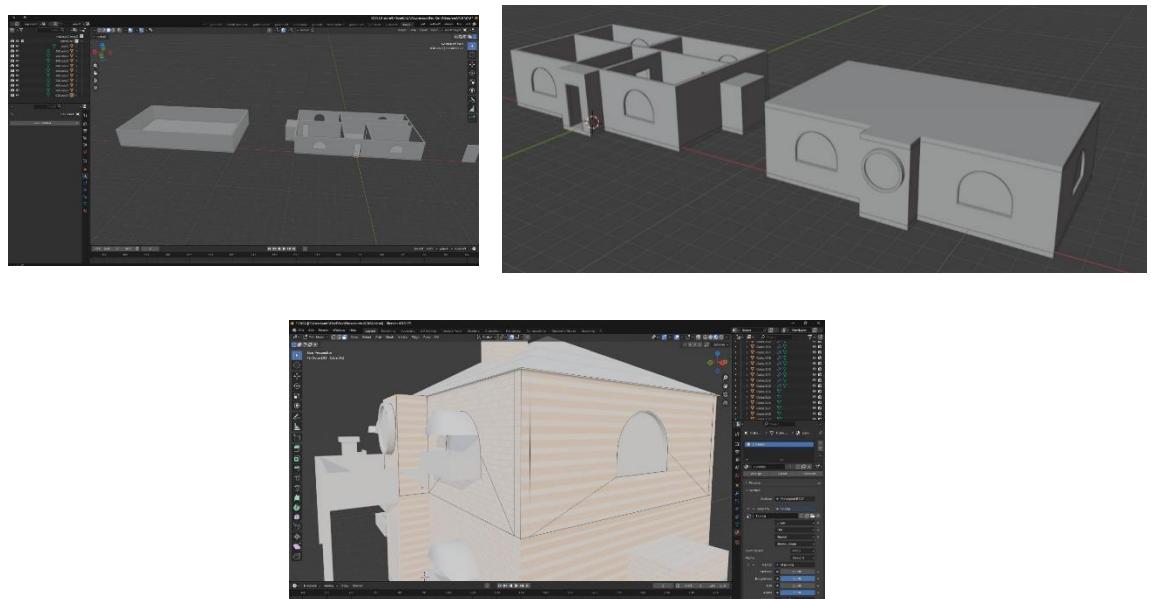
El diseño y modelado 3D del proyecto se realizó en Blender, tomando como base un conjunto de referencias visuales y bocetos que permitieron establecer un estilo coherente con la estética colorida y distintiva de Hello Kitty. A partir de estas referencias se construyó primero la estructura general de 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.

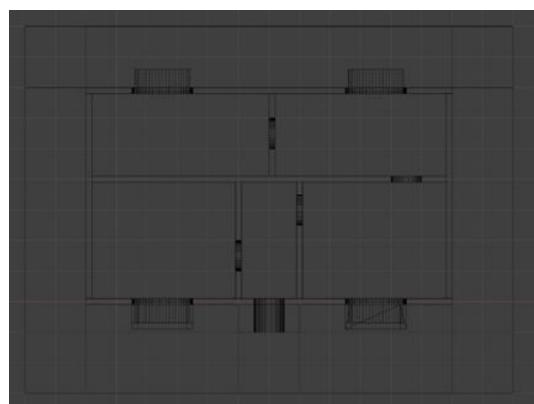


Imágenes iniciales de la estructura de la casa.

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.

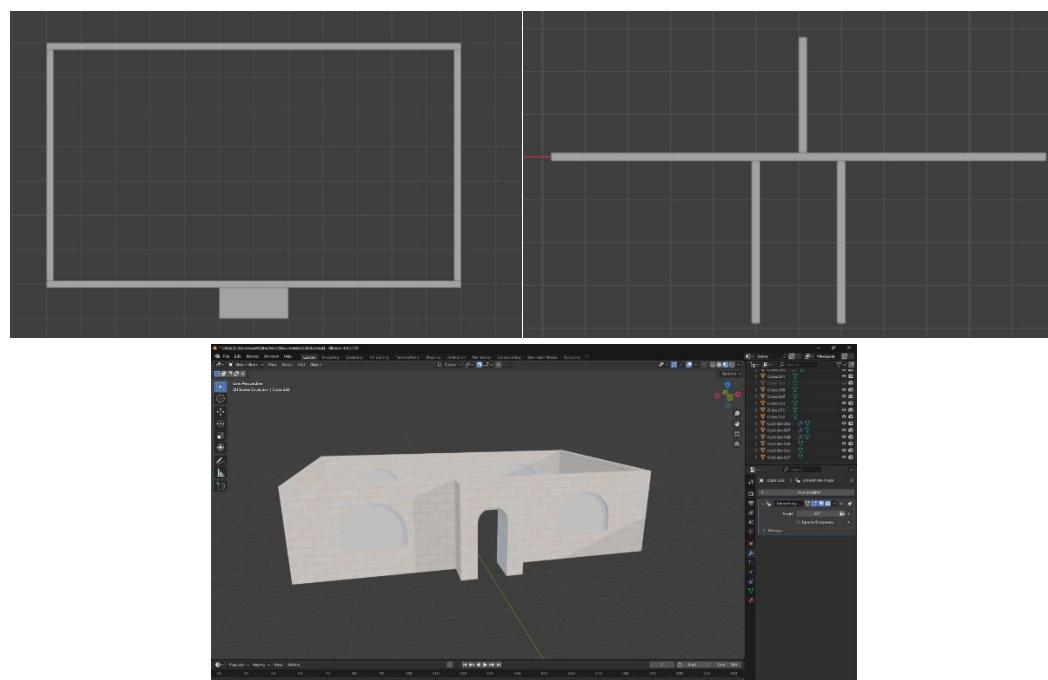
Este procedimiento permitió construir una estructura sólida, topológicamente coherente y libre de los problemas detectados en las primeras versiones.

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.



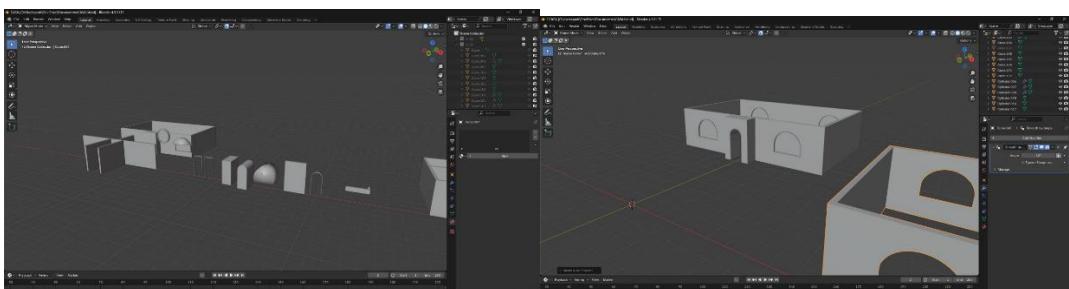
Imágenes Vista de la casa tras su reconstrucción con texturas base y distribución de habitaciones.

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.

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.



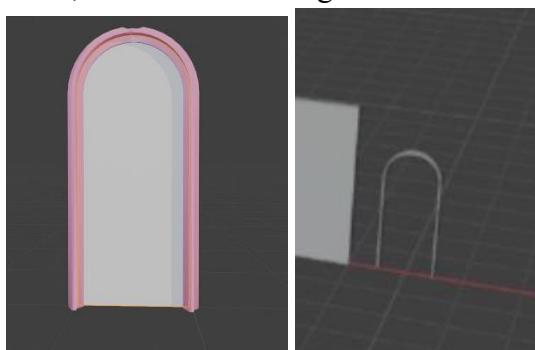
Imágenes Vista de la casa tras su reconstrucción con texturas base y distribución de habitaciones.

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.



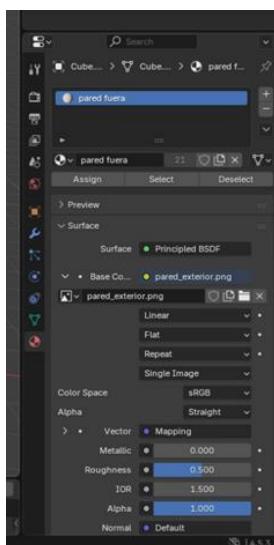
Imágenes Vista de la casa tras su reconstrucción con texturas base y distribución de habitaciones.

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.

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.

