



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

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

**PROFESOR: ING. CARLOS ALDAIR ROMAN BALBUENA
SEMESTRE 2025-2**

**PROYECTO FINAL:
MANUAL TÉCNICO**

**ALUMNO:
LIMÓN SOSA ZAIR ODIN
GRUPO: 05**

FECHA DE ENTREGA: 20/05/25

INDICE

MANUAL TÉCNICO - PROYECTO FINAL	3
DESCRIPCIÓN DEL PROYECTO	3
OBJETIVOS.....	3
DIAGRAMA DE FLUJO DEL SOFTWARE	3
DIAGRAMA DE GANTT.....	4
ALCANCE DEL PROYECTO.....	5
LIMITANTES	5
ANALISIS DE COSTOS DEL PROYECTO	6
METODOLOGÍA DE SOFTWARE APLICADA.....	7
DOCUMENTACION DE LA ANIMACIONES	21
DOCUMENTACIÓN DEL CÓDIGO.....	27
CONCLUSIÓN	47
REFERENCIAS.....	48

MANUAL TÉCNICO - PROYECTO FINAL

DESCRIPCIÓN DEL PROYECTO

Este proyecto consiste en la recreación tridimensional de una fachada ficticia inspirada en la casa de Charlie Brown, del universo Peanuts. Fue desarrollado en OpenGL con C++ y emplea modelos creados en Autodesk Maya, los cuales fueron exportados e integrados usando la librería Assimp. La escena contiene tres habitaciones distintas, cada una con al menos cinco objetos modelados, texturizados y algunos animados, así como interacciones que le aportan dinamismo y coherencia al entorno.

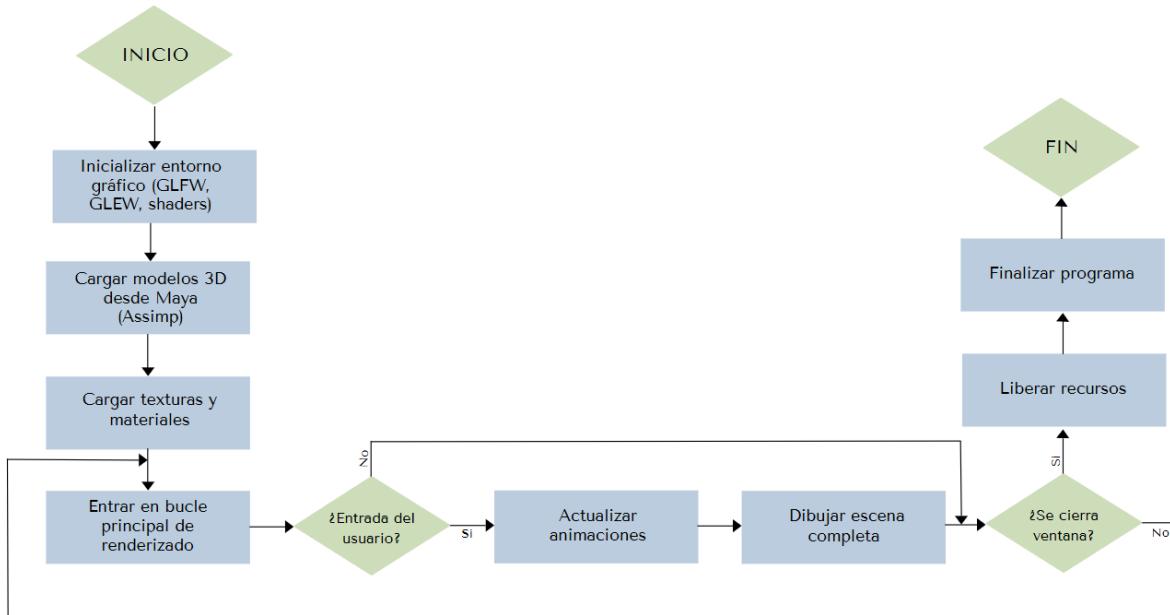
OBJETIVOS

Los objetivos de este proyecto serán:

- Modelar e implementar una escena 3D completa y funcional basada en una referencia ficticia.
- Aplicar herramientas y técnicas de modelado 3D profesional con Autodesk Maya.
- Programar animaciones e interacciones usando OpenGL y C++.
- Documentar adecuadamente el desarrollo y uso del software.

DIAGRAMA DE FLUJO DEL SOFTWARE

DIAGRAMA DE FLUJO DEL SOFTWARE INTERACTIVO – PROYECTO PEANUTS



El diagrama representa el flujo de ejecución de un software interactivo 3D, desde su inicio hasta el cierre. Comienza con la inicialización del entorno gráfico, utilizando herramientas como GLFW, GLEW y shaders, necesarias para crear una ventana de renderizado y controlar gráficos en tiempo real.

Posteriormente, se cargan los modelos 3D previamente creados en Maya, a través de la librería Assimp, seguida por la carga de texturas y materiales que definen la apariencia visual de dichos modelos.

Una vez completada esta fase de preparación, el programa entra en el bucle principal de renderizado, que se ejecuta continuamente mientras la ventana esté activa. Dentro de este bucle, el sistema verifica si hay entrada del usuario (por ejemplo, teclas, clics o movimientos). Si la hay, se actualizan las animaciones o elementos interactivos en pantalla.

Después de esto, se dibuja la escena completa, es decir, se genera visualmente el entorno con sus modelos, iluminación y animaciones. Al final de cada ciclo, el programa revisa si el usuario ha cerrado la ventana. Si no se ha cerrado, el bucle se repite. Si se cierra, el software libera los recursos utilizados (memoria, texturas, buffers) y finaliza el programa correctamente.

DIAGRAMA DE GANTT

TAREAS	SEMANA 1	SEMANA 2	SEMANA 3	SEMANA 4	SEMANA 5	SEMANA 6	SEMANA 7	SEMANA 8	SEMANA 9-11
	11 - 16 de marzo	17 - 23 de marzo	24 - 30 de marzo	1 - 6 de abril	7 - 13 de abril	14 - 20 de abril	21 - 27 de abril	28 abril - 4 de mayo	5 - 20 de mayo
Planificación general									
Bocetos de habitaciones									
Selección de herramientas									
Modelado 3D básico en Maya de objetos y habitaciones									
Aplicar texturas y materiales en Maya									
Exportar modelos (.obj)									
Integrar modelos en OpenGL con Assimp									
Programar animaciones									
Ajustar cámara y navegación									
Optimización gráfica									
Pruebas finales y correcciones									
Iniciar documentación									
Finalizar manual técnico									
Subida del proyecto completo a GitHub									

Este diagrama de Gantt representa la planificación de un proyecto técnico dividido en tareas específicas distribuidas a lo largo de 11 semanas (del 11 de marzo al 20 de mayo). A continuación se describe la distribución de tareas por semana:

Fase inicial de planeación (Semanas 1-3)

- Planificación general: Semana 1 (11 - 16 marzo)
- Bocetos de habitaciones: Semana 1
- Selección de herramientas: Semana 1
- Modelado 3D básico en Maya: Semanas 2 a 4 (17 marzo - 6 abril)

Desarrollo en Maya (Semanas 2-5)

- Aplicar texturas y materiales: Semanas 2 a 5
- Exportar modelos (.obj): Semanas 4 a 5

Integración en OpenGL (Semanas 4-6)

- Integrar modelos con Assimp: Semanas 4 a 6

Programación y lógica (Semanas 5-9)

- Programar animaciones: Semanas 5 a 8
- Ajustar cámara y navegación: Semanas 5 a 8
- Optimización gráfica: Semanas 7 a 9

Pruebas y documentación (Semanas 8-11)

- Pruebas finales y correcciones: Semanas 8 a 9
- Iniciar documentación: Semanas 9 a 10
- Finalizar manual técnico: Semanas 10 a 11
- Subida del proyecto a GitHub: Semana 11 (20 de mayo)

ALCANCE DEL PROYECTO

- Tres habitaciones con ambientaciones distintas.
- Al menos 15 objetos modelados en Maya (15 totales porque se trabajó en equipo).
- Animaciones implementadas (Realizadas por Limón Sosa Zair Odin):
 - ◆ Recorrido de Snoopy
 - ◆ Abrir y cerrar puertas del cuarto principal y del cuarto de Odin
 - ◆ Abrir y cerrar ventanas
 - ◆ Abrir y cerrar puertas del ropero
 - ◆ Tirar botella
 - ◆ Girar copa
 - ◆ Manecillas del reloj
 - ◆ Cámara del entorno
- Exportación de modelos en formato compatible (.obj).
- Renderizado e interacción en tiempo real usando OpenGL.

LIMITANTES

- Recursos visuales limitados a fines académicos.
- No había conocimiento previo para utilizar Git y GitHub.
- Peso restringido de los modelos (< 100 MB cada uno).
- No se permite recrear escenarios reales (UNAM, empresas).
- El desarrollo se realizó en plataformas compatibles con Windows.
- El proyecto puede no correr de forma fluida en computadoras con GPU integrada o sin soporte completo de OpenGL 3.3 o superior.

- La animación de Snoopy es activada por teclas; no hay detección de colisiones.
- No se utilizaron simulaciones físicas avanzadas.
- El proyecto no incluye efectos de sonido o música de fondo.
- El programa fue desarrollado y probado en Windows. Su ejecución en otros sistemas operativos como macOS o Linux podría requerir ajustes.

ANALISIS DE COSTOS DEL PROYECTO

1. Costos por producción técnica

a) Modelado y texturizado en Maya 3D

- Modelado de entorno completo (casa, interiores, Snoopy, objetos animables):
 - ↳ **25 horas aprox.**
- Texturizado, organización de UVs y materiales:
 - ↳ **10 horas aprox.**
- Simulaciones físicas con *nCloth* (sábanas, tetera):
 - ↳ **5 horas aprox.**
- Total: **40 horas**

b) Exportación y ajustes en Maya

- Separación de objetos animables (.obj individuales)
- Configuración de pivotes para animaciones precisas
- Total: **8 horas**

c) Integración en OpenGL

- Implementación de shaders, luces y cámara
- Programación de sistema de carga de modelos y texturas
- Animación de cada objeto: puertas, botella, copa, TV, manecillas del reloj
- Control con teclado, interpolaciones de animación, y efectos visuales
- Total: **50 horas**

d) Documentación y depuración

- Estructura de carpetas, config. inicial, resolución de errores
- Diagrama de Gantt, guía de uso, notas técnicas
- Total: **10 horas**

2. Tiempo total invertido

11 semanas × 3 horas por día × 7 días = 231 horas aproximadamente

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

3. Valor del conocimiento técnico

Aplicando habilidades en:

- **Maya 3D:** modelado, simulación, exportación de assets
- **OpenGL + C++:** shaders, iluminación, eventos, animación, cámara
- **Assimp + GLEW + GLFW + GLM**
- **Organización de escenas complejas y programación gráfica**

Esto equivale al trabajo de un **artista técnico (Technical Artist)** y un **desarrollador gráfico**, perfiles muy demandados. Aprender esto de forma autodidacta le añade aún más valor.

4. Estimación del precio por hora

Según estándares del mercado:

- Principiante / Estudiante: **\$10–15 USD/hora**

- Intermedio con conocimientos autodidactas (tu caso): **\$20–30 USD/hora**
- Profesional Freelance: **\$40–60 USD/hora**

Usando un valor justo: **\$5USD/hora × 231 horas = \$1,155 USD**

Debido a que en México no se pagan más de 10 dólares la hora.

5. Ajuste por valor entregable

Este proyecto **no es solo un código o modelado aislado**, sino un producto completo y funcional que incluye:

- Escena 3D interactiva
- Elementos animados con control dinámico
- Gráficos optimizados y sistema de navegación
- Documentación técnica y estructura modular
- Versatilidad para futuras adaptaciones o juegos

Se puede considerar como un **prototipo funcional de una escena de videojuego o instalación interactiva**, lo cual permite:

Subir el precio por producto terminado (no solo por horas).

Estimar un valor final **entre \$1,200 y \$1,500 USD** si se presenta como entregable profesional.

Conclusión (resumen para cotización):

Proyecto de desarrollo gráfico 3D e interactivo

Tiempo invertido: 231 horas

Nivel técnico: Básico-Estudiante

Valor estimado: **\$1,200 – \$1,500 USD**

Incluye: Modelado 3D, texturizado, animación, codificación en OpenGL, sistema de navegación, control de eventos y documentación técnica.

METODOLOGÍA DE SOFTWARE APLICADA

Para el desarrollo de este proyecto se aplicó una metodología **incremental y colaborativa**, adaptada a los tiempos establecidos por el calendario académico. Esta metodología permitió construir el proyecto en fases bien definidas, facilitando la integración progresiva de modelos, funcionalidades y animaciones, así como la asignación de tareas entre los dos integrantes del equipo. El proceso de desarrollo se dividió en las siguientes etapas:

Planificación inicial

- Se definió el alcance, la temática (casa de Charlie Brown) y se identificaron los elementos clave a modelar y animar.

Durante la fase inicial del proyecto se estableció que el entorno interactivo estaría inspirado en el universo de Peanuts, específicamente en la casa de Charlie Brown. El alcance abarcó desde el modelado de interiores y exteriores básicos hasta la integración de elementos animados dentro de un entorno navegable. Se determinaron los elementos esenciales a incluir, tales como: la cama, la televisión, la mesa, los libros, la maceta, la tetera, el bowl de frutas, el sofá y el personaje de Snoopy, quien sería el foco de animación principal. Se tomó como referencia tanto la estética original de los dibujos como una adaptación en estilo low poly, con un enfoque visual caricaturesco. Se priorizó que cada objeto tuviera un propósito visual o interactivo, y que en conjunto recrearan un espacio coherente, reconocible y funcional para el usuario.

- Se acordó la distribución de tareas entre los integrantes (modelado, programación, integración y documentación).

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: Odin fue el principal encargado de esta área, desarrollando la mayoría de los objetos 3D en Maya. Esto incluyó el diseño de la estructura interior de la casa, mobiliario, objetos decorativos y la optimización de las mallas para su exportación.

Programación: Fernanda asumió principalmente la responsabilidad del desarrollo del entorno en OpenGL. Se encargó de implementar la carga de modelos con Assimp, la gestión del renderizado, la interacción del usuario y, especialmente, de programar la animación de Snoopy.

Integración: Ambos integrantes colaboraron en esta etapa, colocando los modelos en el entorno interactivo, asegurando que cada objeto tuviera la escala, posición y orientación adecuadas. También se coordinó el ajuste de iluminación, cámara y navegación.

Animación: Odin y Fernanda compartieron la elaboración de las animaciones, cada uno con las animaciones que decidieron hacer en el inicio del curso.

Documentación: Odin y Fernanda compartieron la elaboración del manual técnico, la redacción de la documentación del proyecto, y la producción del video explicativo. También gestionaron la preparación final del repositorio para su entrega y publicación en GitHub.

Diseño y modelado 3D

- Cada habitación y objeto fue diseñado en Autodesk Maya, siguiendo bocetos previos de referencia.