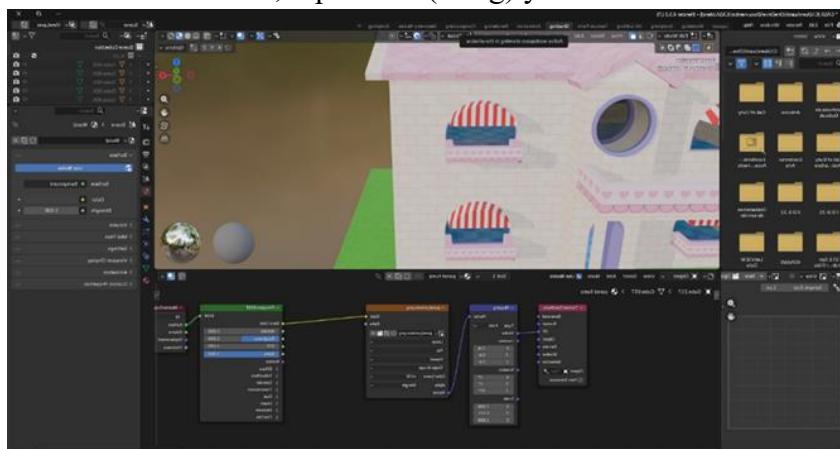


Imagen de Configuración básica de nodos utilizada para manipular las coordenadas de textura.

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.

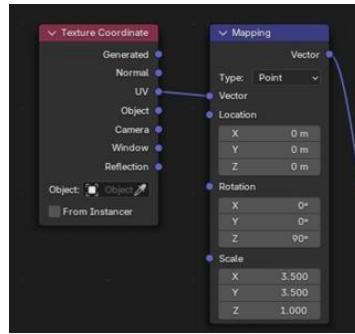


Imagen de Ajuste de rotación y repetición de textura mediante el nodo *Mapping*.

Texturizado específico de superficies curvas (toldos).

Los toldos presentan geometría semicircular, 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.

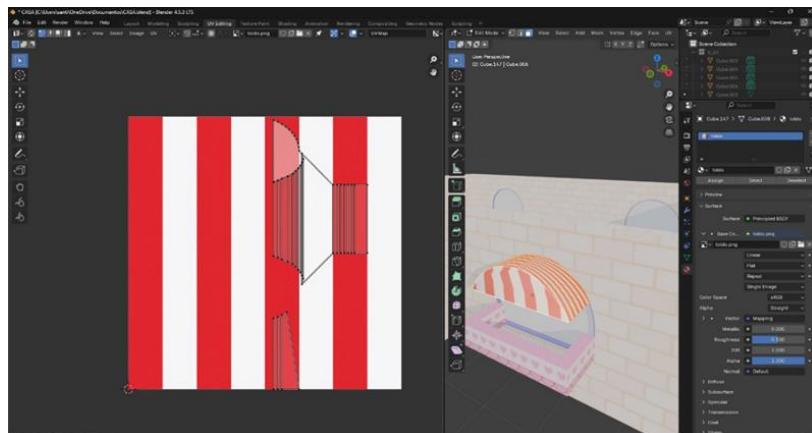


Imagen de Acomodo manual de la isla UV sobre la textura de rayas para corregir distorsiones.

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.

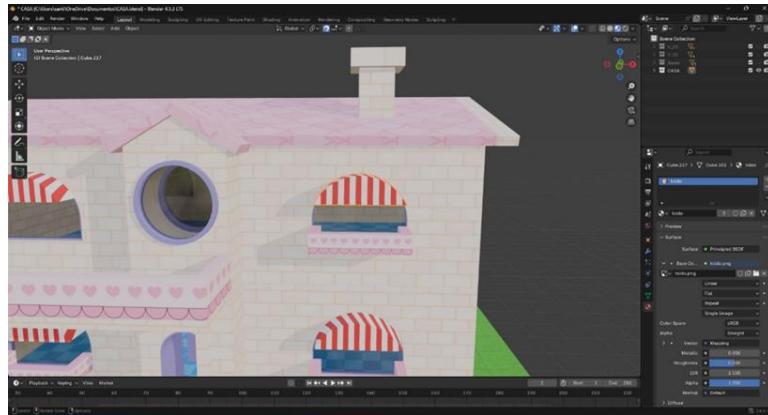
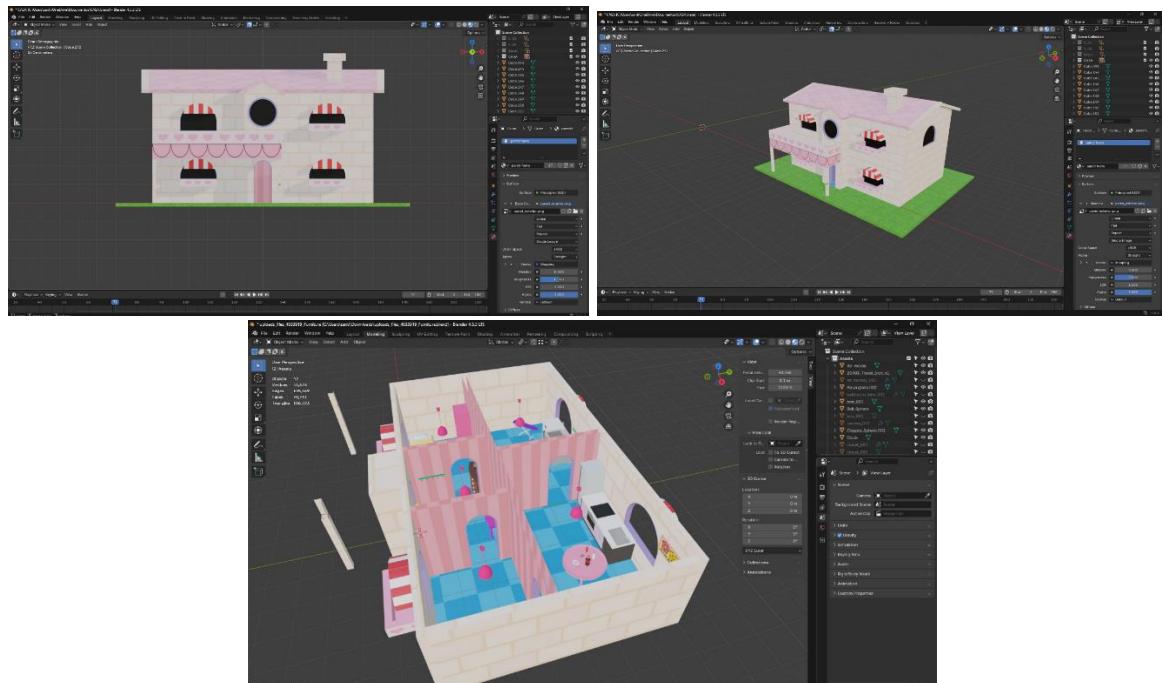


Imagen de Vista final de la fachada completamente texturizada en Blender.

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.

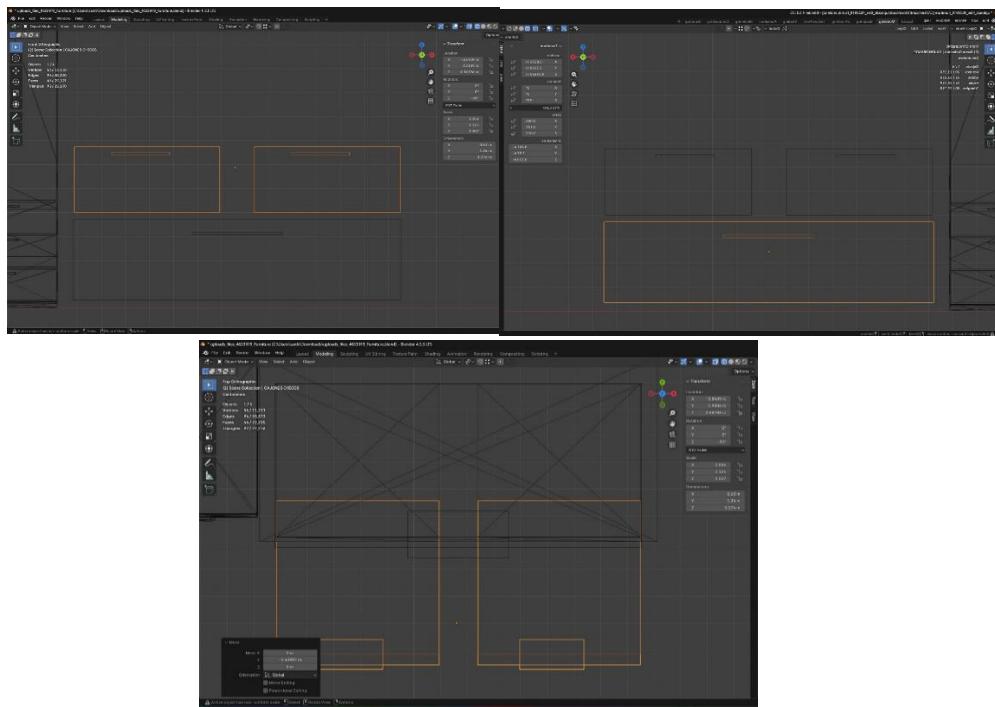


Imágenes finales de la casa.

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

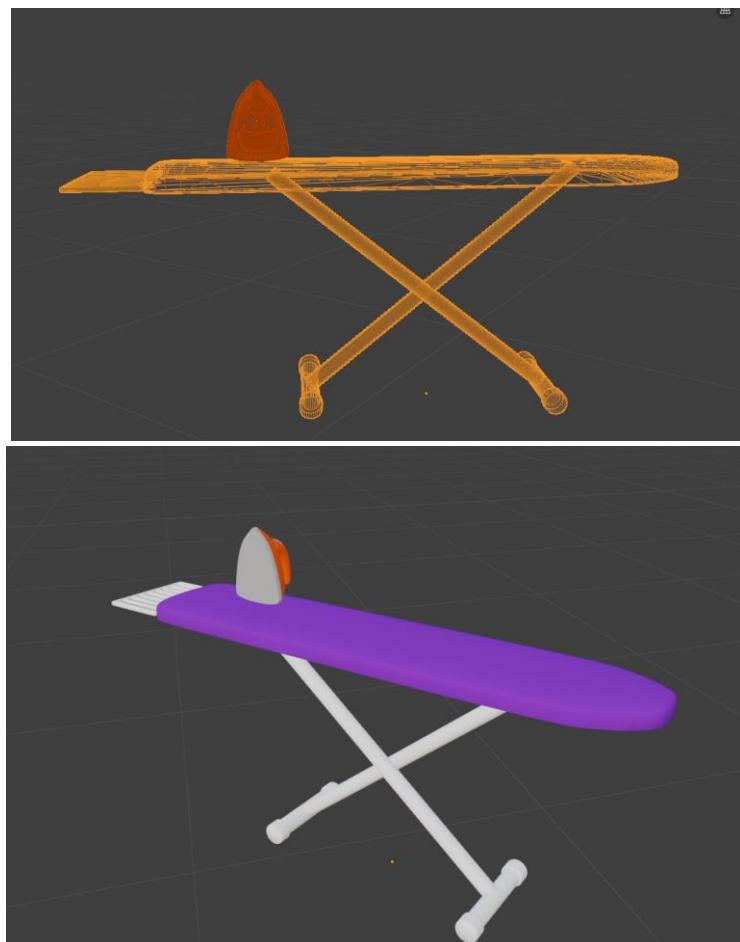


Imagen. Burro de planchar y plancha de ropa.

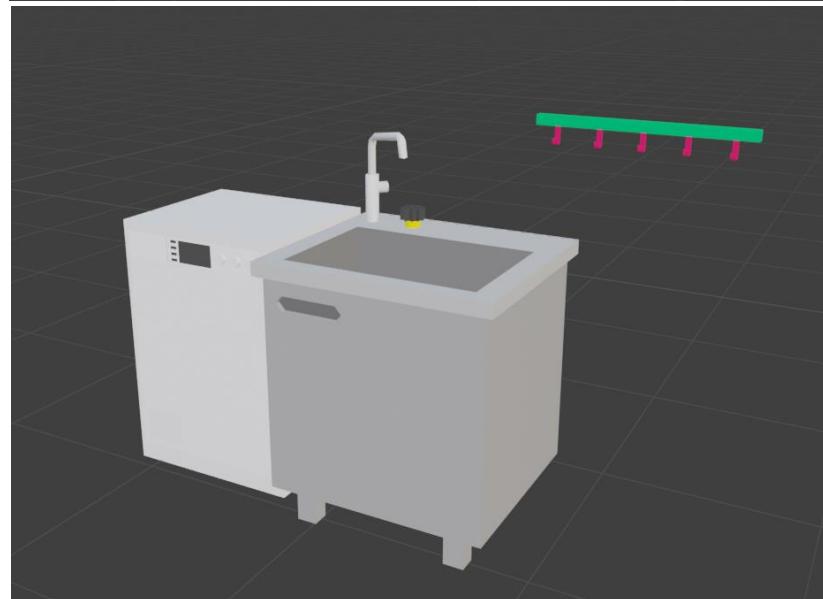
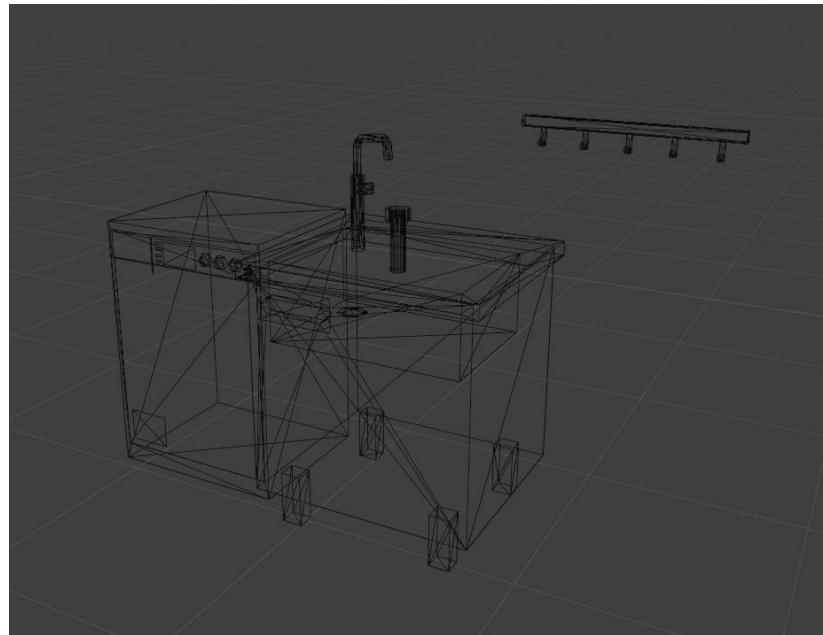


Imagen. Lavadora sin tambor, lavabo del cuarto de lavado y perchero de pared.

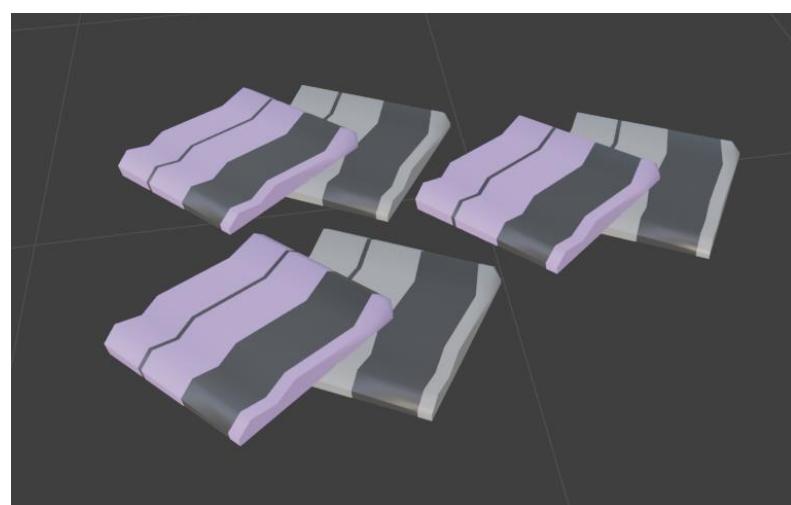


Imagen. Ropa tirada en el cuarto de lavado.