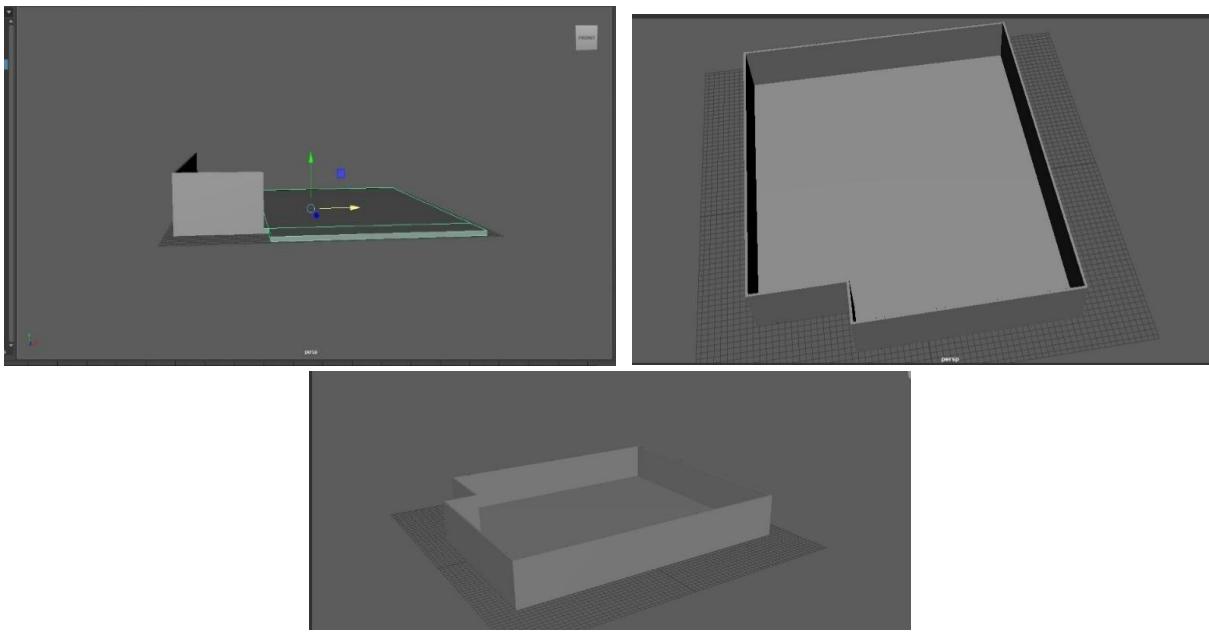
En este punto mencionaremos cómo fue que creamos cada uno de nuestros objetos comenzando con la casa.

- Cada habitación y objeto fue diseñado en Autodesk Maya, siguiendo bocetos previos de referencia.

En este punto mencionaremos cómo fue que creamos cada uno de nuestros objetos comenzando con la casa.

Casa

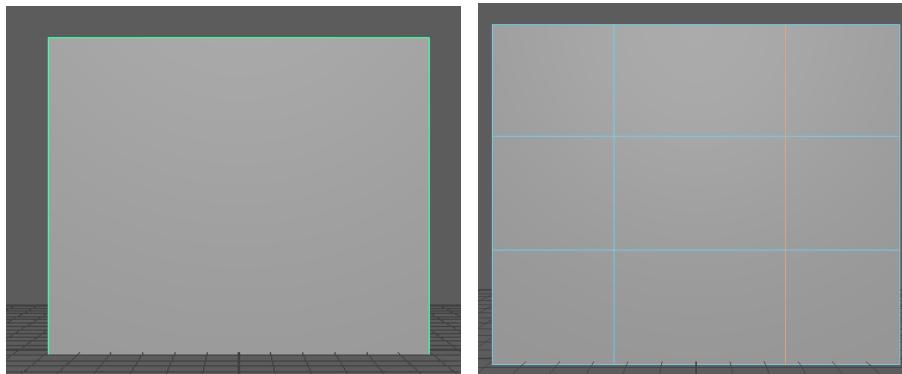
La casa fue realizada principalmente mediante formar cúbicas que vienen de opción dentro de Maya, en este caso, nuestra casa se conforma por 6 paredes con vistas al exterior que le dan forma a la fachada del proyecto.



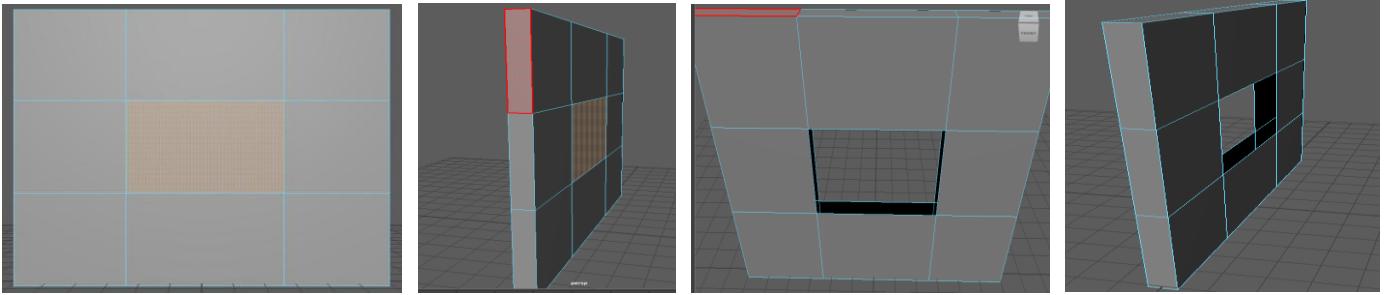
Imágenes iniciales de la estructura de la casa

A estas paredes se les aplicaron diversas herramientas para poder crear los marcos de las ventanas y puertas. Los pasos a seguir para poder crear los marcos fueron los siguientes:

1. Seleccionar el objeto en este caso una pared, una vez seleccionado activar mediante el click derecho la opción Edges. Una vez estando en Edges con la tecla shift derecho y el click derecho activar la opción Insert Edge Loop Tool lo cual te permitirá agregar aristas (que después se convertirán en caras) al cuerpo que observamos, dibujamos los edges de acuerdo con nuestras necesidades ya sea una ventana o una puerta.

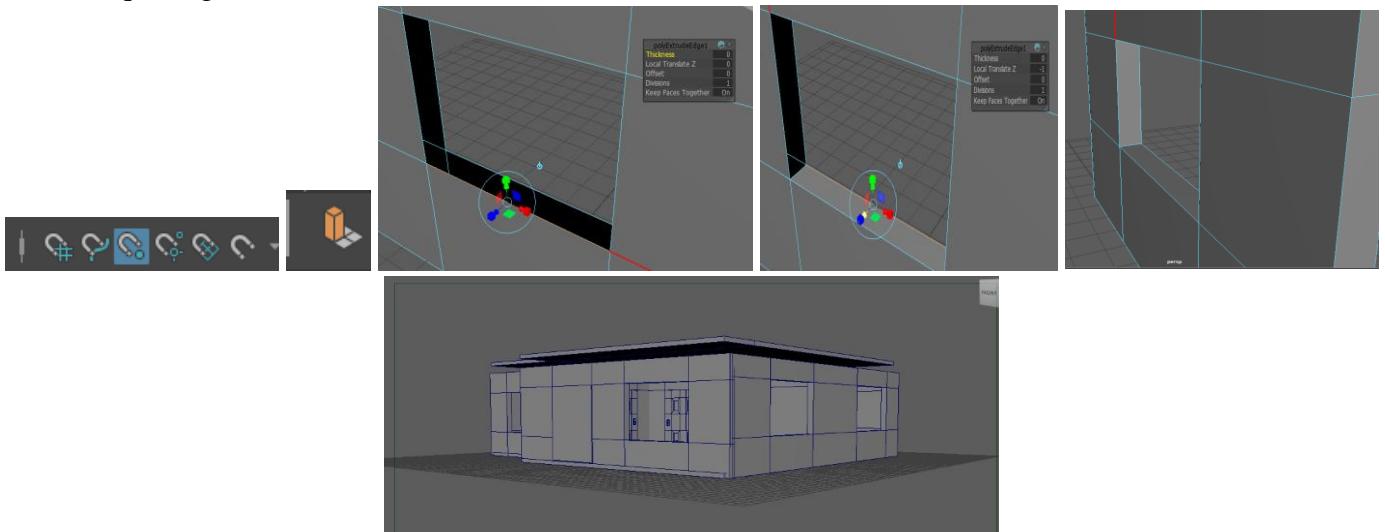


2. Una vez que ya tenemos los nuevos edges dibujados a corde a nuestras necesidades, lo que hicimos fue utilizar la herramienta seleccionar y posteriormente activamos la opción de Face con click derecho sobre el objeto. Seleccionamos las caras que no nos sirvan, en este caso hay que recordar que los cuerpos tienen cierto número de profundidad por lo cual los cuerpos tienen un espacio vacío dentro de ellos y una vez seleccionadas las caras que deseamos eliminar, presionamos la tecla de borrado y veremos ahora un hueco sobre el cuerpo y la parte negra que se ve es el espacio vacío que contiene nuestra pared debido a que el cuerpo tiene profundidad.



Paredes con el hueco de los marcos de la ventana.

- Como ya se mencionó, se queda la pared con el hueco y se puede ver el vacío del cuerpo. Para poder arreglar esto, haremos uso de la herramienta Extrude la cual es una herramienta que funciona para crear geometría a partir de una cara, arista o vértice de un objeto 3D existente. En este caso lo haremos a partir de una arista por lo cual entramos en modo Edge y seleccionamos una arista del marco de la ventana. Una vez seleccionada la arista seleccionamos de la barra de herramientas el Extrude y se pondrá un pivote en medio de la arista seleccionada la cual de momento no la moveremos. Seleccionamos de la barra de herramientas Snap To Points la cual sirve para alinear un objeto, vértice, arista o cara directamente sobre el punto (vértice) de otro objeto, facilitando una colocación precisa para que no queden huecos y sea uniforme al objeto general y repetimos esto con todos los marcos que hagamos.

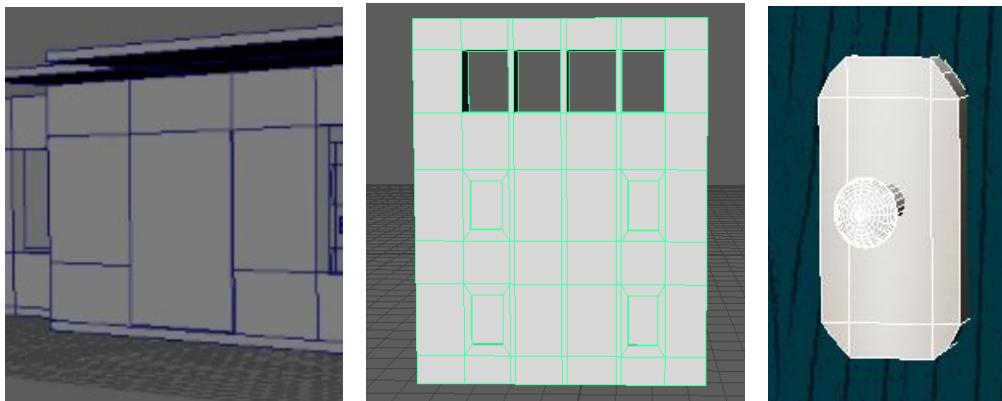


Además, se agregó el techo mediante una calca exacta hecha con Ctrl+D y el ajustamos la posición para que quede donde deseamos, esto solo funcionará para ser la base de nuestros cuerpos que serán después el tejado de la fachada.

- Una vez que tuvimos los marcos de ventanas y puertas, decidimos crear las paredes internas, como tal el único patrón de distribución al espacio fue que en su mayor parte los 3 cuartos que tiene nuestra casa contaran con el mismo o similar espacio, aunque no con la misma forma. De tal manera que quedó de la siguiente manera la distribución.



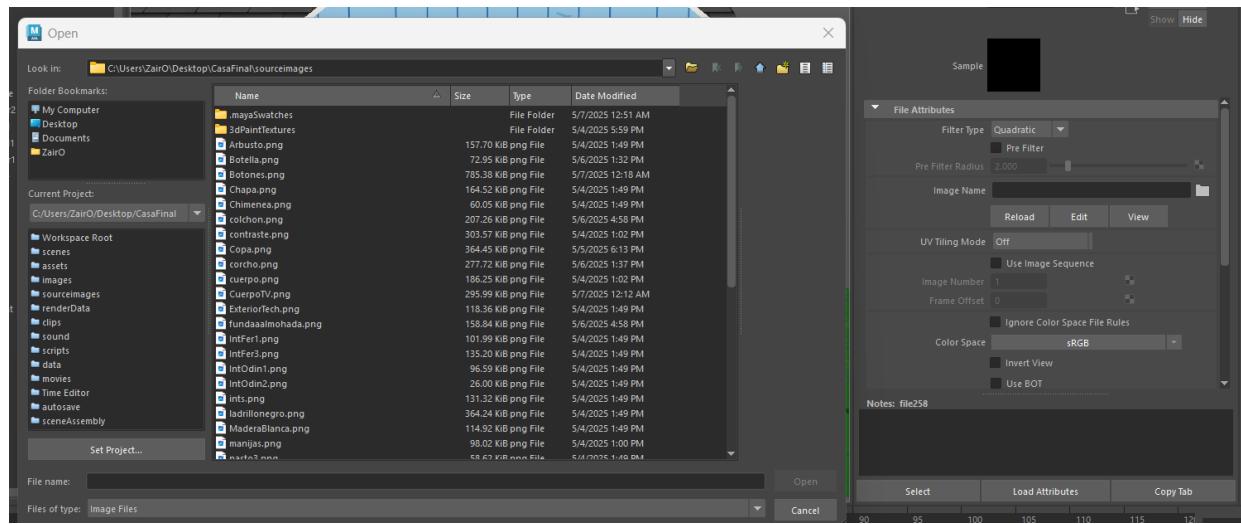
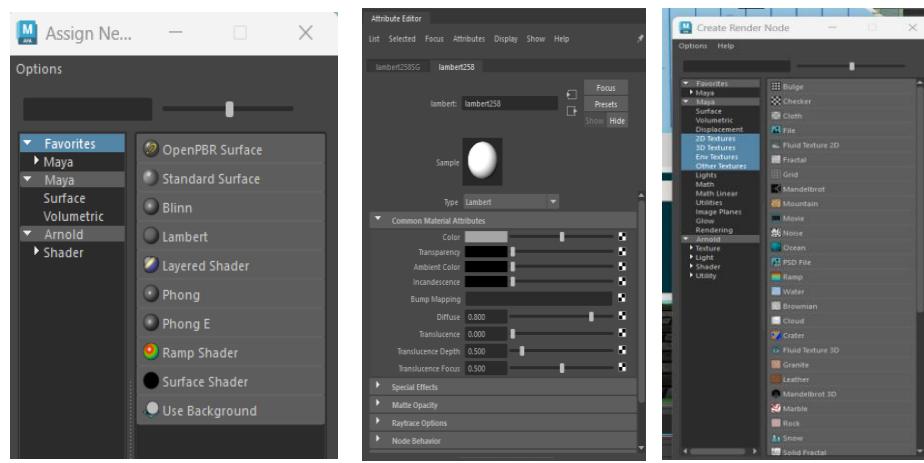
5. En la imagen anterior podemos ver que ya contábamos con puertas, las puertas fueron realizadas mediante una forma cúbica de Maya a la cual primeramente escalamos para que entrara en el hueco de las paredes donde dejamos los marcos, una vez que ya entraban, procedimos a utilizar la herramienta Insert Edge Loop Tool para agregarle aristas al cuerpo y así formar más caras en el cuerpo. Para así luego mediante Extrude darle los 4 detalles de profundidad que tiene una puerta. En el caso de la puerta está formada por 4 cuerpos, un cubo con eliminación de caras y con aplicación de la herramienta Extrude e Insert Edge Loop Tool para hacer los detalles, otro cubo escalado de tal forma que simule la base de una chapa, un cilindro que es parte de la chapa y una esfera modificada en sus vértices para que se achate y sea parte de la agarradera de la chapa.



6. Una parte esencial del proyecto fue la asignación de texturas, para ello recibimos apoyo de inteligencia artificial para la creación de estas. El prompt la mayor parte de las veces fue “*Necesito que me ayudes a generar una textura similar a la imagen de referencia que sea de 512x512 px y sobre todo que sea -tileable- y en formato png*” aunque no siempre se cumplió una similitud al 100%. En algunos casos los prompts también eran de recomendaciones para poder combinar bien los colores dentro de los cuartos. Es importante resaltar que para la mayoría de los objetos se utilizó el material Lambert para poder asignar la textura como lo pueden ser en las paredes, puertas, suelo, techo y algunos objetos complementarios de la casa. El material “Blinn” lo usamos para los vidrios de las ventanas y para algunos objetos como lo pudieron ser la botella y las copas de vidrio.

Para poder asignar una textura en un objeto seguimos los siguientes pasos:

- Seleccionar el objeto o la cara a la cual deseamos asignar la textura.
- Con click derecho seleccionar la opción Assign New Material / New Material.
- Seleccionar el material de acuerdo con las necesidades. Y una vez seleccionado se abrirá el panel lateral de Attribute Editor, en color escogimos algún color que tuviera el objeto o bien en el cuadrado blanco con negro, se presiona y abre un menú donde podemos seleccionar diferentes tipos de material y nosotros siempre elegímos la opción de File la cual debe seleccionar Image Name y agregar el archivo mediante el explorador de archivos.