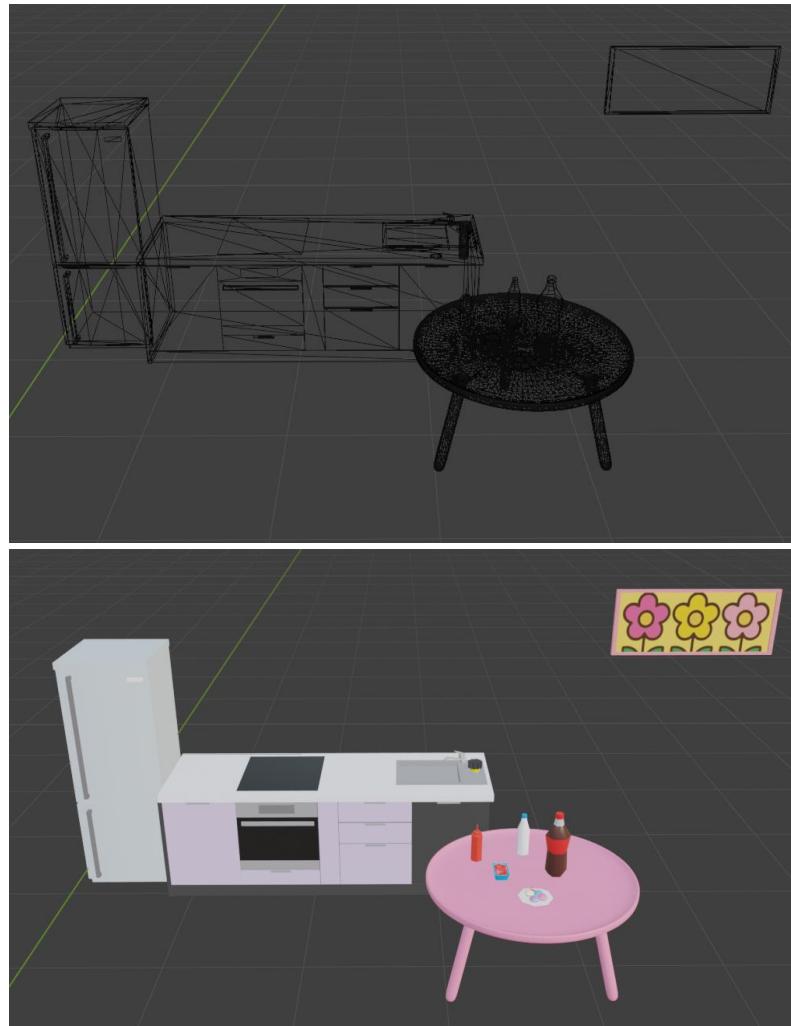
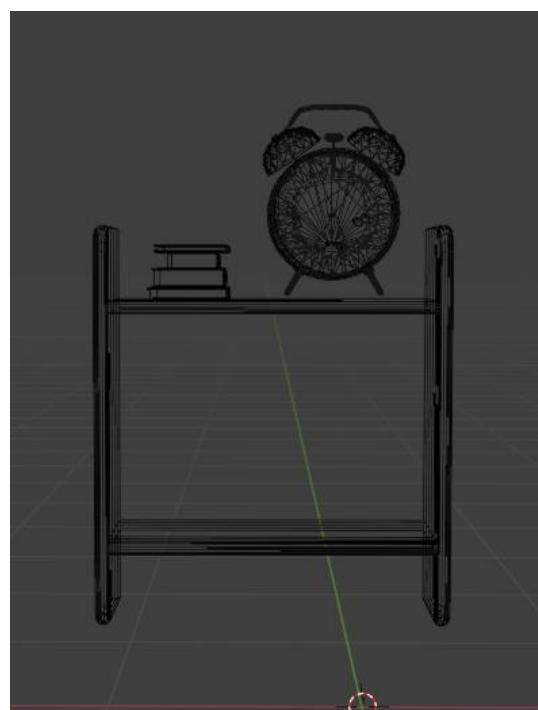


Imagen. Refrigerador, estufa, lavabo, mesa con comida y cuadro de una pintura.



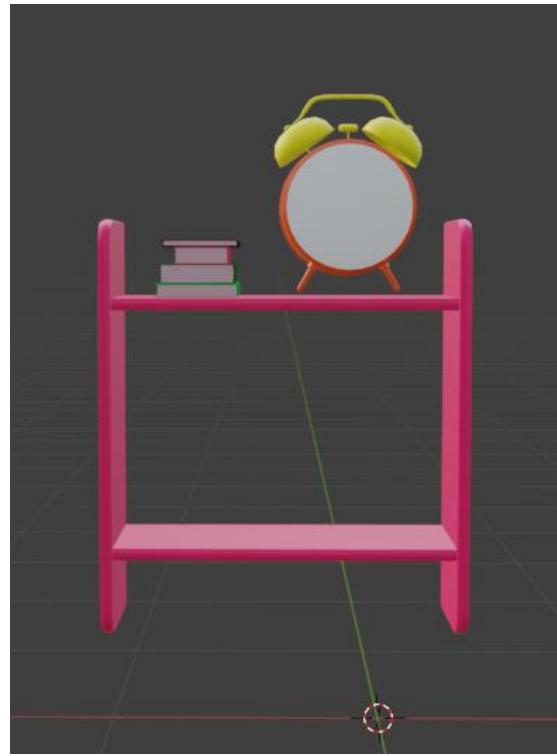


Imagen. Repisa, libros y reloj.

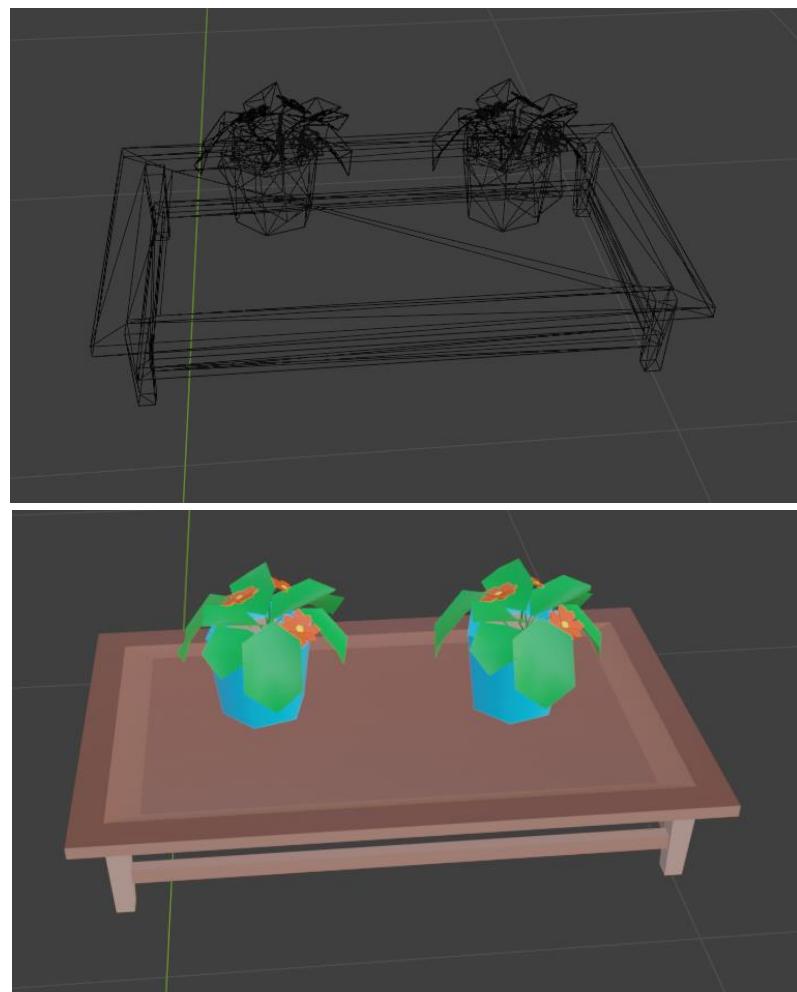


Imagen. Flores sobre mesa.

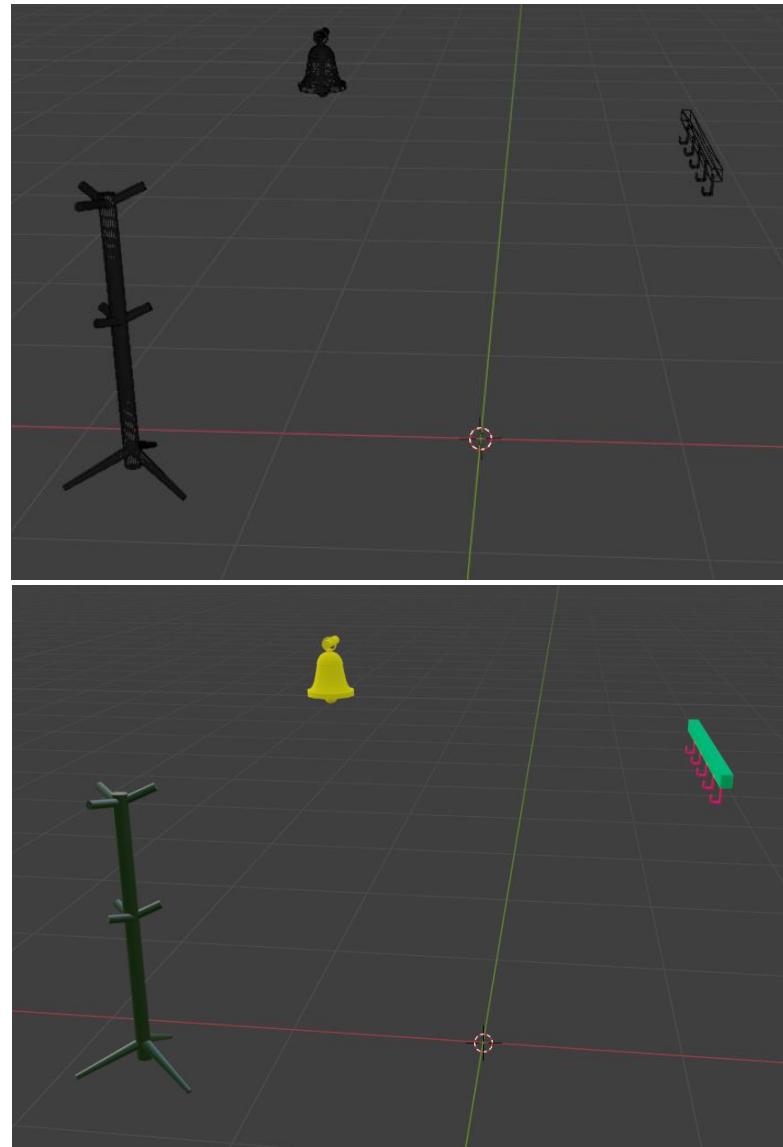
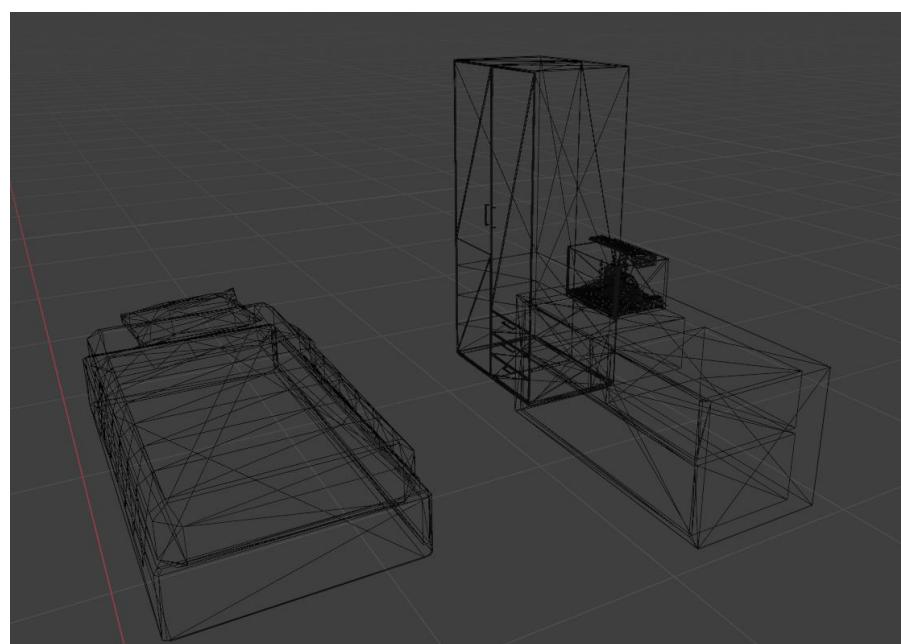