- d) Es importante utilizar la herramienta UV en Maya, ya que permite definir cómo se aplicará una textura 2D sobre un objeto 3D. El proceso de mapeado UV genera coordenadas que conectan cada punto del modelo con su equivalente en la imagen de textura, asegurando que se visualice correctamente.

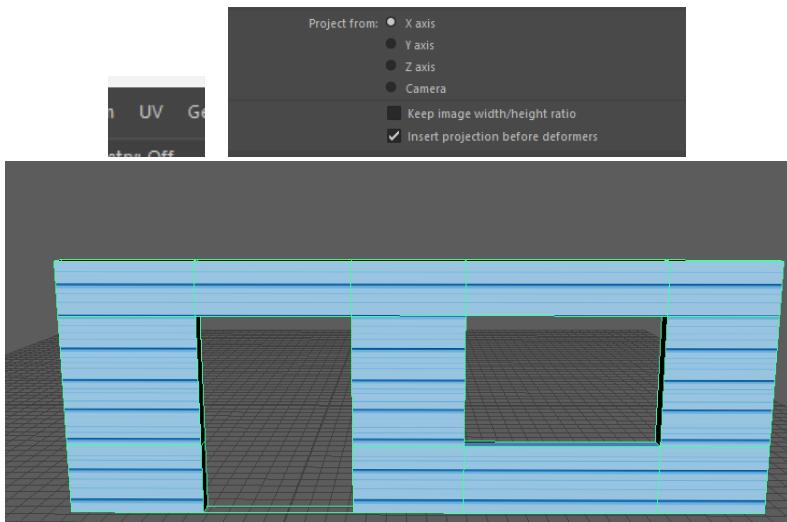
Para ello, es necesario aplicar una de las siguientes proyecciones según la geometría del objeto:

Cylindrical (cilíndrica): Ideal para objetos con forma tubular, como botellas o columnas. La proyección se realiza envolviendo la textura alrededor del eje principal del objeto.

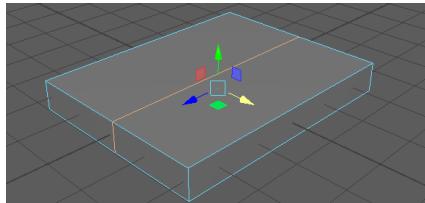
Spherical (esférica): Útil para esferas u objetos redondeados, permite una distribución uniforme de la textura desde el centro hacia afuera.

Planar (plana): Adecuada para superficies planas o ligeramente curvas. Se puede proyectar desde un eje específico (X, Y o Z) o desde la cámara activa, según convenga.

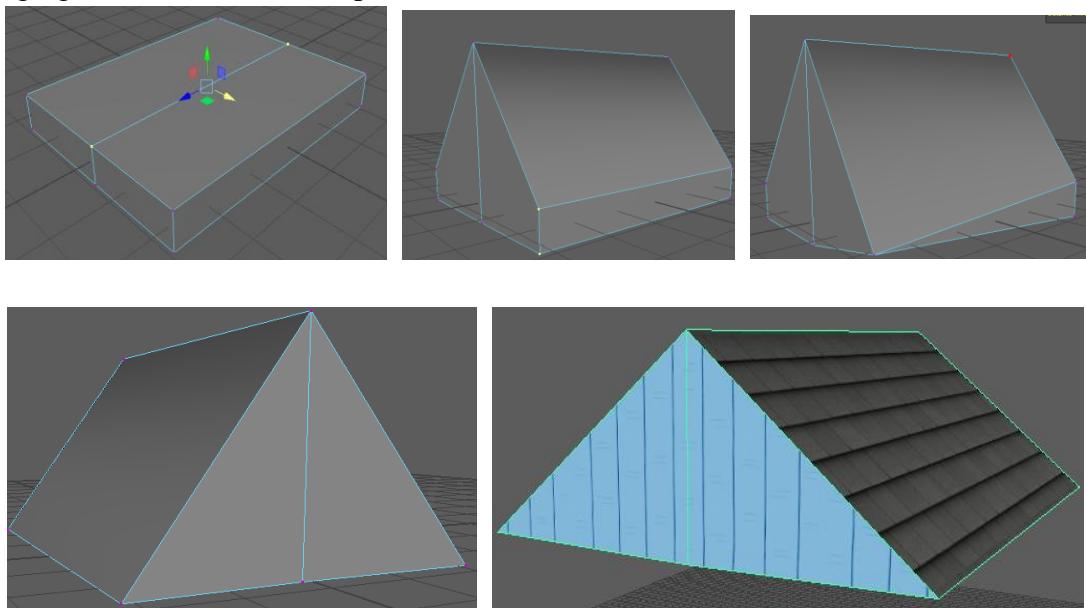
En el caso de objetos cilíndricos o esféricos, se recomienda usar Cylindrical o Spherical respectivamente, proyectándolos desde el cuerpo del objeto. Si el objeto es más plano o irregular, se puede usar Planar, eligiendo el eje que mejor se adapte a la orientación del objeto o utilizando una proyección desde la cámara.



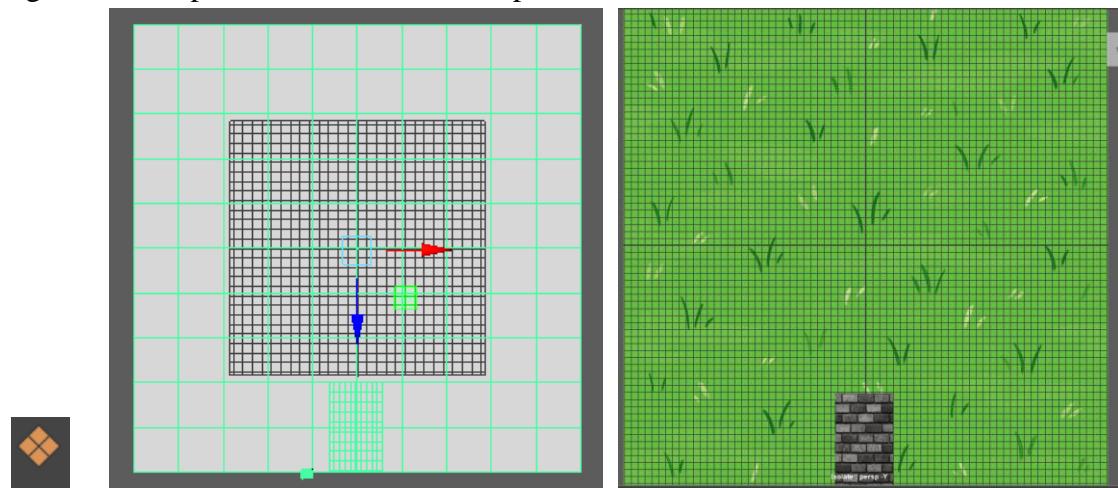
7. Para poder realizar la construcción del techo, ocupamos 2 objetos de forma cúbica, los ajustamos de acuerdo con la división de caras que tiene la base del techo que hicimos en el punto 3, en este caso la base contiene 2 divisiones por lo cual una vez teniendo los 2 cubos ajustados encima de las caras de la base, procedimos a usar Insert Edge Loop Tool y agregamos solo una arista de esta forma:



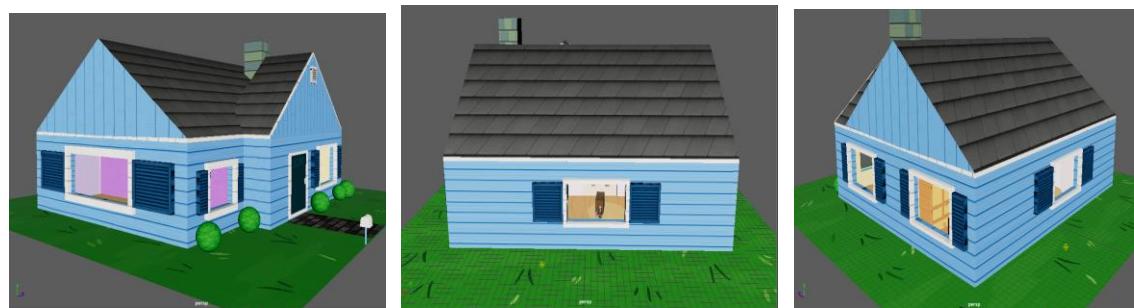
Esto lo que nos permite es poder levantar los vértices del Edge agregado y poder hacer la forma de prisma triangular la cual se requiere para el diseño de nuestro tejado. Para ello fue necesario ocupar la herramienta de Vertex con click derecho. Una vez estando ya dentro de vertex seleccionamos los vértices de en medio y con la herramienta de Move Tool aumentar en el eje Y. Para poder eliminar esa forma de los laterales lo que haremos es seleccionar los vértices laterales de modo que se haga un par, una vez seleccionados utilizamos la herramienta de Merge Vertices y aplicamos esto en las 4 parejas laterales de vértices, esto lo que hace es juntar ambos vértices y convertirlos solo en 1. Por últimos agregamos la textura correspondiente a cada una de las caras.



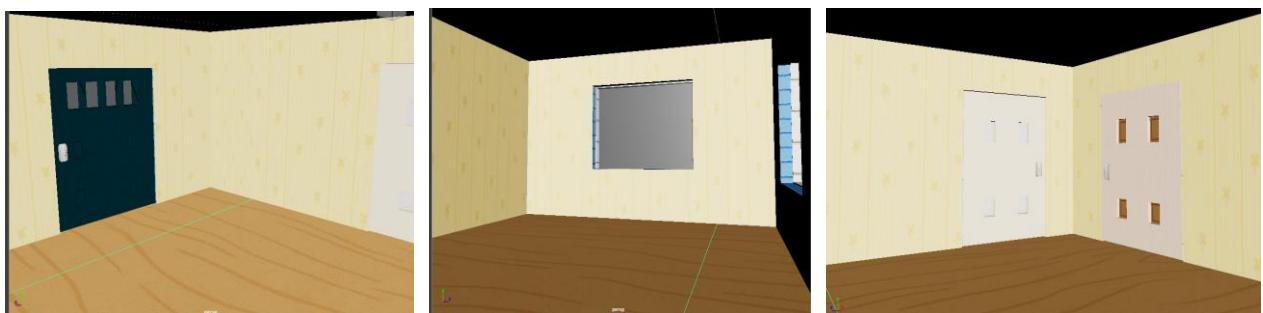
8. Ahora, procedemos a ponerle suelo a nuestro proyecto, para esto ocupamos dos planos de los cuales tenemos disponibles en el software, uno será para el pasto y el otro para el camino de ladrillos, una vez agregados en las posiciones finales, se les pone la textura.



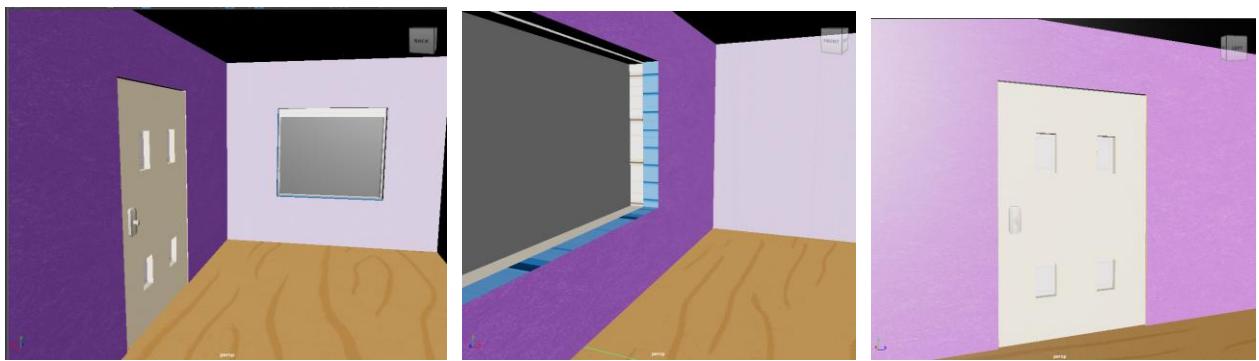
9. Despues de agregar todas las texturas a la fachada y al interior, nuestra casa se ve se la siguiente manera:



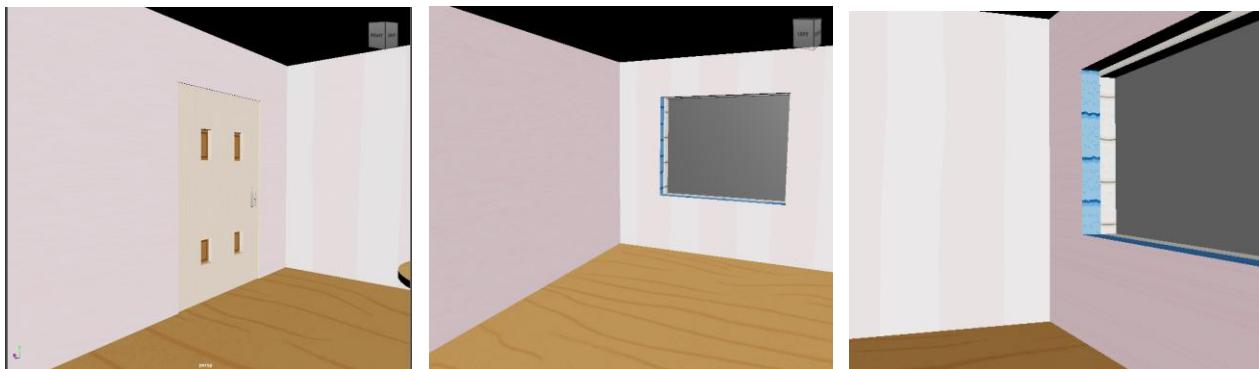
Interior cuarto 1:



Interior cuarto 2:



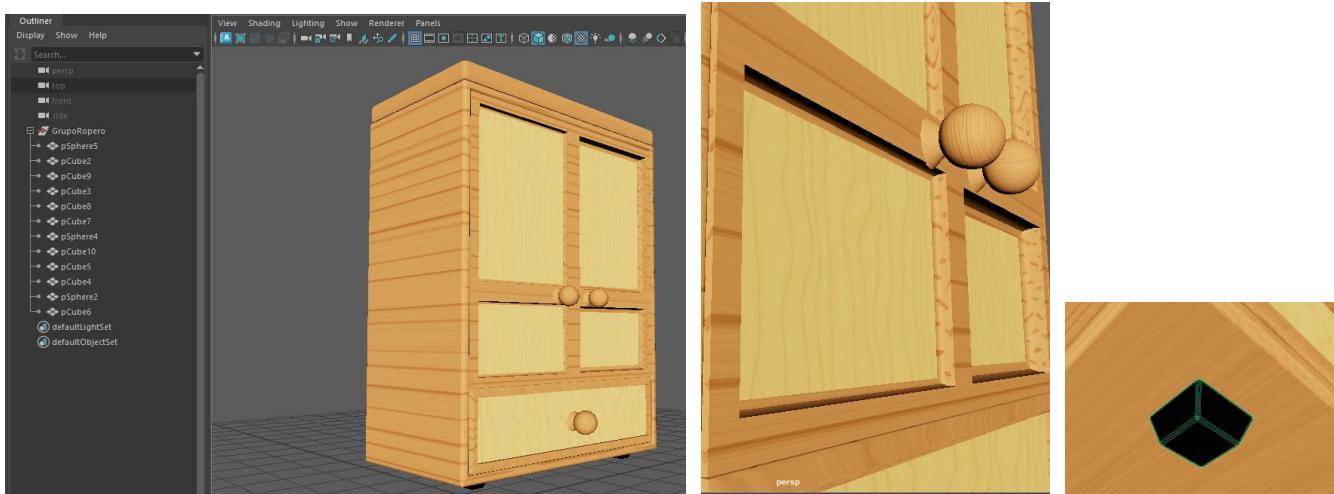
Interior cuarto 3:



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

Ropero:

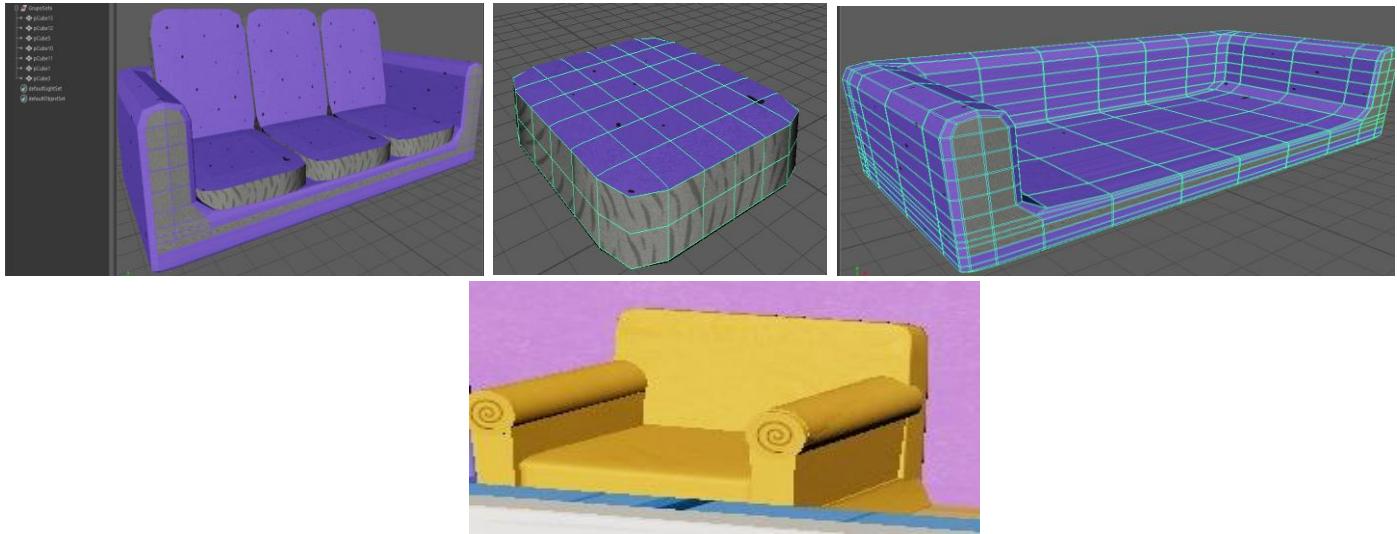
El ropero está construido en su mayoría por cubos y 3 esferas que son las manijas de los cajones. Las puertas del ropero y el cajón están planeadas para animarse en un futuro por lo cual decidimos que sea objetos independientes, estas mismas también se les aplico la herramienta de extrude para dar ese efecto de “profundidad”. Las patas son cubos a los cuales se les aplicó la herramienta Insert Edge Loop Tool para que optaran esa forma. También este objeto es texturizado con un total de 3 texturas.



Sofá:

Fue construido por 6 cubos, el principal es la base del sofá Se agregaron divisiones en altura, ancho y profundidad para permitir modelado más detallado. Se usaron herramientas como Insert Edge Loop Tool para definir bordes

y detalles curvos. La forma curva de los descansabrazos se logró con Extrude, ajustes de vértices. Para los cojines se aplicaron divisiones para redondear los bordes.



Mesa:

Esta escena esta formada por diversos objetos. La televisión esta construida con 2 cubos independientes, el cubo que contiene la textura será animado en un futuro por eso se mantuvo independiente a la caja de la televisión. Ambos objetos se les agrega edges mediante Insert Edge Loop Tool para que cambiaren su forma a una que no se vea completamente cuadrada.

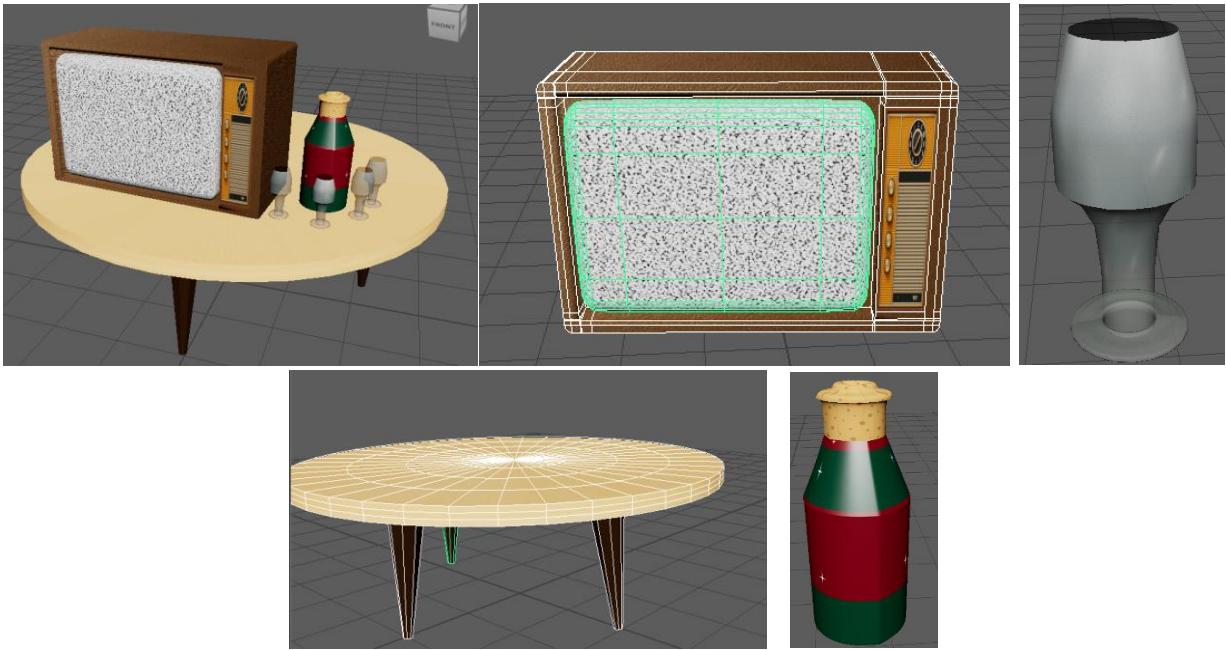
Para las copas fueron modeladas en Maya utilizando un cilindro poligonal como base. A partir de la cara superior, se aplicaron múltiples operaciones de Extrude seguidas de ajustes con Scale para formar la silueta del objeto. Se generaron secciones como la base ancha, el tallo delgado y el cáliz, modificando progresivamente el tamaño de cada extrusión. Finalmente, se insertaron edge loops para definir mejor los bordes y mantener la forma al aplicar suavizado. Este método permitió construir una copa simétrica y detallada mediante modelado poligonal directo.

Con la botella se modeló a partir de un único cilindro poligonal, utilizando la herramienta Extrude para construir toda su forma, incluyendo el corcho. A partir de la base, se realizaron extrusiones hacia arriba para definir el cuerpo principal de la botella, ajustando el Scale en cada paso para generar las transiciones entre las secciones cilíndricas rectas y el estrechamiento del cuello. En la parte superior, se continuó extruyendo para formar el cuello y luego el corcho, ampliando ligeramente la última extrusión para crear el borde ancho que lo distingue. Todo esto se logró sin separar geometrías, manteniendo la continuidad del objeto en un solo mesh. Finalmente, se aplicaron materiales y texturas a diferentes secciones del modelo utilizando mapeo UV para simular la etiqueta navideña y la apariencia del corcho. Este enfoque permitió una forma coherente y eficiente sin necesidad de añadir objetos adicionales.

La mesa fue modelada en Maya a partir de geometrías básicas. La superficie de la mesa se construyó utilizando un polyCylinder, al cual se le ajustó la altura para hacerlo delgado y se le aumentaron las subdivisiones radiales para mejorar la definición del borde. Posteriormente, se aplicó una textura de madera clara para simular el acabado superior.

Las patas de la mesa se modelaron a partir de pCubes, que fueron escalados de forma no uniforme para lograr una forma trapezoidal (más ancha en la parte superior y más estrecha en la base). Se duplicaron las patas y se

posicionaron en los cuatro extremos del cilindro para mantener simetría y soporte. Finalmente, se asignó un material de madera oscura a las patas para diferenciar visualmente los elementos de la mesa.



Cama:

La estructura principal de la cama (colchón, cabecera y base) se construyó con pCubes, escalados y posicionados para definir el volumen general. Las almohadas también se realizaron con cubos suavizados o con subdivisiones para darles una apariencia mullida.

La sábana fue creada a partir de un Plane, el cual se posicionó encima del colchón. Para simular el comportamiento realista de la tela, a este plano se le aplicó el sistema de simulación nCloth, transformándolo en una malla dinámica. Se ajustaron sus atributos físicos para darle un comportamiento de caída natural sobre la cama. Las colisiones se configuraron utilizando nRigid sobre el colchón y almohadas, de modo que la tela pudiera adaptarse correctamente a la forma de estos objetos.

Finalmente, se utilizó la línea de tiempo de la escena para reproducir la simulación. Tras algunos fotogramas, cuando la sábana alcanzó una posición natural y estable, se detuvo la reproducción y se aplicó el estado de la simulación como pose final, dando como resultado una sábana que se ve suavemente caída y adaptada al colchón.



Reloj Pared:

Cuerpo principal:

Se creó utilizando un cilindro que fue rotado de forma que solo se muestre una de sus caras frontales. Este cilindro actúa como el marco del reloj, dándole un borde tridimensional con apariencia de madera o material sólido.

Texturizado:

Al cilindro se le aplicó un material tipo Lambert, lo que proporciona un acabado mate, sin reflejos especulares, ideal para representar materiales como madera o plástico opaco.

Manecilla de horas y minutos:

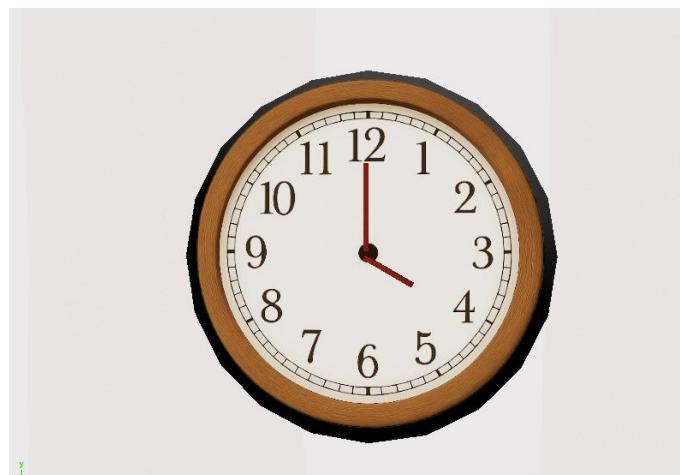
Ambas se modelaron con pCube (cubos) individuales. Luego fueron escaladas para adoptar una forma delgada y alargada, apropiada para simular agujas de reloj.

Ubicación y pivote:

Cada cubo fue reubicado de forma precisa para que su pivote coincida con el centro del reloj, permitiendo una rotación limpia desde el centro.

Textura y color:

También utilizan un material Lambert, probablemente con colores planos (por ejemplo, rojo para los minutos y negro para las horas) para diferenciarlas.



Casa Snoopy:

La casita fue creada usando únicamente cubos, sin necesidad de otras primitivas. Para formar la estructura principal (paredes y base), se utilizó un cubo escalado en los ejes X, Y y Z para darle la forma de un bloque rectangular. El techo a dos aguas se construyó también a partir de cubos rotados en un ángulo y posicionados simétricamente sobre la parte superior de la estructura, simulando el típico techo inclinado.



Mesas rectangulares:

Cada mesa fue construida de forma sencilla utilizando primitivas básicas:

- Tablero superior: Un pCube escalado en los ejes X y Z para formar la superficie rectangular de la mesa.

- Patas: También son pCubes, escalados de forma no uniforme (más altos en Y y más delgados en X/Z) para darles forma de patas cónicas o trapezoidales. Se duplicaron y posicionaron en las esquinas de la tabla.

Los libros fueron modelados con cubos simples (pCubes) escalados para representar la forma rectangular de un libro cerrado. Se duplicaron y rotaron ligeramente para generar una composición natural (uno encima del otro, con ángulo).

El televisor también se modeló con pCubes: Un cubo grande para el cuerpo principal. Otro más delgado como pantalla.

A diferencia de los anteriores, la tetera y el bowl no fueron hechos con cubos. Se utilizaron primitivas más específicas:

- La tetera fue creada con la forma predefinida teapot.
- El bowl fue generado con un polySphere, al cual se le eliminaron las caras superiores y se aplicó Extrude hacia adentro para darle profundidad.
- Las naranjas son esferas (pSpheres) colocadas dentro del bowl.



Maceta:

Se utilizó un polyCylinder como base. Se escaló en la parte superior para hacerla más ancha que la base, creando una forma troncocónica (como una maceta real). Posteriormente, se seleccionaron las caras superiores y se aplicó un Extrude hacia adentro con “Offset”, y luego un Extrude hacia abajo para crear el hueco interno de la maceta. Las hojas fueron duplicadas y rotadas para que salieran del centro en distintas direcciones.

Cada hoja fue probablemente creada a partir de pCube muy delgado, al cual se le ajustaron los vértices para dar curvatura y forma de hoja puntiaguda.



→ Los modelos fueron exportados en formato .obj cuidando la geometría, escalado y compatibilidad con OpenGL. Px

Una vez completado el modelado de los objetos en Autodesk Maya, se procedió a la exportación de cada uno en formato .obj, el cual es ampliamente compatible con motores gráficos y librerías como Assimp y OpenGL. Durante este proceso se prestó especial atención a varios aspectos técnicos clave:

Geometría optimizada: Se verificó que cada modelo tuviera una malla limpia y sin errores, evitando caras invertidas, normales mal orientadas, polígonos no manifolds o innecesarios. También se aplicaron combinaciones de objetos cuando fue necesario para simplificar la estructura jerárquica del archivo.

Escalado uniforme: Se ajustó la escala global de los modelos para que fueran coherentes entre sí y adecuados al entorno general. Esto evitó problemas de visualización y permitió que al integrarlos en OpenGL no requirieran transformaciones excesivas.

Pivotes y orientación: Se centraron los pivotes de los objetos y se alinearon correctamente respecto al eje Y (arriba) para asegurar una integración directa y sin desajustes en la escena renderizada.

Compatibilidad con OpenGL: Antes de exportar, se congelaron las transformaciones y se eliminaron los historiales de construcción en Maya para evitar inconsistencias. Además, se exportaron sin materiales embebidos, ya que en OpenGL se cargan mediante shaders y texturas separadas. Se garantizó que los nombres de los archivos fueran claros y consistentes para facilitar su carga mediante código.

Desarrollo incremental del software

- Se inició con la base gráfica del proyecto usando C++ y OpenGL, incorporando bibliotecas como GLFW, GLEW y Assimp.

Una vez definidos los modelos y su formato de exportación, se procedió a desarrollar la estructura base del entorno gráfico utilizando el lenguaje de programación C++ en conjunto con OpenGL para el renderizado en tiempo real. Se integraron tres bibliotecas fundamentales para facilitar este proceso:

GLFW fue utilizada para la creación de ventanas, gestión del contexto de OpenGL y el manejo de eventos de entrada del usuario (teclado y mouse). Esto permitió definir la resolución, habilitar el modo interactivo y capturar acciones del usuario para mover la cámara o activar funciones dentro de la escena.

GLEW (OpenGL Extension Wrangler) se incorporó para acceder a las extensiones modernas de OpenGL, asegurando la compatibilidad con shaders personalizados y funcionalidades avanzadas como VBOs (Vertex Buffer Objects) y VAOs (Vertex Array Objects).

Assimp (Open Asset Import Library) se empleó para cargar los modelos en formato .obj exportados desde Maya. Esta biblioteca facilitó la lectura de geometrías, jerarquías y materiales, convirtiéndolos en estructuras aptas para renderizado en OpenGL.

Con estas herramientas se estableció un entorno gráfico estable, modular y escalable, sirviendo como la base sobre la cual se irían incorporando los elementos del proyecto.

- Se integraron progresivamente los modelos exportados, añadiendo texturas e iluminación.

A partir de la base gráfica ya funcional, se comenzó la integración de los modelos 3D previamente creados y exportados desde Maya. Esta fase se realizó de manera progresiva, agregando los elementos uno por uno dentro del entorno visual para mantener el control sobre su posicionamiento, escalado y apariencia.

Durante la integración:

Se asociaron texturas y materiales a cada modelo, cargándolos como imágenes externas (normalmente en formato .png o .jpg) y mapeándolos correctamente mediante coordenadas UV previamente definidas en Maya.