Imagen. Pechero, perchero de pared y campana.



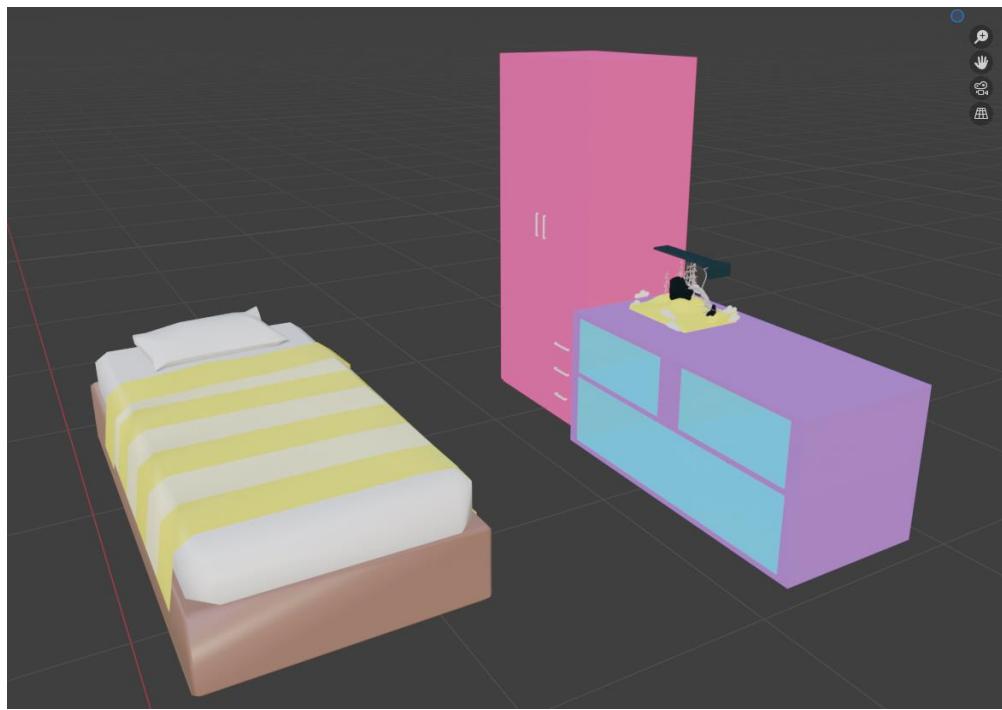


Imagen. Cama, ropero alto, ropero chico y pecera.

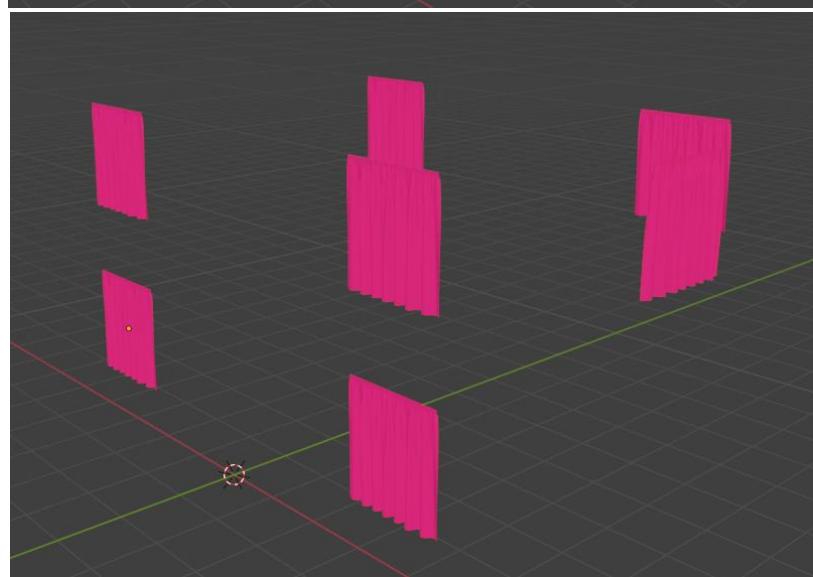
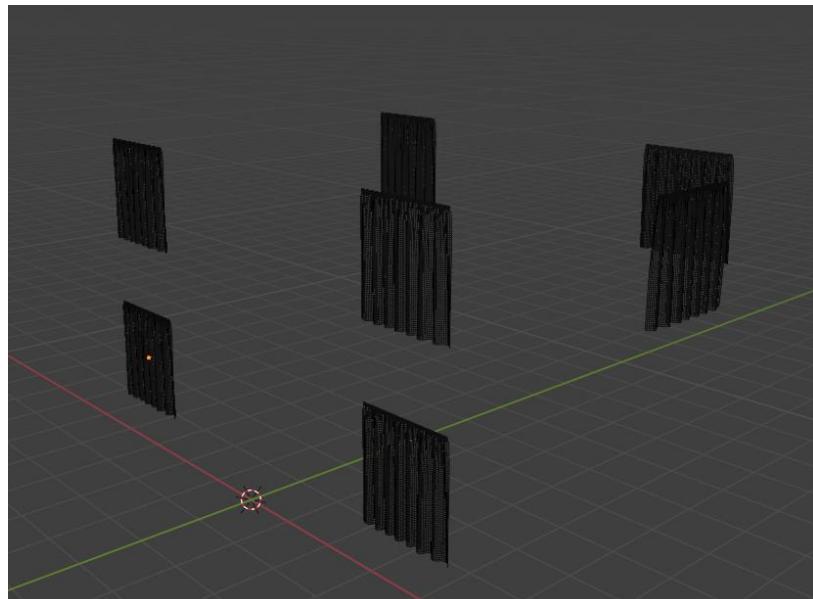


Imagen. Cortinas.

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.