Se utilizaron shaders en GLSL para manejar los materiales, aplicando iluminación básica con modelos como Phong o Blinn-Phong, lo que permitió simular brillo espectral, difuso y ambiental, mejorando el realismo visual de los objetos.

La iluminación se configuró utilizando luces direccionales y, en algunos casos, luces puntuales o spotlights para destacar zonas específicas del entorno.

Este proceso aseguró una presentación visual coherente y atractiva, facilitando además la detección temprana de errores o desfases en los modelos.

Pruebas continuas

- A lo largo del desarrollo, se realizaron pruebas constantes para verificar la carga correcta de modelos, ajustes visuales y funcionamiento de animaciones.
- Se priorizó la corrección de errores de carga, iluminación defectuosa o problemas de interacción.

Optimización y documentación

- Se optimizaron los modelos para mejorar el rendimiento sin comprometer la calidad visual.
- Se elaboró una documentación técnica que detalla tanto el proceso como el uso del software, incluyendo un video explicativo.

Control de versiones y colaboración

- Se utilizó Git y GitHub para organizar y respaldar el código fuente, los modelos y recursos del proyecto.
- Esto permitió un flujo de trabajo coordinado y facilitó el seguimiento del progreso.

Esta metodología permitió un equilibrio entre la libertad creativa (diseño artístico) y el enfoque técnico (programación y documentación), asegurando que el resultado fuera funcional, visualmente coherente y ejecutable en tiempo real.

DOCUMENTACION DE LA ANIMACIONES

Para las animaciones de mis objetos se implementaron las siguientes variables globales que posteriormente nos servirán para los procesos de control de animación y para la lógica de estas. Algunas de estas tienen ya con bandera de false ya que nos servirán para guardar los estados de las animaciones y para evitar toggles.

```

//Variables dePuertaPrincipal
float rotacionPuertaPrincipal = 0.0f;
bool puertaPrincipalAbierta = false;
bool animandoPuertaPrincipal = false;

//Variables para botella
float rotacionBotella = 0.0f;
bool botellaCayendo = false;      // Si se está animando
bool botellaCaida = false;        // Estado actual (caída o no)
bool toggleBotella = false;       // Control de tecla

//Variables para la copa
float anguloCopa = 0.0f;
bool temblarCopa = false;
bool toggleCopa = false;

//Variables para la SeñalTv
float ruidoTimer = 0.0f;
float ruidoIntervalo = 0.1f; // velocidad de cambio entre frames
int ruidoFrameActual = 0;
bool animarRuido = false;
static bool teclaPresionada = false;

//Variables para PuertaOdin
float rotacionPuertaOdin = 0.0f;
bool puertaOdinAbierta = false;
bool animandoPuertaOdin = false;

//Variables PuertaRoperol
float rotacionRoperol = 0.0f;    // Comienza cerrada
bool puertaAbierta = false;       // Estado actual
bool animandoRoperol = false;     // Evita spam del toggle

//Variable PuertaRopero2
// Variable PuertaRopero2
float rotacionRopero2 = 0.0f;      // Comienza cerrada
bool puertaRopero2Abierta = false;  // Estado actual
bool animandoRopero2 = false;       // Evita spam del toggle

```

```

//Variables Manecilla Hora
bool animarHora = false;           // Controla si está activa la animación
bool toggleHora = false;           // Evita múltiples toggles al dejar presionada la tecla
float anguloHora = 0.0f;            // Ángulo acumulado

//Variables Manecilla Minutos
bool animarMinutos = false;
bool toggleMinutos = false;
float anguloMinutos = 0.0f;

```

El siguiente bloque de código es parte de la función “DoMovement” la cual almacena la información de las teclas presionadas. Teniendo de esta manera las siguientes teclas interactivas con el usuario.

Acción	Tecla
Abrir/Cerrar Puerta Cuarto Principal	P
Abrir/Cerrar Puerta Cuarto de Odin	O
Abrir/Cerrar Puerta Ropero 1	M
Abrir/Cerrar Puerta Ropero 2	N
Tirar/Parar Botella	B
Vibración Copa/Parar Vibración	C
Manecillas del reloj (Pueden presionarse al mismo tiempo o en tiempos diferentes, lo ideal es al mismo tiempo) / Pararlas	I – Manecilla de las horas U – Manecilla de los minutos
Señal de TV	R

```

//PuertaPrincipal
if (keys[GLFW_KEY_P] && !animandoPuertaPrincipal) {
    puertaPrincipalAbierta = !puertaPrincipalAbierta;
    animandoPuertaPrincipal = true;
}
if (!keys[GLFW_KEY_P]) {
    animandoPuertaPrincipal = false;
}

//Botella
if (keys[GLFW_KEY_B] && !toggleBotella) {
    botellaCaida = !botellaCaida;           // Cambia el estado
    botellaCayendo = true;                  // Inicia animación
    toggleBotella = true;
}
if (!keys[GLFW_KEY_B]) {
    toggleBotella = false;
}

//Copa
if (keys[GLFW_KEY_C] && !toggleCopa) {
    temblarCopa = !temblarCopa;   // Cambia el estado
    toggleCopa = true;
}
if (!keys[GLFW_KEY_C]) {
    toggleCopa = false;
}

//Señal
if (keys[GLFW_KEY_R] && !teclaPresionada) {
    animarRuido = !animarRuido;
    teclaPresionada = true;
}
if (!keys[GLFW_KEY_R]) {
    teclaPresionada = false;
}

//PuertaRopero 1
if (keys[GLFW_KEY_M] && !animandoRopero1) {
    puertaAbierta = !puertaAbierta; // Cambia el estado
    animandoRopero1 = true;        // Evita múltiples toggles
}
if (!keys[GLFW_KEY_M]) {
    animandoRopero1 = false;       // Se suelta la tecla
}

// PuertaRopero2
if (keys[GLFW_KEY_N] && !animandoRopero2) {
    puertaRopero2Abierta = !puertaRopero2Abierta; // Cambia el estado
    animandoRopero2 = true;                      // Evita múltiples toggles
}
if (!keys[GLFW_KEY_N]) {
    animandoRopero2 = false;                   // Se suelta la tecla
}

//PuertaOdin
if (keys[GLFW_KEY_O] && !animandoPuertaOdin) {
    puertaOdinAbierta = !puertaOdinAbierta;
    animandoPuertaOdin = true;
}
if (!keys[GLFW_KEY_O]) {
    animandoPuertaOdin = false;
}

//PuertaFer
if (keys[GLFW_KEY_F] && !togglePuertaFer) {
    estadoPuertaFer = !estadoPuertaFer;
    togglePuertaFer = true;
}
if (!keys[GLFW_KEY_F]) {
    togglePuertaFer = false;
}

// Manecilla Hora
if (keys[GLFW_KEY_I] && !toggleHora) {
    animarHora = !animarHora;
    toggleHora = true;
}
if (!keys[GLFW_KEY_I]) {
    toggleHora = false;
}

// Manecilla Minutos
if (keys[GLFW_KEY_U] && !toggleMinutos) {
    animarMinutos = !animarMinutos;
    toggleMinutos = true;
}
if (!keys[GLFW_KEY_U]) {
    toggleMinutos = false;
}

```

Los siguientes bloques de código son la lógica de animación el renderizado de los objetos, ambos claramente diferenciados cuando inician y acaban sus procesos

```

// =====
//     PuertaPrincipal
// =====
// Animar rotación progresiva de PuertaPrincipal
// ANIMACIÓN
if (puertaPrincipalAbierta && rotacionPuertaPrincipal > -120.0f) {
    rotacionPuertaPrincipal -= 50.0f * deltaTime;
    if (rotacionPuertaPrincipal < -120.0f)
        rotacionPuertaPrincipal = -120.0f;
} else if (!puertaPrincipalAbierta && rotacionPuertaPrincipal < 0.0f) {
    rotacionPuertaPrincipal += 50.0f * deltaTime;
    if (rotacionPuertaPrincipal > 0.0f)
        rotacionPuertaPrincipal = 0.0f;
}

// RENDERIZADO
model = glm::mat4(1.0f);
glm::vec3 pivotPrincipal = glm::vec3(-4.068352699279785f, 4.48995304107666f, 21.405784606933594f);
model = glm::translate(model, pivotPrincipal);
model = glm::rotate(model, glm::radians(rotacionPuertaPrincipal), glm::vec3(0.0f, 1.0f, 0.0f));
model = glm::translate(model, -pivotPrincipal);
glm::UniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
PuertaPrincipal.Draw(LightingShader);

// =====
//     PuertaOdin
// =====
// ANIMACIÓN
if (puertaOdinAbierta && rotacionPuertaOdin > -120.0f) {
    rotacionPuertaOdin -= 50.0f * deltaTime;
    if (rotacionPuertaOdin < -120.0f)
        rotacionPuertaOdin = -120.0f;
} else if (!puertaOdinAbierta && rotacionPuertaOdin < 0.0f) {
    rotacionPuertaOdin += 50.0f * deltaTime;
    if (rotacionPuertaOdin > 0.0f)
        rotacionPuertaOdin = 0.0f;
}

// RENDERIZADO
model = glm::mat4(1.0f);
glm::vec3 pivotOdin = glm::vec3(1.004035472869873f, 4.038388252258301f, -2.5722479820251465f);
model = glm::translate(model, pivotOdin);
model = glm::rotate(model, glm::radians(rotacionPuertaOdin), glm::vec3(0.0f, 1.0f, 0.0f)); // Eje Y
model = glm::translate(model, -pivotOdin);
glm::UniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
PuertaOdin.Draw(LightingShader);

```

```
// =====
//      Copo
// =====
//ANIMACION
if (temblarCopo) {
    anguloCopo = sin(glm::getTime() * 10.0f) * 5.0f; // Oscila suavemente entre -5° y +5°
}

//RENDERIZADO
model = glm::mat4(1.0f);
glm::vec3 pivotCopo = glm::vec3(-0.959242f, 3.597939f, -17.372900f);
model = glm::translate(model, pivotCopo);
model = glm::rotate(model, glm::radians(anguloCopo), glm::vec3(0.0f, 0.0f, 1.0f)); // Gira sobre eje Z (ajustable)
model = glm::translate(model, -pivotCopo);
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
Copo.Draw(LightingShader);

// =====
//      SeñalTv
// =====
//RENDERIZADO
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, ruidoFrames[ruidoFrameActual]);
model = glm::mat4(1.0f);
model = glm::translate(model, glm::vec3()); // Usa la posición adecuada
 glEnable(GL_BLEND);
 glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
 glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
 glUniform1i(glGetUniformLocation(lightingShader.Program, "transparency"), 0);
SenalTV.Draw(LightingShader);
```

```

// =====
//      Botella
// =====
//ANIMACION
if (botellaCayendo) {
    float velocidad = 45.0f * deltaTime;

    if (botellaCaida && rotacionBotella < 90.0f) {
        rotacionBotella += velocidad;
        if (rotacionBotella >= 90.0f) {
            rotacionBotella = 90.0f;
            botellaCayendo = false;
        }
    }
    else if (!botellaCaida && rotacionBotella > 0.0f) {
        rotacionBotella -= velocidad;
        if (rotacionBotella <= 0.0f) {
            rotacionBotella = 0.0f;
            botellaCayendo = false;
        }
    }
}

//RENDERIZADO
model = glm::mat4(1.0f);
glm::vec3 pivotBotella = glm::vec3(-2.303459f, 4.206707f, -17.304603f);
model = glm::translate(model, pivotBotella);
model = glm::rotate(model, glm::radians(rotacionBotella), glm::vec3(0.0f, 0.0f, 1.0f));
model = glm::translate(model, -pivotBotella);

glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
Botella.Draw(lightningShader);

// =====
//      PuertaRoperol
// =====
//ANIMACION
if (puertaAbierta && rotacionRoperol > -120.0f) {
    rotacionRoperol -= 50.0f * deltaTime;
    if (rotacionRoperol < -120.0f)
        rotacionRoperol = -120.0f;
}
else if (!puertaAbierta && rotacionRoperol < 0.0f) {
    rotacionRoperol += 50.0f * deltaTime;
    if (rotacionRoperol > 0.0f)
        rotacionRoperol = 0.0f;
}

// RENDERIZADO
model = glm::mat4(1.0f);
glm::vec3 pivotRoperol = glm::vec3(15.191308f, 4.863689f, -8.121943f);

model = glm::translate(model, pivotRoperol);
model = glm::rotate(model, glm::radians(rotacionRoperol), glm::vec3(0.0f, -1.0f, 0.0f)); // Eje Y
model = glm::translate(model, -pivotRoperol);
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
PuertaRoperol.Draw(lightningShader);

```

```

// =====
//      PuertaRopero2
// =====
// ANIMACIÓN
if (puertaRopero2Abierta && rotacionRopero2 > -120.0f) {
    rotacionRopero2 -= 50.0f * deltaTime;
    if (rotacionRopero2 < -120.0f)
        rotacionRopero2 = -120.0f;
}
else if (!puertaRopero2Abierta && rotacionRopero2 < 0.0f) {
    rotacionRopero2 += 50.0f * deltaTime;
    if (rotacionRopero2 > 0.0f)
        rotacionRopero2 = 0.0f;
}
//RENDERIZADO
model = glm::mat4(1.0f);
glm::vec3 pivotRopero2 = glm::vec3(20.6259765625f, 4.822100639343262f, -8.121943473815918f);
model = glm::translate(model, pivotRopero2);
model = glm::rotate(model, glm::radians(rotacionRopero2), glm::vec3(0.0f, 1.0f, 0.0f)); // Eje Y
model = glm::translate(model, -pivotRopero2);
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
PuertaRopero2.Draw(lightingShader);

// =====
//      Personaje
// =====
//RENDERIZADO
model = glm::mat4(1.0f);
model = glm::translate(model, playerPosition);
model = glm::scale(model, glm::vec3(0.7f)); // Ajusta el tamaño del modelo
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
Personaje.Draw(lightingShader);

// =====
//      Cuerpo del Reloj
// =====
//RENDERIZADO
glm::mat4 modelReloj = glm::mat4(1.0f);

glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(modelReloj));
Reloj.Draw(lightingShader);

// =====
//      Manecilla Hora
// =====
//ANIMACION
if (animarHora) {
    anguloHora += deltaTime * 6.0f; // 6 por segundo 1 vuelta en 60 segundos
    if (anguloHora >= 360.0f) anguloHora -= 360.0f;
}

//RENDERIZADO
glm::mat4 modelHora = glm::mat4(1.0f);
glm::vec3 pivotHora = glm::vec3(-9.090096001747589f, 4.95666558468782f, -8.980611363899742f);
modelHora = glm::translate(modelHora, pivotHora);

```

```

glm::vec3 pivotHora = glm::vec3(-9.090096001747589f, 4.95666558468782f, -8.980611363899742f);
modelHora = glm::translate(modelHora, pivotHora);
modelHora = glm::rotate(modelHora, glm::radians(anguloHora), glm::vec3(-1.0f, 0.0f, 0.0f));
modelHora = glm::translate(modelHora, -pivotHora);
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(modelHora));
ManecillaHora.Draw(lightingShader);

// =====
//      Manecilla Minutos
// =====
//ANIMACION
if (animarMinutos) {
    anguloMinutos += deltaTime * 72.0f; // Coordinado con ManecillaHora (1 vuelta en 5s)
    if (anguloMinutos >= 360.0f) anguloMinutos -= 360.0f;
}

//RENDERIZADO
glm::mat4 modelMin = glm::mat4(1.0f);
glm::vec3 pivotMin = glm::vec3(-9.107255224796003f, 4.902407769451714f, -8.968057866693192f);
modelMin = glm::translate(modelMin, pivotMin);
modelMin = glm::rotate(modelMin, glm::radians(anguloMinutos), glm::vec3(-1.0f, 0.0f, 0.0f)); // Eje igual que en Maya
modelMin = glm::translate(modelMin, -pivotMin);
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(modelMin));
ManecillaMinutos.Draw(lightingShader);

// =====
// FIN DE RENDERIZADO DE OBJETOS
// =====

```

DOCUMENTACIÓN DEL CÓDIGO

A continuación, mencionaremos los archivos de los cuales consta nuestro proyecto y daremos una explicación de estos.

ProyectoFinal.cpp:

Este archivo es el núcleo principal del programa, responsable de inicializar el entorno gráfico, cargar shaders y modelos, gestionar entradas del usuario y renderizar la escena 3D del proyecto interactivo (casa de Charlie Brown con Snoopy animado). Se usa C++ junto con OpenGL, GLFW, GLEW, GLM y Assimp.

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.
- GLM: Proporciona operaciones matemáticas para vectores y matrices.
- SOIL2 y stb_image.h: Cargan texturas.
- Shader.h, Camera.h, Model.h: Archivos personalizados que encapsulan funciones de shader, cámara y carga de modelos .obj.

Prototipos de funciones

Se declaran tres funciones que luego serán definidas:

- KeyCallback(...): Maneja entradas del teclado.
- MouseCallback(...): Maneja movimiento del mouse.
- DoMovement(): Ejecuta el movimiento del personaje u objetos según las teclas presionadas.

Dimensiones de ventana

Se definen las dimensiones iniciales de la ventana de renderizado:

Cámara y movimiento

Se inicializa una cámara en tercera persona, con:

- Posición base: (0.0f, 0.0f, 3.0f)
- Control de mouse (firstMouse, lastX, lastY)
- keys[1024]: Arreglo para detectar qué teclas están presionadas.
- cameraOffset: Posición de la cámara respecto al personaje, simulando vista en tercera persona.

Luz y jugador

- lightPos: Posición inicial de una luz (inicializada en cero).
- playerPosition: Posición del personaje/jugador en el mundo.
- cameraOffset: Define desde dónde mira la cámara al jugador.

```
✓ #include <iostream>
└ #include <cmath>

// GLEW
#include <GL/glew.h>

// GLFW
#include <GLFW/glfw3.h>

// Other Libs
#include "stb_image.h"

// GLM Mathematics
✓ #include <glm/glm.hpp>
└ #include <glm/gtc/matrix_transform.hpp>
  #include <glm/gtc/type_ptr.hpp>

// Load Models
#include "SOIL2/SOIL2.h"

// Other includes
✓ #include "Shader.h"
└ #include "Camera.h"
  #include "Model.h"
```

```
// Function prototypes
void KeyCallback(GLFWwindow* window, int key, int scanCode, int action, int mode);
void MouseCallback(GLFWwindow* window, double xPos, double yPos);
void DoMovement();

// Window dimensions
const GLuint WIDTH = 800, HEIGHT = 600;
int SCREEN_WIDTH, SCREEN_HEIGHT;

// Camera
Camera camera(glm::vec3(0.0f, 0.0f, 3.0f));
GLfloat lastX = WIDTH / 4.0;
GLfloat lastY = HEIGHT / 4.0;
bool keys[1024];
bool firstMouse = true;
// Light attributes
glm::vec3 lightPos(0.0f, 0.0f, 0.0f);
bool active;

glm::vec3 playerPosition = glm::vec3(0.0f, 0.8f, 0.0f);
glm::vec3 cameraOffset = glm::vec3(0.0f, 4.0f, 12.0f); // Cámara en tercera persona

// Positions of the point lights
```

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 es esencial para iniciar cualquier aplicación gráfica con OpenGL. Sin él, no se puede abrir una ventana ni renderizar gráficos. Prepara todos los recursos que luego serán usados para cargar modelos, manejar entradas y dibujar en pantalla.

Se definen variables para medir el tiempo entre cuadros (frames), útil para movimiento suave e independiente de FPS.

```
108 // Deltatime
109 GLfloat deltaTime = 0.0f; // Time between current frame and last frame
110 GLfloat lastFrame = 0.0f; // Time of last frame
111
```

Inicia la librería GLFW, que permite crear ventanas y manejar entradas (teclado, mouse).

```
112 int main()
113 {
114     // Init GLFW
115     glfwInit();
116 }
```

Crea una ventana de 800x600 píxeles con el título "Fuentes de luz".

```
// Create a GLFWwindow object that we can use for GLFW's functions
GLFWwindow* window = glfwCreateWindow(WIDTH, HEIGHT, "Fuentes de luz", nullptr, nullptr);
```

Verifica si la ventana se creó correctamente. Si falla, se imprime un error y se termina el programa.

```
if (nullptr == window)
{
    std::cout << "Failed to create GLFW window" << std::endl;
    glfwTerminate();
}
return EXIT_FAILURE;
```

Se activa el contexto de OpenGL para esa ventana.

Se obtiene el tamaño real del framebuffer (puede diferir por el DPI).

```
glfwMakeContextCurrent(window);
glfwGetFramebufferSize(window, &SCREEN_WIDTH, &SCREEN_HEIGHT);
```

Asigna funciones para:

Teclado (KeyCallback)

Mouse (MouseCallback)

Estas permiten controlar movimiento de cámara y jugador.

```
// Set the required callback functions
glfwSetKeyCallback(window, KeyCallback);
glfwSetCursorPosCallback(window, MouseCallback);
```

Habilita el uso de funciones modernas de OpenGL y carga los punteros de función de GLEW.

```
// Set this to true so GLEW knows to use a modern approach to retrieving function pointers and extensions
glewExperimental = GL_TRUE;
// Initialize GLEW to setup the OpenGL Function pointers
if (GLEW_OK != glewInit())
{
    std::cout << "Failed to initialize GLEW" << std::endl;
    return EXIT_FAILURE;
}
```

Define el área de la ventana en la que OpenGL va a dibujar (de 0 a ancho/alto).

```
    }
    // Define the viewport dimensions
    glViewport(0, 0, SCREEN_WIDTH, SCREEN_HEIGHT);
```

Crea objetos Shader que cargan y compilan los shaders GLSL desde archivo. Estos se usarán para:

lightingShader: aplicar iluminación Phong a modelos.

lampShader: dibujar las fuentes de luz visibles.

```
// Define the viewport dimensions
glViewport(0, 0, SCREEN_WIDTH, SCREEN_HEIGHT);

Shader lightingShader("Shader/lighting.vs", "Shader/lighting.frag");
Shader lampShader("Shader/lamp.vs", "Shader/Lamp.frag");
```

Se cargan los modelos exportados desde Maya en formato .obj:

Dog: la casa.

personaje: el modelo de Snoopy.

Usan una clase Model (probablemente basada en Assimp) que gestiona mallas, texturas y transformación.

```
// Cargar modelos
Model Dog((char*)"Models/casafinal.obj"); // Modelo de la casa
Model personaje((char*)"Models/snoopy.obj"); //Modelo del personaje
```

Esto prepara un Vertex Array Object (VAO) y un Vertex Buffer Object (VBO):

Se cargan al VBO los datos de vértices del cubo (definidos anteriormente).

Se especifican atributos de los vértices:

Posición (x, y, z)

Normal (nx, ny, nz)

Ambos atributos están intercalados, por eso se usa $6 * \text{sizeof(GLfloat)}$ como stride. Esto permite dibujar el cubo que se usa como lámpara o luz.

```
// First, set the container's VAO (and VBO)
GLuint VBO, VAO;
 glGenVertexArrays(1, &VAO);
 glGenBuffers(1, &VBO);
 glBindVertexArray(VAO);
 glBindBuffer(GL_ARRAY_BUFFER, VBO);
 glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// Position attribute
 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat), (GLvoid*)0);
 glEnableVertexAttribArray(0);
// normal attribute
 glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 * sizeof(float)));
 glEnableVertexAttribArray(1);
```

Aquí se vinculan las texturas difusas y especulares a las unidades de textura 0 y 1 respectivamente. Esto es esencial para aplicar materiales realistas en los modelos.

```
// Set texture units
lightingShader.Use();
glUniform1i(glGetUniformLocation(lightingShader.Program, "Material.diffuse"), 0);
glUniform1i(glGetUniformLocation(lightingShader.Program, "Material.specular"), 1);
```

Se crea la matriz de proyección en perspectiva, que define cómo los objetos se verán en 3D (más lejos = más pequeños).

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

Este bloque de código pertenece al ciclo principal de renderizado (game loop) de un programa en OpenGL. Aquí es donde se realiza el trabajo visual de actualizar, procesar y dibujar la escena 3D en cada cuadro.

Control del tiempo delta time.

Calcula cuánto tiempo ha pasado entre el cuadro actual y el anterior. Esto es esencial para hacer que los movimientos sean independientes del framerate (FPS).

```
// Calculate deltatime of current frame
GLfloat currentTime = glfwGetTime();
deltaTime = currentTime - lastFrame;
lastFrame = currentTime;
```

Procesamiento de entradas.

Se consultan eventos del teclado/mouse.

DoMovement() actualiza la posición del jugador, cámara o luces según lo que se haya presionado.

```
// Check if any events have been activated (key pressed, mouse moved etc.) and call corresponding response functions
glfwPollEvents();
DoMovement();
```

Luz direccional

Define una fuente de luz direccional (como el sol):

ambient: luz ambiental base, ilumina todo de manera uniforme.

diffuse: luz directa que afecta superficies según su orientación.

specular: reflejo especular brillante.

direction: dirección de donde viene la luz.

```
//Load Model

// Use cooresponding shader when setting uniforms/drawing objects
lightingShader.Use();

glUniform1i(glGetUniformLocation(lightingShader.Program, "diffuse"), 0);
glUniform1i(glGetUniformLocation(lightingShader.Program, "specular"), 1);

GLint viewPosLoc = glGetUniformLocation(lightingShader.Program, "viewPos");
glUniform3f(viewPosLoc, cameraGetPosition().x, cameraGetPosition().y, cameraGetPosition().z);

// Directional light

glUniform3f(glGetUniformLocation(lightingShader.Program, "dirLight.ambient"), 0.3f, 0.3f, 0.3f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "dirLight.diffuse"), 0.6f, 0.6f, 0.6f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "dirLight.specular"), 1.0f, 1.0f, 1.0f);

glUniform3f(glGetUniformLocation(lightingShader.Program, "dirLight.ambient"), 0.3f, 0.3f, 0.3f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "dirLight.diffuse"), 0.5f, 0.5f, 0.5f);

glUniform3f(glGetUniformLocation(lightingShader.Program, "dirLight.direction"), -0.2f, -1.0f, -0.3f);
```

Este bloque de código continúa el ciclo principal de renderizado y se enfoca en configurar las fuentes de luz (puntuales y spotlight), establecer propiedades del material, y definir la cámara y transformaciones de vista/proyección.

Configura cuatro luces puntuales, siendo la primera dinámica.

Activa un spotlight controlado por la cámara.

Define cómo reacciona el material a la luz (brillo).

Establece la posición y orientación de la cámara para visualizar la escena desde una perspectiva adecuada.

Prepara todo lo necesario para dibujar los modelos con iluminación realista.

```
// Point Light 1
pointLightPositions[0] = glm::vec3(0.0f, 5.0f, 0.0f);
glm::vec3 lightColor;
lightColor.x = abs(sin(glmfGetTime() + Light1.x));
lightColor.y = abs(sin(glmfGetTime() + Light1.y));
lightColor.z = sin(glmfGetTime() + Light1.z);

glUniform3f(glGetUniformLocation(lightningShader.Program, "pointlights[0].ambient"), 0.2f, 0.2f, 0.2f);
glUniform3f(glGetUniformLocation(lightningShader.Program, "pointlights[0].diffuse"), 0.6f, 0.6f, 0.6f);
glUniform3f(glGetUniformLocation(lightningShader.Program, "pointlights[0].specular"), 1.0f, 1.0f, 1.0f);

glUniform3f(glGetUniformLocation(lightningShader.Program, "pointlights[0].position"), pointLightPositions[0].x, pointLightPositions[0].y, pointLightPositions[0].z);
glUniform3f(glGetUniformLocation(lightningShader.Program, "pointlights[0].ambient"), 0.05f, 0.05f, 0.05f);
glUniform3f(glGetUniformLocation(lightningShader.Program, "pointlights[0].diffuse"), lightColor.x, lightColor.y, lightColor.z);
glUniform3f(glGetUniformLocation(lightningShader.Program, "pointlights[0].specular"), 1.0f, 0.2f, 0.2f);
glUniform1f(glGetUniformLocation(lightningShader.Program, "pointlights[0].constant"), 1.0f);
glUniform1f(glGetUniformLocation(lightningShader.Program, "pointlights[0].linear"), 0.045f);
glUniform1f(glGetUniformLocation(lightningShader.Program, "pointlights[0].quadratic"), 0.075f);

// Point Light 2
glUniform3f(glGetUniformLocation(lightningShader.Program, "pointlights[1].position"), pointLightPositions[1].x, pointLightPositions[1].y, pointLightPositions[1].z);
glUniform3f(glGetUniformLocation(lightningShader.Program, "pointlights[1].ambient"), 0.05f, 0.05f, 0.05f);
glUniform3f(glGetUniformLocation(lightningShader.Program, "pointlights[1].diffuse"), 0.0f, 0.0f, 0.0f);
glUniform3f(glGetUniformLocation(lightningShader.Program, "pointlights[1].specular"), 0.0f, 0.0f, 0.0f);
glUniform1f(glGetUniformLocation(lightningShader.Program, "pointlights[1].constant"), 1.0f);
glUniform1f(glGetUniformLocation(lightningShader.Program, "pointlights[1].linear"), 0.0f);
glUniform1f(glGetUniformLocation(lightningShader.Program, "pointlights[1].quadratic"), 0.0f);

// Point Light 3
glUniform3f(glGetUniformLocation(lightningShader.Program, "pointlights[2].position"), pointLightPositions[2].x, pointLightPositions[2].y, pointLightPositions[2].z);
glUniform3f(glGetUniformLocation(lightningShader.Program, "pointlights[2].ambient"), 0.0f, 0.0f, 0.0f);
glUniform3f(glGetUniformLocation(lightningShader.Program, "pointlights[2].diffuse"), 0.0f, 0.0f, 0.0f);
glUniform3f(glGetUniformLocation(lightningShader.Program, "pointlights[2].specular"), 0.0f, 0.0f, 0.0f);
glUniform1f(glGetUniformLocation(lightningShader.Program, "pointlights[2].constant"), 1.0f);
glUniform1f(glGetUniformLocation(lightningShader.Program, "pointlights[2].linear"), 0.0f);
glUniform1f(glGetUniformLocation(lightningShader.Program, "pointlights[2].quadratic"), 0.0f);

// Point Light 4
glUniform3f(glGetUniformLocation(lightningShader.Program, "pointlights[3].position"), pointLightPositions[3].x, pointLightPositions[3].y, pointLightPositions[3].z);
glUniform3f(glGetUniformLocation(lightningShader.Program, "pointlights[3].ambient"), 0.0f, 0.0f, 0.0f);
glUniform3f(glGetUniformLocation(lightningShader.Program, "pointlights[3].diffuse"), 0.0f, 0.0f, 0.0f);
glUniform3f(glGetUniformLocation(lightningShader.Program, "pointlights[3].specular"), 0.0f, 0.0f, 0.0f);
glUniform1f(glGetUniformLocation(lightningShader.Program, "pointlights[3].constant"), 1.0f);
glUniform1f(glGetUniformLocation(lightningShader.Program, "pointlights[3].linear"), 0.0f);
glUniform1f(glGetUniformLocation(lightningShader.Program, "pointlights[3].quadratic"), 0.0f);

// Spotlight
glUniform3f(glGetUniformLocation(lightningShader.Program, "spotlight.position"), camera.GetPosition().x, camera.GetPosition().y, camera.GetPosition().z);
glUniform3f(glGetUniformLocation(lightningShader.Program, "spotlight.direction"), camera.GetFront().x, camera.GetFront().y, camera.GetFront().z);
glUniform3f(glGetUniformLocation(lightningShader.Program, "spotlight.ambient"), 0.2f, 0.2f, 0.8f);
glUniform3f(glGetUniformLocation(lightningShader.Program, "spotlight.diffuse"), 0.2f, 0.2f, 0.8f);
glUniform3f(glGetUniformLocation(lightningShader.Program, "spotlight.specular"), 0.0f, 0.0f, 0.0f);
glUniform1f(glGetUniformLocation(lightningShader.Program, "spotlight.constant"), 1.0f);
glUniform1f(glGetUniformLocation(lightningShader.Program, "spotlight.linear"), 0.3f);
```

```

// SpotLight
glUniform3f(glGetUniformLocation(lightingShader.Program, "spotlight_position"), camera.GetPosition().x, camera.GetPosition().y, camera.GetPosition().z);
glUniform3f(glGetUniformLocation(lightingShader.Program, "spotlight_direction"), camera.GetFront().x, camera.GetFront().y, camera.GetFront().z);
glUniform3f(glGetUniformLocation(lightingShader.Program, "spotlight_ambient"), 0.2f, 0.2f, 0.8f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "spotlight_diffuse"), 0.2f, 0.2f, 0.8f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "spotlight_specular"), 0.0f, 0.0f, 0.0f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "spotlight_constant"), 1.0f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "spotlight_linear"), 0.3f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "spotlight_quadratic"), 0.7f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "spotlight_cutfOff"), glm::cos(glm::radians(12.0f)));
glUniform1f(glGetUniformLocation(lightingShader.Program, "spotlight_outerCutfOff"), glm::cos(glm::radians(18.0f)));

// Set material properties
glUniform1f(glGetUniformLocation(lightingShader.Program, "material.shininess"), 16.0f);

// Create camera transformations
glm::mat4 view;
view = camera.GetViewMatrix();

// Get the uniform locations
GLint modelLoc = glGetUniformLocation(lightingShader.Program, "model");
GLint viewLoc = glGetUniformLocation(lightingShader.Program, "view");
GLint projLoc = glGetUniformLocation(lightingShader.Program, "projection");

// Pass the matrices to the shader
glUniformMatrix4fv(viewLoc, 1, GL_FALSE, glm::value_ptr(view));
glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(projection));

glm::mat4 model(1);

//Carga de modelo
//view = camera.GetViewMatrix();/*
glm::vec3 newCamPos = playerPosition + cameraOffset;
glm::vec3 newFront = glm::normalize(playerPosition - newCamPos);

view = glm::lookAt(newCamPos, playerPosition, glm::vec3(0.0f, 1.0f, 0.0f));
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));