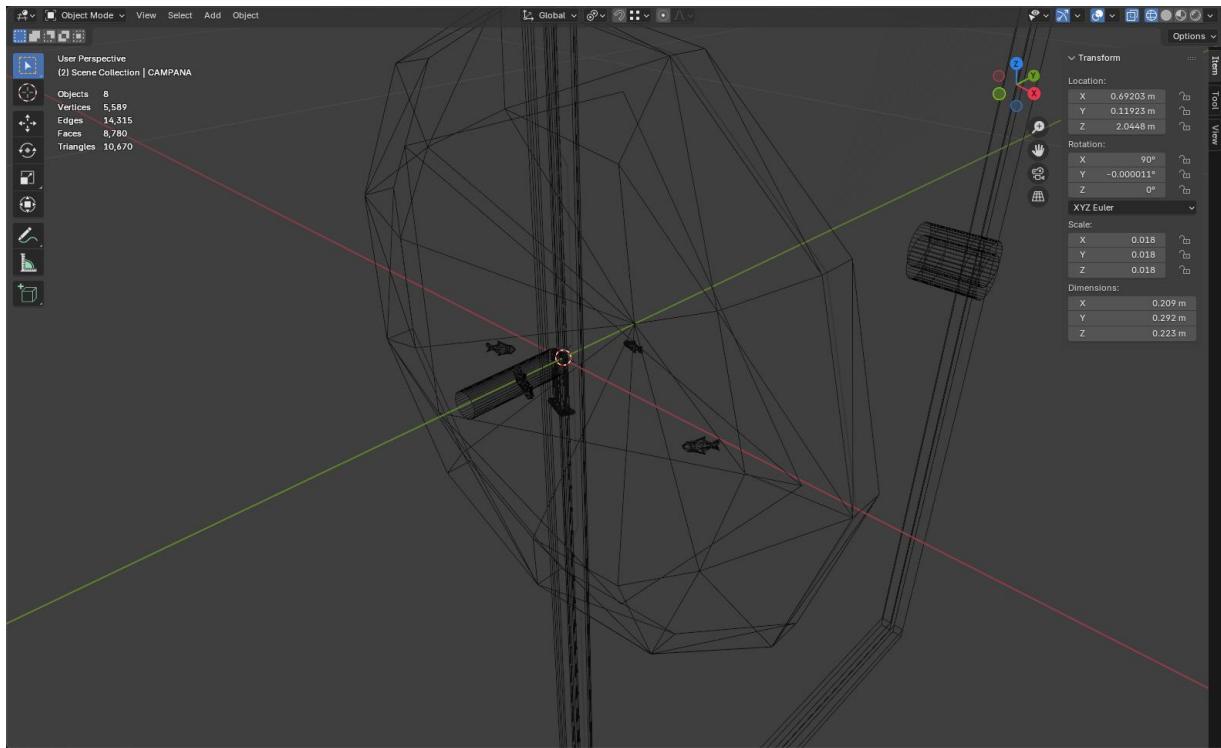


Imagen. Tambor de la lavadora, peces de la pecera, manecilla del reloj

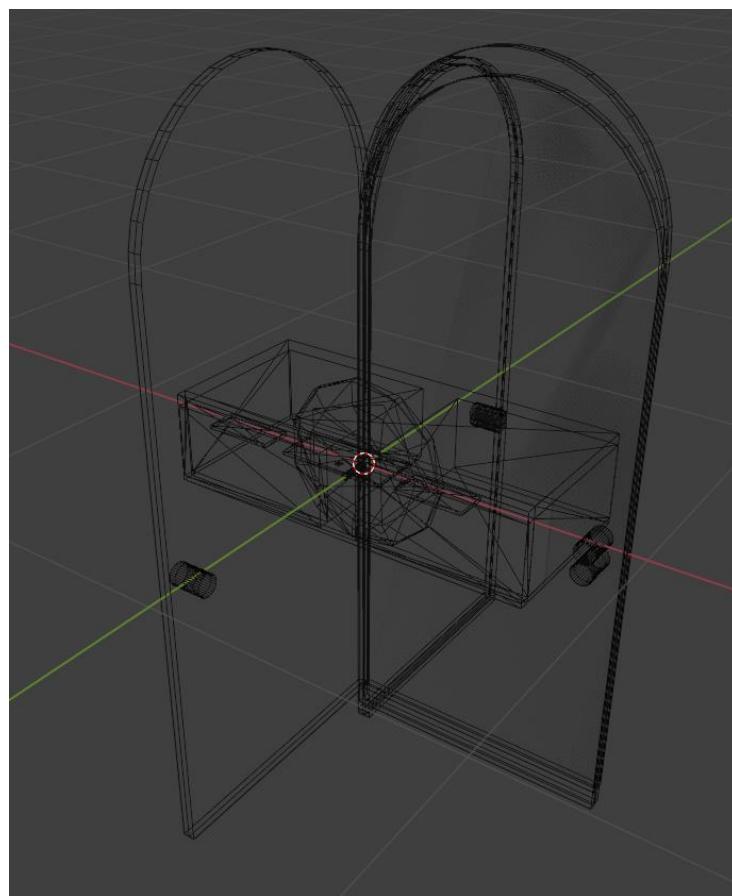


Imagen. Puertas de la casa y cajones.

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:

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.

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.

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.

Preparación para OpenGL:

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

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 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"), 0.6f, 0.6f, 0.65f);
    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, 0.16f, 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 Agua.

Dentro de la función *main()* no solo se configuran los modelos y shaders, sino también las animaciones y efectos dinámicos, entre ellos el sistema de partículas que simula el agua del lavabo. Para ello se crea un arreglo de 150 partículas, cada una con atributos necesarios para su comportamiento físico: posición en 3D, velocidad, tiempo de vida, tamaño, estado activo y un indicador que distingue entre gotas principales y salpicaduras. Durante la inicialización, todas se establecen como inactivas, con vida igual a cero y un tamaño base de 0.28 unidades, dejándolas preparadas para entrar en acción cuando el usuario active la animación.

```

// Configuración del sistema de partículas DE AGUA
const int NUM_GOTAS = 150; // Aumentado para incluir salpicaduras
struct Particula {
    glm::vec3 posicion;
    glm::vec3 velocidad;
    float vida;
    float tamano; // Tamaño de la gota
    bool activa;
    bool esSalpicadura; // Diferenciar gotas principales de salpicaduras
};

for (int i = 0; i < NUM_GOTAS; i++)
{
    if (particulas[i].activa)
    {
        model = glm::mat4(1.0f);
        model = glm::translate(model, particulas[i].posicion);

        model[0] = glm::vec4(cameraRight, 0.0f);
        model[1] = glm::vec4(cameraUp, 0.0f);
        model[2] = glm::vec4(glm::cross(cameraRight, cameraUp), 0.0f);

        float anchoGota = particulas[i].tamano;
        float altoGota = particulas[i].tamano * 1.5f;

        if (particulas[i].esSalpicadura)
        {
            anchoGota *= 0.6f;
            altoGota *= 0.6f;
        }

        model = glm::scale(model, glm::vec3(anchoGota, altoGota, 1.0f));
    }
}

// Configuración de físicas DEL AGUA
const float GRAVEDAD = 9.8f;
const float TIEMPO_VIDA_GOTA = 3.0f; // Más tiempo de vida
const float TIEMPO_VIDA_SALPICADURA = 0.8f; // Salpicaduras viven menos
const float TASA_GENERACION = 0.015f; // Más gotas por segundo

```

Dentro de esta animación también tenemos este bloque que inicializa y configura la textura utilizada para representar cada partícula de agua dentro del sistema de billboards. Primero se genera un identificador de textura y se enlaza al tipo GL_TEXTURE_2D, tras lo cual se establecen sus parámetros: *GL_CLAMP_TO_EDGE* para evitar artefactos en los bordes y filtrado lineal con mipmaps para asegurar una apariencia suave a diferentes distancias.

Luego, mediante **stb_image**, se carga el archivo *GOTA.png*, aplicando un volteo vertical para corregir la inversión habitual entre coordenadas de imagen y las de OpenGL. El código detecta automáticamente el número de canales (RGB o RGBA) para asignar el formato adecuado antes de enviar los datos a la GPU mediante *glTexImage2D*. Finalmente, se generan los mipmaps y se libera la memoria de la imagen cargada.

```

// ====== CARGAR TEXTURA DE GOTA ======
GLuint textureGota;
 glGenTextures(1, &textureGota);
 glBindTexture(GL_TEXTURE_2D, textureGota);

 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);

 int texWidth, texHeight, nrChannels;
 stbi_set_flip_vertically_on_load(true);
 unsigned char* data = stbi_load("Images/GOTA.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 gota cargada: " << texWidth << "x" << texHeight << " (" << nrChannels << " canales)" << std::endl;
 }
 else
 {
    std::cout << "Error al cargar textura: Images/GOTA.png" << std::endl;
 }
 stbi_image_free(data);

```

Animación del Vapor:

La animación del vapor utiliza prácticamente la misma estructura que el sistema de agua. Primero se inicializa un arreglo de partículas de vapor, dejando todas en estado inactivo y con tiempo de vida igual a cero, listas para activarse cuando se presione la tecla correspondiente.

Finalmente, tanto el agua como el vapor comparten la misma geometría de billboard, que es simplemente un cuadrado plano que siempre mira hacia la cámara. Este cuadrado funciona como un “portador” de la textura (*GOTA.png* o *VAPOR.png*), permitiendo simular efectos visuales sin necesidad de geometrías complejas.

La geometría del billboard se define con cuatro vértices distribuidos en un cuadrado de 0.1 unidades de lado, centrado en el origen. Cada vértice incluye sus coordenadas 2D y sus coordenadas de textura:

```
// ===== 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 y cajones

- Las cortinas se mueven en el eje X usando variables posCortinaDerecha y posCortinaIzquierda, mientras que los cajones se animan trasladándolos en el eje Z.