```

Este bloque de código se encarga de dibujar los modelos principales de la escena (la casa y el personaje), y también renderizar visualmente las luces puntuales como pequeños cubos usando un shader especial (lampShader). Vamos por partes:

Este bloque de código en OpenGL se encarga de dibujar dos modelos principales en la escena 3D: la casa y el personaje, aplicando transformaciones y habilitando el canal alfa para manejar transparencias si es necesario.

```

model = glm::mat4(1);
	glEnable(GL_BLEND); //Aactiva la funcionalidad para trabajar el canal alfa
	glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
	glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
	glUniform1i(glGetUniformLocation(lightingShader.Program, "transparency"), 0);
Dog.Draw(lightingShader);

model = glm::mat4(1.0f);
model = glm::translate(model, playerPosition);
model = glm::scale(model, glm::vec3(0.7f)); // Ajusta el tamaño del modelo

	glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
personaje.Draw(lightingShader);

glDisable(GL_BLEND); //Desactiva el canal alfa
 glBindVertexArray(0);

```

Este bloque de código se encarga de dibujar visualmente las luces puntuales como cubos pequeños dentro de la escena 3D, usando un shader más simple llamado lampShader. No afectan la iluminación (esa ya está definida en otro shader), pero sí permiten que el usuario vea físicamente dónde están colocadas las luces.

```

// Also draw the lamp object, again binding the appropriate shader
LampShader.Use();
// Get location objects for the matrices on the lamp shader (these could be different on a different shader)
modelLoc = glGetUniformLocation(LampShader.Program, "model");
viewLoc = glGetUniformLocation(LampShader.Program, "view");
projLoc = glGetUniformLocation(LampShader.Program, "projection");

// Set matrices
glUniformMatrix4fv(viewLoc, 1, GL_FALSE, glm::value_ptr(view));
glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(projection));
model = glm::mat4(1);
model = glm::translate(model, lightPos);
model = glm::scale(model, glm::vec3(0.2f)); // Make it a smaller cube
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
// Draw the light object (using light's vertex attributes)
for (GLuint i = 0; i < 4; i++)
{
    model = glm::mat4(1);
    model = glm::translate(model, pointLightPositions[i]);
    model = glm::scale(model, glm::vec3(0.2f)); // Make it a smaller cube
    glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
    glBindVertexArray(VAO);
    glDrawArrays(GL_TRIANGLES, 0, 36);
}
glBindVertexArray(0);

```

Este bloque de código define la función DoMovement(), que se ejecuta en cada fotograma para actualizar la posición del personaje y de una fuente de luz en función de las teclas que el usuario presione. Este bloque permite mover al personaje (representado por playerPosition) en las direcciones:

Z hacia atrás (S o flecha ↑): playerPosition.z -= speed;

Z hacia adelante (W o flecha ↓): playerPosition.z += speed;

X hacia la izquierda (D o flecha ←): playerPosition.x -= speed;

X hacia la derecha (A o flecha →): playerPosition.x += speed;

El personaje se mueve en el plano XZ, como si caminara sobre el piso.

Estas teclas mueven la primera luz puntual (pointLightPositions[0]):

X con T y G

Y con F y H

Z con U y J

Esto te permite repositionar dinámicamente una luz en la escena, lo cual es útil para experimentar con efectos de iluminación.

```

// Moves/alters the camera positions based on user input
void DoMovement()
{
    float speed = 20.0f * deltaTime;

    // Movimiento del personaje
    if (keys[GLFW_KEY_S] || keys[GLFW_KEY_UP])
        playerPosition.z -= speed;
    if (keys[GLFW_KEY_W] || keys[GLFW_KEY_DOWN])
        playerPosition.z += speed;
    if (keys[GLFW_KEY_D] || keys[GLFW_KEY_LEFT])
        playerPosition.x -= speed;
    if (keys[GLFW_KEY_A] || keys[GLFW_KEY_RIGHT])
        playerPosition.x += speed;

    // Movimiento de la luz
    if (keys[GLFW_KEY_T])
        pointLightPositions[0].x += 0.01f;
    if (keys[GLFW_KEY_G])
        pointLightPositions[0].x -= 0.01f;
    if (keys[GLFW_KEY_F])
        pointLightPositions[0].y += 0.01f;
    if (keys[GLFW_KEY_H])
        pointLightPositions[0].y -= 0.01f;
    if (keys[GLFW_KEY_U])
        pointLightPositions[0].z -= 0.1f;
    if (keys[GLFW_KEY_J])
        pointLightPositions[0].z += 0.01f;
}

```

Este bloque de código define la función KeyCallback(...), que es invocada automáticamente por GLFW cada vez que el usuario presiona o suelta una tecla. Su propósito es gestionar eventos de teclado y actualizar el comportamiento del programa en consecuencia.

```

// Is called whenever a key is pressed/released via GLFW
void KeyCallback(GLFWwindow* window, int key, int scanCode, int action, int mode)
{
    if (GLFW_KEY_ESCAPE == key && GLFW_PRESS == action)
    {
        glfwSetWindowShouldClose(window, GL_TRUE);
    }

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

    if (keys[GLFW_KEY_SPACE])
    {
        active = !active;
        if (active)
        {
            Light1 = glm::vec3(1.0f, 1.0f, 0.0f);
        }
        else
        {
            Light1 = glm::vec3(0); //Cuando es solo un valor en los 3 vectores pueden dejar solo una componente
        }
    }
}

```

Cierra la ventana con ESC.

Detecta y guarda el estado de teclas presionadas.

Cuando una tecla es presionada (GLFW_PRESS), se marca como true en el arreglo keys[].

Cuando se suelta (GLFW_RELEASE), se marca como false.

Esto permite que otras funciones, como DoMovement(), verifiquen continuamente si una tecla está presionada.

Altera la luz con la barra espaciadora (SPACE)

Cuando se presiona SPACE, se alterna la variable booleana active:

Si está activa: se asigna el color amarillo intenso a Light1 (1.0f, 1.0f, 0.0f).

Si está inactiva: se apaga la luz con un color negro (0, 0, 0).

Esto sirve para encender/apagar una luz puntual del entorno.

```
// Is called whenever a key is pressed/released via GLFW
void KeyCallback(GLFWwindow* window, int key, int scancode, int action, int mode)
{
    if (GLFW_KEY_ESCAPE == key && GLFW_PRESS == action)
    {
        glfwSetWindowShouldClose(window, GL_TRUE);
    }

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

    if (keys[GLFW_KEY_SPACE])
    {
        active = !active;
        if (active)
        {
            Light1 = glm::vec3(1.0f, 1.0f, 0.0f);
        }
        else
        {
            Light1 = glm::vec3(0); //Cuando es solo un valor en los 3 vectores pueden dejar solo una componente
        }
    }
}
```

Este bloque de código define la función MouseCallback(...), que se ejecuta automáticamente cada vez que el usuario mueve el mouse dentro de la ventana GLFW. Su objetivo es controlar la orientación de la cámara en tercera persona, orbitando alrededor del personaje.

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

    float xOffset = xPos - lastX;
    float yOffset = lastY - yPos; // Invertido: mueve hacia arriba cuando subes el mouse

    lastX = xPos;
    lastY = yPos;

    float sensitivity = 0.1f;
    xOffset *= sensitivity;
    yOffset *= sensitivity;

    // Ángulos de rotación
    static float yaw = -90.0f;
    static float pitch = 10.0f;

    yaw += xOffset;
    pitch += yOffset;

    // Limitar pitch para que la cámara no se mete debajo
    if (pitch > 89.0f) pitch = 89.0f;
    if (pitch < 5.0f) pitch = 5.0f; // Ajusta para evitar cámara demasiado baja

    // Cálculo de posición orbital de la cámara alrededor del personaje
    float radius = 6.0f;
    float camX = radius * cos(glm::radians(pitch)) * cos(glm::radians(yaw));
    float camY = radius * sin(glm::radians(pitch));
    float camZ = radius * cos(glm::radians(pitch)) * sin(glm::radians(yaw));

    // Opcional: asegurar que no baje demasiado
    if ((playerPosition.y + camY) < 1.0f)
        camY = 1.0f - playerPosition.y;

    cameraOffset = glm::vec3(camX, camY, camZ);
```

Shader.h:

Este archivo define la clase Shader, cuya función principal es compilar, vincular y gestionar shaders en OpenGL. Permite cargar shaders desde archivos externos (.vs y .frag), compilar el código fuente GLSL y activar el programa resultante para ser usado en el pipeline de renderizado.

Funcionamiento del constructor

1. Carga de archivos:
 - Usa std::ifstream para abrir los archivos del vertex shader y fragment shader.
 - Lee su contenido y lo convierte en cadenas de texto.
2. Compilación:
 - Crea objetos shader (glCreateShader) para el vertex y fragment.
 - Usa glShaderSource y glCompileShader para compilar cada uno.
 - Verifica errores de compilación y los muestra en consola si existen.
3. Vinculación del programa:
 - Crea un programa de shader con glCreateProgram, adjunta los shaders compilados, y los une con glLinkProgram.
 - Comprueba si hubo errores en el proceso de enlace.
4. Limpieza:
 - Elimina los objetos shader temporales (glDeleteShader) tras su vinculación.
5. Uniform color:
 - Obtiene la localización del uniform "color" en el shader, almacenándola en uniformColor, lo cual permite modificar dinámicamente el color desde el programa en C++.

Funciones principales

- **Use():**
Activa el shader program para que OpenGL lo utilice en las siguientes instrucciones de dibujo.
- **getColorLocation():**
Devuelve la localización del uniform llamado "color" dentro del shader activo.

```

shader.h: #ifndef SHADER_H
#define SHADER_H

#include <string>
#include <fstream>
#include <iostream>
#include <GL/glew.h>

class Shader
{
public:
    GLuint Program;
    GLuint uniformColor;
    // Constructor generates the shader on the fly
    Shader(const GLchar *vertexPath, const GLchar *fragmentPath)
    {
        // 1. Retrieve the vertex/fragment source code from filePath
        std::string vertexCode;
        std::string fragmentCode;
        std::ifstream vShaderfile;
        std::ifstream fShaderfile;
        // Ensure ifstream objects can throw exceptions:
        vShaderfile.exceptions(std::ios::badbit);
        fShaderfile.exceptions(std::ios::badbit);
        try
        {
            // Open files
            vShaderfile.open(vertexPath);
            fShaderfile.open(fragmentPath);
            std::stringstream vShaderStream, fShaderStream;
            // Read file's buffer contents into streams
            vShaderStream <> vShaderfile.rdbuf();
            fShaderStream <> fShaderfile.rdbuf();
            // Close file handles
            vShaderfile.close();
            fShaderfile.close();
            // Convert stream into string
            vertexCode = vShaderStream.str();
            fragmentCode = fShaderStream.str();
        }
        catch (std::ifstream::failure e)
        {
            std::cout << "ERROR::SHADER::FILE_NOT_SUCCESFULLY_READ" << std::endl;
        }
        const GLchar *vShaderCode = vertexCode.c_str();
        const GLchar *fShaderCode = fragmentCode.c_str();
        // 2. Compile shaders
        GLuint vertex, fragment;
        GLint success;
        std::string infoLog[512];
        // Vertex Shader
        vertex = glCreateShader(GL_VERTEX_SHADER);
        glShaderSource(vertex, 1, &vShaderCode, NULL);
        glCompileShader(vertex);
        // Print compile errors if any
        glGetShaderiv(vertex, GL_COMPILE_STATUS, &success);
        if (!success)
        {
            glGetShaderInfoLog(vertex, 512, NULL, infoLog);
            std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog << std::endl;
        }
        // Fragment Shader
        fragment = glCreateShader(GL_FRAGMENT_SHADER);
        glShaderSource(fragment, 1, &fShaderCode, NULL);
        glCompileShader(fragment);
        // Print compile errors if any
        glGetShaderiv(fragment, GL_COMPILE_STATUS, &success);
        if (!success)
        {
            glGetShaderInfoLog(fragment, 512, NULL, infoLog);
            std::cout << "ERROR::SHADER::FRAGMENT::COMPILATION_FAILED\n" << infoLog << std::endl;
        }
        std::cout << "ERROR::SHADER::PROGRAM::COMPILATION_FAILED\n" << infoLog << std::endl;
        // Shader Program
        this->Program = glCreateProgram();
        glAttachShader(this->Program, vertex);
        glAttachShader(this->Program, fragment);
        glLinkProgram(this->Program);
        // Print linking errors if any
        glGetProgramiv(this->Program, GL_LINK_STATUS, &success);
        if (!success)
        {
            glGetProgramInfoLog(this->Program, 512, NULL, infoLog);
            std::cout << "ERROR::SHADER::PROGRAM::LINKING_FAILED\n" << infoLog << std::endl;
        }
        // Le damos la localidad de color
        uniformColor = glGetUniformLocation(this->Program, "color");
        // Delete the shaders as they're linked into our program now and no longer necessary
        glDeleteShader(vertex);
        glDeleteShader(fragment);
    }
    // Uses the current shader
    void Use()
    {
        glUseProgram(this->Program);
    }
    GLuint getColorLocation()
    {
        return uniformColor;
    }
}

```

Camara.h:

Este archivo define una clase Camera que simula una cámara en un entorno 3D, permitiendo moverse en el espacio, rotar la vista y generar matrices de vista (view matrix) para su uso en OpenGL. Es esencial para lograr una experiencia de navegación libre o en tercera persona dentro de la escena 3D.

Características principales

Componente

glm::vec3 position

glm::vec3 front

glm::vec3 up, right, worldUp

GLfloat yaw, pitch

GLfloat movementSpeed, mouseSensitivity, zoom

Descripción

Posición actual de la cámara en el mundo 3D.

Dirección hacia la que está mirando la cámara.

Vectores ortogonales que definen la orientación de la cámara.

Ángulos de rotación que determinan la dirección de la vista.

Parámetros que controlan la velocidad de movimiento, la sensibilidad del mouse y el campo de visión.

Funciones públicas importantes

Función	Descripción
<i>Camera(...)</i>	Constructor que inicializa la cámara con valores por defecto o definidos.
<i>GetViewMatrix()</i>	Retorna la matriz de vista (<code>glm::lookAt</code>) que posiciona la cámara para renderizar la escena desde su punto de vista.
<i>ProcessKeyboard(...)</i>	Recibe una dirección (adelante, atrás, izquierda, derecha) y actualiza la posición según la velocidad.
<i>ProcessMouseMovement(...)</i>	Ajusta la dirección de la cámara (yaw/pitch) en función del movimiento del mouse. Incluye restricción de pitch para evitar giros irreales.
<i>GetZoom()</i> , <i>GetPosition()</i> , <i>GetFront()</i>	Devuelven parámetros clave de la cámara: campo de visión, posición actual y dirección de la vista.

Lógica interna destacada

- Sistema de Ejes:
 - Calcula el vector front usando yaw y pitch con trigonometría.
 - Luego, calcula los vectores right y up usando el producto cruzado, para mantener una orientación ortogonal correcta.
- Movimiento fluido:
 - Usa deltaTime para ajustar la velocidad según el tiempo entre fotogramas, garantizando movimiento constante en cualquier hardware.
- Independencia del sistema de entrada:
 - Utiliza un enum Camera_Movement para definir los movimientos, permitiendo que el manejo del teclado se mantenga desacoplado del backend de entrada.

```

camera.h: #pragma once

// Std. Includes
#include <vector>

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

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

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

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

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

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

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

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

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

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

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

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

        this->yaw += xoffset;
        this->pitch += yoffset;

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

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

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

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

```

```

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

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

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

    ...

private:
    // Camera Attributes
    glm::vec3 position;
    glm::vec3 front;
    glm::vec3 up;
    glm::vec3 right;
    glm::vec3 worldUp;
    ...

    // Euler Angles
    GLfloat yaw;
    GLfloat pitch;
    GLfloat zoom;

    // Camera options
    GLfloat movementSpeed;
    GLfloat mouseSensitivity;
    GLfloat zoom;

    // Calculates the front vector from the Camera's (updated) Euler Angles
    void updateCameraVectors()
    {
        // Calculate the new Front vector
        glm::vec3 front;
        front.x = cos(glm::radians(this->yaw)) * cos(glm::radians(this->pitch));
        front.y = sin(glm::radians(this->pitch));
        front.z = sin(glm::radians(this->yaw)) * cos(glm::radians(this->pitch));
        this->front = glm::normalize(front);
        // Also re-calculate the Right and Up vector
        this->right = glm::normalize(glm::cross(this->front, this->worldUp)); // Normalize the vectors, because their length gets closer to 0 the more you look up or down which results in slower movement.
        this->up = glm::normalize(glm::cross(this->right, this->front));
    }
    ...
};

```

Core.frag:

Este fragment shader permite que los objetos 3D importados (como la casa, muebles o Snoopy) aparezcan con texturas realistas, en lugar de colores planos. Esencial para lograr la apariencia visual final del proyecto, utilizando materiales con imágenes aplicadas a sus superficies.

Explicación línea por línea

Línea	Descripción
#version 330 core	Define la versión de GLSL usada (OpenGL 3.3 Core Profile).
in vec2 TexCoords;	Coordenadas UV interpoladas por vértice desde el vertex shader. Indican qué parte de la textura se usará en el fragmento.
out vec4 color;	Color final que se renderizará en la pantalla. Se asigna al framebuffer.
uniform sampler2D texture_diffuse;	Texto 2D (por ejemplo, una imagen .jpg o .png) pasada desde C++ como uniform.
vec4 texColor = texture(...);	Recupera el color de la textura según las coordenadas TexCoords.
color = texColor;	Se asigna directamente el color de la textura al fragmento.

```

#version 330 core

in vec2 TexCoords;

out vec4 color;

uniform sampler2D texture_diffuse;

void main()
{
    vec4 texColor = texture(texture_diffuse, TexCoords);
    color = texColor;
}

```

Core.vs:

Este vertex shader es crucial porque:

Transforma las posiciones de los vértices desde coordenadas de modelo a espacio de clip, para que OpenGL pueda renderizar correctamente cada objeto.

Envía un color uniforme a cada fragmento, permitiendo aplicar efectos visuales simples como cambiar el color de un objeto sin texturizarlo.

Explicación línea por línea

Línea	Descripción
<code>#version 330 core</code>	Define que se usa GLSL versión 3.3 (OpenGL moderno, core profile).
<code>layout (location = 0) in vec3 position;</code>	Recibe la posición del vértice como entrada, desde el VAO.
<code>out vec3 ourColor;</code>	Salida hacia el fragment shader: el color asociado a este vértice.
<code>uniform mat4 model;</code>	Matriz de transformación que ubica el objeto en el mundo.
<code>uniform mat4 view;</code>	Matriz de cámara: define la vista del observador.
<code>uniform mat4 projection;</code>	Matriz de proyección: define el tipo de proyección (perspectiva o ortogonal).
<code>uniform mat4 transform;</code>	Matriz opcional (no usada en el cálculo final en este caso).
<code>uniform vec3 color;</code>	Color uniforme para el objeto (asignado desde C++).
<code>gl_Position = projection * view * model * vec4(position, 1.0f);</code>	Calcula la posición final del vértice transformado para renderizarlo en pantalla.
<code>ourColor = color;</code>	Pasa el color al fragment shader para que lo use en el renderizado.

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

out vec3 ourColor;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
uniform mat4 transform;
uniform vec3 color;

void main()
{
    gl_Position = projection*view*model*vec4(position, 1.0f);
    ourColor = color;
}
```

Lamp.frag:

Este shader se usa para dibujar objetos que simbolizan las lámparas o fuentes de luz en la escena. Aunque las luces en OpenGL son conceptuales (no se ven directamente), este fragment shader permite que el usuario visualice en pantalla el punto donde se encuentra la fuente de iluminación, ayudando en el diseño y prueba de efectos lumínicos.

Explicación línea por línea

Línea	Descripción
<code>#version 330 core</code>	Define que se utiliza GLSL versión 3.3 (modo core profile).
<code>out vec4 FragColor;</code>	Variable de salida que representa el color final del píxel.

FragColor = Asigna un color blanco puro con opacidad total (*vec4(1.0f)* equivale a (1.0, 1.0, 1.0, 1.0) en RGBA).

```
#version 330 core
out vec4 FragColor;

void main()
{
    FragColor = vec4(1.0f);
}
```

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.

Explicación línea por línea

Línea	Descripción
<i>#version 330 core</i>	Se usa GLSL 3.3 (modo core profile).
<i>layout (location = 0) in vec3 position;</i>	Entrada del vértice: posición 3D proveniente del VAO.
<i>uniform mat4 model, view, projection;</i>	Matrices necesarias para transformar los vértices desde el modelo hasta el espacio de clip.
<i>gl_Position = projection * view * model * vec4(position, 1.0f);</i>	Cálculo de la posición final del vértice transformado, lista para el rasterizado.

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

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

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

Lighting.frag:

Este shader permite que los objetos modelados en Maya se vean iluminados de forma dinámica y realista en la escena interactiva, integrando múltiples fuentes de luz con efectos visuales avanzados. Juega un rol clave en la ambientación, inmersión y estética general del entorno 3D de la casa de Charlie Brown.

Estructuras definidas

Estructura	Contenido principal	Función
Material	sampler2D diffuse, specular, float shininess	Propiedades del material texturizado

<i>DirLight</i>	direction, ambient, diffuse, specular	Luz direccional, como el sol
<i>PointLight</i>	position, constant, linear, quadratic, ambient, diffuse, specular	Luz puntual con atenuación
<i>SpotLight</i>	position, direction, cutOff, outerCutOff, más atenuación y colores	Luz tipo linterna o foco

Entradas (Inputs GLSL)

Nombre	Tipo	Descripción
<i>FragPos</i>	vec3	Posición del fragmento en el espacio mundial
<i>Normal</i>	vec3	Vector normal del fragmento
<i>TexCoords</i>	vec2	Coordenadas UV para mapear texturas
<i>viewPos</i>	vec3 (uniform)	Posición de la cámara/observador
<i>material</i>	Material	Propiedades del material del objeto
<i>dirLight</i>	DirLight	Fuente de luz direccional
<i>pointLights[]</i>	PointLight[4]	Arreglo de hasta 4 luces puntuales
<i>spotLight</i>	SpotLight	Luz tipo foco dirigida desde la cámara
<i>transparency</i>	int	Flag para habilitar descarte de fragmentos con alfa bajo

Funcionamiento principal (main)

1. **Normalización de vectores:** se preparan el normal y el vector de vista (viewDir).
2. **Iluminación base:**
 - Se calcula el efecto de la **luz direccional** con CalcDirLight().
3. **Luces puntuales:**
 - Se acumulan las contribuciones de cada una usando un for que llama a CalcPointLight().
4. **Spotlight:**
 - Se añade el efecto de foco con CalcSpotLight().
5. **Transparencia:**
 - Si color.a < 0.1 y transparency == 1, el fragmento se **descarta** y no se renderiza.

Funciones auxiliares

Función	Uso
CalcDirLight(...)	Iluminación con luz paralela
CalcPointLight(...)	Iluminación con atenuación en base a distancia
CalcSpotLight(...)	Iluminación con ángulo de corte y suavizado
Cada una de estas funciones sigue el modelo Phong (ambient + diffuse + specular), aplicando texturas y parámetros físicos realistas	

```

#version 330 core
#define NUMBER_OF_POINT_LIGHTS 4

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

struct DirLight
{
    vec3 direction;
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};

struct PointLight
{
    vec3 position;
    float constant;
    float linear;
    float quadratic;
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};

struct SpotLight
{
    vec3 position;
    vec3 direction;
    float cutOff;
    float outerCutOff;
    float constant;
    float linear;
    float quadratic;
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};

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

out vec4 color;

uniform vec3 viewPos;
uniform DirLight dirLight;
uniform PointLight pointLights[NUMBER_OF_POINT_LIGHTS];
uniform SpotLight spotLight;

```

```

uniform PointLight pointLights[NUMBER_OF_POINT_LIGHTS];
uniform SpotLight spotLight;
uniform Material material;
uniform int transparency;

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

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

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

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

    // Spot light
    result += CalcSpotLight( spotLight, norm, FragPos, viewDir );
    color = vec4(result, 1.0);

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

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

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

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

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

    return ( ambient + diffuse + specular );
}

// Calculates the color when using a point light.
vec3 CalcPointLight( PointLight light, vec3 normal, vec3 fragPos, vec3 viewDir )
{
    vec3 lightDir = normalize( light.position - fragPos );
    float distance = length( light.position - fragPos );
    float attenuation = 1.0f / ( light.constant + light.linear * distance + light.quadratic * ( distance * distance ) );

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

    return ( ambient + diffuse + specular );
}

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

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

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

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

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

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

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

```

Lighting.vs:

Este shader prepara toda la información necesaria para que lighting.frag pueda calcular correctamente los efectos de iluminación (Phong) y texturizado sobre los modelos. Su correcto funcionamiento garantiza que la iluminación se aplique con precisión, y que las texturas se vean correctamente alineadas con los modelos 3D importados desde Maya.

Es especialmente importante en tu proyecto porque permite renderizar objetos como Snoopy, la casa, o muebles, con iluminación avanzada, materiales realistas y texturas detalladas.

Explicación línea por línea

Línea

```

layout (location = 0) in vec3 position;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec2 texCoords;
out vec3 Normal;
out vec3 FragPos;
out vec2 TexCoords;
uniform mat4 model, view, projection;

```

gl_Position = ...

```

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

```

Descripción

Posición del vértice (x, y, z), entrada desde el VAO.
Vector normal del vértice, utilizado para iluminación.
Coordenadas UV para aplicar textura.
Normal transformada que se enviará al fragment shader.
Posición del fragmento en el espacio mundial.
Coordenadas UV que serán usadas en lighting.frag.
Matrices de transformación necesarias para ubicar el objeto en pantalla.
Calcula la posición final del vértice en espacio de clip.
Guarda la posición del vértice en el espacio global para cálculo de iluminación.

Normal = mat3(transpose(inverse(model)))

** normal;*

TexCoords = texCoords;

Transforma la normal correctamente en caso de que el modelo haya sido escalado no uniformemente.

Pasa las coordenadas UV al fragment shader.

```

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

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

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

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

```

Modelloading.frag:

Este shader es crucial para representar modelos que usan texturas con transparencia, como objetos con huecos o bordes irregulares (telones, follaje, cristales, etc.). Además, mejora el realismo visual y evita renderizar fragmentos que no deberían mostrarse, optimizando el renderizado.

Se aplica típicamente a modelos importados mediante Assimp que traen sus propias texturas (texture_diffuse1).

Explicación por secciones

Línea	Descripción
#version 330 core	Especifica el uso de GLSL 3.3 en perfil core.
in vec2 TexCoords;	Coordenadas UV interpoladas para mapear la textura sobre el modelo.
uniform sampler2D texture_diffuse1;	Uniforme que representa la textura difusa cargada desde C++ y asignada al modelo.
vec4 texColor = texture(...);	Se obtiene el color del píxel correspondiente de la textura.
if(texColor.a < 0.1) discard;	Si el valor de alfa (transparencia) es muy bajo, se descarta el fragmento, haciéndolo invisible.
FragColor = texColor;	Se asigna el color final del fragmento, incluyendo canal alfa.

```

#version 330 core
out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D texture_diffuse1;

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

```

Modelloading.vs:

Este shader es clave para permitir que los modelos importados (como los de Maya en formato .obj) sean renderizados correctamente con sus texturas difusas. Es parte esencial del sistema de materiales y texturizado en tu escena interactiva.

En combinación con modelloading.frag, permite renderizar con precisión modelos complejos que tienen materiales personalizados y detalles visuales importantes para la estética del entorno.

Explicación por partes

Sección	Descripción
<code>layout (location = 0) in vec3 aPos;</code>	Recibe la posición del vértice desde el buffer de vértices.
<code>layout (location = 1) in vec3 aNormal;</code>	Entrada del vector normal, no utilizado aquí pero preparado para iluminación.
<code>layout (location = 2) in vec2 aTexCoords;</code>	Recibe las coordenadas UV del vértice.
<code>out vec2 TexCoords;</code>	Transmite las UV al fragment shader para mapear texturas.
<code>uniform mat4 model, view, projection;</code>	Matrices estándar para transformar el vértice desde espacio local a pantalla.
<code>gl_Position = ...</code>	Calcula la posición final del vértice en el sistema de coordenadas de clip.
<code>TexCoords = aTexCoords;</code>	Transfiere las coordenadas UV al siguiente paso del pipeline.

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

out vec2 TexCoords;

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

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

CONCLUSIÓN

A lo largo del desarrollo de este proyecto pude aplicar y fortalecer mis habilidades tanto en el modelado 3D como en la lógica de programación gráfica. Mi enfoque principal estuvo en la construcción y texturizado de los modelos dentro de Maya, donde aprendí a aprovechar herramientas como extrude, scale y simulaciones como nCloth para dar realismo a objetos como la cama, la sábana o la tetera. Fue especialmente enriquecedor ver cómo objetos creados desde primitivas simples podían cobrar vida con materiales adecuados y una correcta organización de mallas.

Además, al integrarlos dentro del entorno interactivo usando OpenGL, comprendí de forma práctica cómo funcionan aspectos fundamentales como los shaders, las transformaciones en coordenadas 3D, la configuración de materiales y los distintos tipos de luces. La implementación del sistema de cámara en tercera persona y la

iluminación dinámica me permitió entender a profundidad el impacto que tiene la interacción entre código y modelo visual.

Uno de los mayores desafíos surgió en la etapa de animación, donde descubrí la crítica importancia de los pivotes en cada modelo. Una colocación incorrecta del pivote puede generar rotaciones incoherentes o desplazamientos no deseados, afectando la lógica del movimiento programado. Gracias a una correcta configuración previa en Maya, logré controlar con precisión las transformaciones en OpenGL, lo cual fue esencial para animaciones como la apertura de puertas, caída de objetos o el giro de manecillas. Esta experiencia me enseñó que la animación en tiempo real no solo depende del código, sino también de una planificación precisa desde la etapa de modelado.

Este proyecto me permitió no solo desarrollar competencias técnicas específicas, sino también reforzar la importancia del trabajo colaborativo y la organización por etapas. Me llevo una experiencia completa del flujo de trabajo que va desde la creación visual hasta la programación funcional de una escena interactiva.

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/>
- Autodesk. (n.d.). Maya - 3D Modeling and Animation Software. Recuperado de <https://www.autodesk.com/products/maya/overview>
- 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>