```

// ===== 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;
        }
    }
}

```

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);

```

Animación de los peces (movimiento circular y ondulante)

En esta parte del código se explica el movimiento de los peces dentro de la pecera tienen una animación automática que combina un movimiento circular y un movimiento ondulado en vertical, lo que hace que parezcan nadar de forma natural alrededor del centro de la pecera.

Primero se define la posición central de la pecera y un radio pequeño para que los peces giren cerca de ese punto:

```

// Posición central de la pecera
glm::vec3 centroPecera = glm::vec3(-4.2546f, 0.81017f, -2.6099f);

float radioCarrusel = 0.08f;

```

A partir del ángulo anguloCarrusel (que se va incrementando cada frame en la lógica de animación), se calcula una trayectoria circular en el plano XZ usando funciones trigonométricas:

```

// Calcular posición en trayectoria circular (carrusel en XZ)
float offsetX = radioCarrusel * cos(anguloCarrusel);
float offsetZ = radioCarrusel * sin(anguloCarrusel);

```

Luego se añade un movimiento senoidal en Y para simular que los peces suben y bajan suavemente:

```

// Movimiento senoidal en Y con amplitud reducida
float amplitudSeno = 0.06f;
float frecuenciaSeno = 2.0f;
float offsetY = amplitudSeno * sin(frecuenciaSeno * tiempoPeces);

// Aplicar transformaciones
model = glm::translate(model, centroPecera);
model = glm::translate(model, glm::vec3(offsetX, offsetY, offsetZ));
model = glm::rotate(model, anguloCarrusel, glm::vec3(0.0f, 1.0f, 0.0f));

glUniformMatrix4fv(glGetUniformLocation(lightingShader.Program, "model"), 1, GL_FALSE, glm::value_ptr(model));
peces.Draw(lightingShader);

```

Finalmente, se construye la matriz model aplicando las transformaciones en este orden:

1. Trasladar al centro de la pecera.
2. Aplicar el offset en X, Y y Z (trayectoria + ondulación).
3. Rotar el modelo según anguloCarrusel para que los peces miren en la dirección de movimiento.

De esta forma, los peces no solo giran alrededor de la pecera, sino que también realizan un movimiento vertical, dando la impresión de nado continuo.

Geometría nativa en OpenGL: silla, sillón, reloj y apagador.

Además de los modelos importados desde Blender, el proyecto incluye varios objetos creados directamente desde código usando cubos como geometría base. En este bloque se dibujan:

- La silla amarilla
- El sillón multicolor
- El reloj de pared
- El apagador / interruptor

Todo se renderiza con el mismo VAO de cubo (VAO_silla) y el lightingShader y aunque cada uno es diferente, todos se construyen siguiendo los mismos pasos.

1. Activar el shader e indicar matrices de proyección y vista

Todos los objetos usan el lightingShader, así que se inicia así:

```
lightingShader.Use();
glUniformMatrix4fv(... "projection" ...);
glUniformMatrix4fv(... "view" ...);
```

2. Configurar el material (color, brillo y especularidad)

Cada objeto ajusta su color y brillo antes de dibujarse:

```
glUniform3f(... "material.ambient", r, g, b);
glUniform3f(... "material.diffuse", r, g, b);
glUniform3f(... "material.specular", ... );
glUniform1f(... "material.shininess", ... );
```

3. Usar la misma geometría base (un cubo)

Todos los objetos OpenGL nativos usan el mismo VAO y el mismo VBO:

```
glBindVertexArray(VAO_silla);
```

4. Crear la forma de cada parte usando transformaciones

Cada pieza del objeto es simplemente un cubo al que se aplican:

translate() → para colocarlo

rotate() → para orientarlo

scale() → para darle la forma final

```
// 4 patas
for (int i = 0; i < 4; i++) {
    float xPos = (i % 2 == 0) ? xPSillon : -xPSillon;
    float zPos = (i < 2) ? zPSillon : -zPSillon;

    model = glm::mat4(1.0f);
    model = glm::translate(model, posSillon);
    model = glm::rotate(model, glm::radians(rotacionSillonY), glm::vec3(0.0f, 1.0f, 0.0f));
    model = glm::translate(model, glm::vec3(xPos, yPataSillon, zPos));
    model = glm::scale(model, escPataSillon);
    glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
    glDrawArrays(GL_TRIANGLES, 0, 36);
}
```

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), animaciones de puertas y muebles, y los sistemas de partículas de agua y vapor.

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 y el movimiento de cajones se adapta automáticamente al rendimiento de la máquina.

Después se actualizan las animaciones continuas:

- El tambor de la lavadora (rotTambor) y la manecilla del reloj (rotManecilla) aumentan su ángulo de rotación mientras sus banderas (animTambor, animManecilla) estén activas.
- En ambos casos, 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, cortinas y cajones. 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)
- Cajón grande y cajón chico con sus posiciones en Z y sus banderas (animCajonGrande, animCajonChico)

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

```
// ===== ANIMACIÓN CAJÓN GRANDE (TECLA J) ======
if (animCajonGrande)
{
    if (abrirCajonGrande)
    {
        posCajonGrandeZ += velocidadCajon;
        if (posCajonGrandeZ >= POS_FINAL_CAJON_GRANDE_Z)
        {
            posCajonGrandeZ = POS_FINAL_CAJON_GRANDE_Z;
            animCajonGrande = false;
        }
    }
    else
    {
        posCajonGrandeZ -= velocidadCajon;
        if (posCajonGrandeZ <= POS_INICIAL_CAJON_GRANDE_Z)
        {
            posCajonGrandeZ = POS_INICIAL_CAJON_GRANDE_Z;
            animCajonGrande = 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

```
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);
    }
}
```

actualizando en KeyCallback.

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 el tambor de la lavadora y la manecilla del reloj usan un toggle simple, mientras que Las puertas, cortinas y cajones usan un toggle inteligente, que decide si se abren o se cierran según su posición actual.

Las teclas G, 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 = !agujaEncendida;
    std::cout << "Agua " << (agujaEncendida ? "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.

Generación de salpicaduras

Finalmente tenemos la función CrearSalpicaduras() que se llama cuando una gota de agua impacta con el lavabo. Su objetivo es crear varias partículas secundarias que simulan el salpique del agua.

```
void CrearSalpicaduras(glm::vec3 posicionImpacto)
{
    int numSalpicaduras = 5 + (rand() % 4);

    for (int j = 0; j < numSalpicaduras; j++)
    {
        for (int i = 0; i < NUM_GOTAS; i++)
        {
            if (!particulas[i].activa)
            {
                particulas[i].activa = true;
                particulas[i].esSalpicadura = true;

                particulas[i].posicion = posicionImpacto;
                particulas[i].posicion.y = alturaLavabo;
            }
        }
    }
}
```

Primero se decide cuántas salpicaduras se van a generar. Luego, para cada una, se busca una partícula libre en el arreglo:

- Se marca como activa y como salpicadura (esSalpicadura = true).
- Se coloca en la posición de impacto, a la altura del lavabo.

Después se calcula una dirección aleatoria en 360° y una distancia:

```
float angulo = (rand() % 360) * 3.14159f / 180.0f;
float distancia = 0.15f + (rand() % 45) / 100.0f;

particulas[i].velocidad = glm::vec3(
    cos(angulo) * distancia * 5.0f,
    0.8f + (rand() % 20) / 100.0f,
    sin(angulo) * distancia * 5.0f
);
```

- $\cos(\text{angulo})$ y $\sin(\text{angulo})$ se usan para dispersar las gotas en círculo.
- La componente Y es positiva para que las gotas “salten” hacia arriba.
- *tamano* y *vida* se ajustan para que estas partículas sean más pequeñas y vivan menos tiempo que las gotas principales:

```
particulas[i].tamano = 0.20f + (rand() % 10) / 100.0f;
particulas[i].vida = TIEMPO_VIDA_SALPICADURA;
break;
```

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 <fstream>
6  #include <sstream>
7  #include <iostream>
8
9  #include <GL/glew.h>
10
11 class Shader
12 {
13 public:
14     GLuint Program;
15     GLuint uniformColor;
16     // Constructor generates the shader on the fly
17     Shader(const GLchar *vertexPath, const GLchar *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         // ensures ifstream objects can throw exceptions:
25         vShaderFile.exceptions(std::ifstream::badbit);
26         fShaderFile.exceptions(std::ifstream::badbit);
27         try
28         {
29             // Open files
30             vShaderFile.open(vertexPath);
31             fShaderFile.open(fragmentPath);
32             std::stringstream vShaderStream, fShaderStream;
33             // Read file's buffer contents into streams
34             vShaderStream << vShaderFile.rdbuf();
35             fShaderStream << fShaderFile.rdbuf();
36             // close file handlers
37             vShaderFile.close();
38             fShaderFile.close();
39             // Convert stream into string
40             vertexCode = vShaderStream.str();
41             fragmentCode = fShaderStream.str();
42         }
43         catch (std::ifstream::failure e)
44         {
45             std::cout << "ERROR::SHADER::FILE_NOT_SUCCESFULLY_READ" << std::endl;
46         }
47         const GLchar *vShaderCode = vertexCode.c_str();
48         const GLchar *fShaderCode = fragmentCode.c_str();
49         // 2. Compile shaders
50         GLuint vertex, fragment;
51         GLint success;
52         GLchar infoLog[512];
53         // Vertex Shader
54         vertex = glCreateShader(GL_VERTEX_SHADER);
55         glShaderSource(vertex, 1, &vShaderCode, NULL);
56         glCompileShader(vertex);
57         // Print compile errors if any
58         glGetShaderiv(vertex, GL_COMPILE_STATUS, &success);
59         if (!success)
60         {
61             glGetShaderInfoLog(vertex, 512, NULL, infoLog);
62             std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog << std::endl;
63         }
64         // Fragment Shader
65         fragment = glCreateShader(GL_FRAGMENT_SHADER);
66         glShaderSource(fragment, 1, &fShaderCode, NULL);
67         glCompileShader(fragment);
68         // Print compile errors if any
69         glGetShaderiv(fragment, GL_COMPILE_STATUS, &success);
70         if (!success)
71         {
72             glGetShaderInfoLog(fragment, 512, NULL, infoLog);
73             std::cout << "ERROR::SHADER::FRAGMENT::COMPILATION_FAILED\n" << infoLog << std::endl;
74         }
75         // Shader Program
76         this->Program = glCreateProgram();
77         glAttachShader(this->Program, vertex);
78         glAttachShader(this->Program, fragment);
79         glLinkProgram(this->Program);
80         // Print linking errors if any
81         glGetProgramiv(this->Program, GL_LINK_STATUS, &success);
82         if (!success)
83         {
84             glGetProgramInfoLog(this->Program, 512, NULL, infoLog);
85             std::cout << "ERROR::SHADER::PROGRAM::LINKING_FAILED\n" << infoLog << std::endl;
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 necessary
90         glDeleteShader(vertex);
91         glDeleteShader(fragment);
92
93         // Uses the current shader
94         void Use()
95         {
96             glUseProgram(this->Program);
97         }
98
99         GLuint getColorLocation()
100        {
101            return uniformColor;
102        }
103    };
104
105 #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.

```

1  #pragma once
2
3  // Std. Includes
4  #include <vector>
5
6  // GL Includes
7  #define GLEW_STATIC
8  #include <GL/glew.h>
9
10 #include <glm/glm.hpp>
11 #include <glm/gtc/matrix_transform.hpp>
12
13 // Defines several possible options for camera movement. Used as abstraction to stay away from window-system specific input methods
14 enum Camera_Movement
15 {
16     FORWARD,
17     BACKWARD,
18     LEFT,
19     RIGHT
20 };
21
22 // Default camera values
23 const GLfloat PITCH = -90.0f;
24 const GLfloat YAW = 0.0f;
25 const GLfloat SPEED = 6.0f;
26 const GLfloat SENSITIVITY = 0.25f;
27 const GLfloat ZOOM = 45.0f;
28
29 // An abstract camera class that processes input and calculates the corresponding Eular Angles, Vectors and Matrices for use in OpenGL
30 class Camera
31 {
32 public:
33     // Constructor with vectors
34     Camera(glm::vec3 position = glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3 up = glm::vec3(0.0f, 1.0f, 0.0f), GLfloat yaw = YAW, GLfloat pitch = PITCH) : front(glm::vec3(0.0f, 0.0f, -1.0f)), movementSpeed(SPEED), mouseSensitivity(SENSITIVITY), zoom(ZOOM)
35     {
36         this->position = position;
37         this->worldUp = up;
38         this->yaw = yaw;
39         this->pitch = pitch;
40         this->updateCameraVectors();
41     }
42
43     // Constructor with scalar values
44     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)
45     {
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->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         // Processes input received from a mouse input system. Expects the offset value in both the x and y direction.
85         void ProcessMouseMovement(GLfloat xOffset, GLfloat yOffset, GLboolean constrainPitch = true)
86         {
87             xOffset *= this->mouseSensitivity;
88             yOffset *= this->mouseSensitivity;
89
90             this->yaw += xOffset;
91             this->pitch += yOffset;
92
93             // Make sure that when pitch is out of bounds, screen doesn't get flipped
94             if (constrainPitch)
95             {
96                 if (this->pitch > 89.0f)
97                 {
98                     this->pitch = 89.0f;
99                 }
100
101                 if (this->pitch < -89.0f)
102                 {
103                     this->pitch = -89.0f;
104                 }
105             }
106
107             // Update Front, Right and Up Vectors using the updated Eular angles
108             this->updateCameraVectors();
109         }
110
111         // Processes input received from a mouse scroll-wheel event. Only requires input on the vertical wheel-axis
112         void ProcessMouseScroll(GLfloat yOffset)
113         {
114
115             GLfloat GetZoom()
116             {
117                 return this->zoom;
118             }
119
120             glm::vec3 GetPosition()
121             {
122                 return this->position;
123             }
124
125             glm::vec3 GetFront()
126             {
127                 return this->front;
128             }
129
130             private:
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     }

```

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      uniform mat4 model;
6      uniform mat4 view;
7      uniform mat4 projection;
8
9      void main()
10     {
11         gl_Position = projection * view * model * vec4(position, 1.0f)
12     }
13
14

```

CONCLUSIONES

General

El desarrollo de este proyecto nos permitió integrar de manera práctica los conocimientos adquiridos en modelado 3D y programación gráfica, dando vida a un entorno interactivo basado en la casa de Hello Kitty. A lo largo del proceso trabajamos desde la creación de modelos en Blender — incluyendo muebles, accesorios y elementos decorativos— hasta su exportación, texturizado e implementación dentro del entorno programado con OpenGL. Esto nos brindó una comprensión completa del flujo de trabajo, desde el diseño visual hasta su ejecución técnica.

Dividimos las actividades según nuestras fortalezas: uno de nosotros se centró principalmente en el modelado y texturizado, mientras que el otro se encargó de la integración, las animaciones y la programación del entorno. Esta organización nos permitió avanzar con mayor fluidez, complementar habilidades y resolver desafíos mediante trabajo colaborativo.

Herramientas como GLFW, GLEW y Assimp fueron fundamentales para estructurar el funcionamiento del proyecto, y el uso de shaders personalizados nos permitió implementar iluminación direccional, puntual y tipo spotlight, aportando realismo a la escena. También añadimos una cámara en tercera persona y luces dinámicas para mejorar la interacción y el aspecto visual del espacio.

En conjunto, este proyecto no solo reforzó nuestros conocimientos técnicos, sino que también nos permitió combinar creatividad, lógica y colaboración para construir una representación interactiva y coherente de la casa de Hello Kitty.

González Frías Ana Paula

Este proyecto fue una experiencia muy enriquecedora que me permitió aplicar de forma práctica conocimientos sobre computación gráfica, modelado 3D en Blender y programación con OpenGL. El proceso de diseñar objetos, exportarlos e integrarlos dentro de un entorno interactivo —en este caso, la casa de Hello Kitty— me ayudó a comprender de manera más profunda el flujo completo de desarrollo en aplicaciones gráficas.

Además, el trabajo colaborativo fue fundamental para mejorar mi comunicación, coordinación y entendimiento del proceso creativo y técnico. Esta experiencia me permitió unir la parte visual con la lógica de programación, logrando una escena funcional, atractiva y coherente.

Orihuela Arellano Santiago

Durante el desarrollo de este proyecto pude fortalecer mis habilidades tanto en el modelado 3D como en la comprensión de los fundamentos de programación gráfica. Mi participación se enfocó principalmente en la creación y texturizado de modelos dentro de Blender, donde aproveché herramientas como extrusión, escalado y técnicas de organización de mallas para construir de forma detallada varios elementos que conforman la casa de Hello Kitty. Fue especialmente gratificante observar cómo objetos creados desde formas básicas podían integrarse como elementos funcionales y estéticamente coherentes en la escena.

Al incorporarlos al entorno gráfico con OpenGL, comprendí con mayor claridad el funcionamiento de los shaders, las transformaciones 3D, la iluminación y el manejo de materiales. Además, participar en la implementación de la cámara y los efectos visuales me permitió ver la importancia del equilibrio entre el diseño visual y la lógica de programación.

Este proyecto no solo reforzó mis habilidades técnicas, sino que también me enseñó la relevancia de la organización por etapas y del trabajo colaborativo para lograr un resultado final integrador y funcional.

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>