

MANUAL **TECNICO**

Remodelación de
Laboratorio de
Computación Gráfica



Equipo 11

Introducción.....	4
Objetivo del proyecto.....	5
Público objetivo.....	6
Cronograma de actividades.....	7
Requerimientos de hardware y software.....	8
Instalación de repositorio y control de versiones.....	10
Prerrequisitos.....	10
Clonación del repositorio con líneas de comando.....	10
Clonación del repositorio con GitHub Desktop.....	11
Verificar la estructura de las carpetas.....	12
Contribuir y reportar issues.....	12
Herramientas de modelado y texturizado.....	14
Blender.....	14
Modelos para salón actual.....	15
Arquitectura general código fuente.....	24
Estructura de archivos y módulos.....	24
Puntos clave de código fuente.....	25
Módulos clave.....	27
Shader lighting.frag.....	27
Uniforms y estructuras.....	28
Función main.....	30
Cálculos de iluminación.....	30
Shader lighting.vs.....	32
Variables globales.....	34
Variables para animación del monitor.....	34
Variables de animación específica de sillas "si" y "sn".....	34
Prototipos de callbacks y funciones de actualización.....	34
Configuración de ventana y cámara.....	35
Luces y animaciones complementarias.....	35
Límites y control de animación monitores.....	35
Visibilidad y estado de modelos individuales para gabinete.....	35
Actualización global del laboratorio.....	37
Variables de iluminación.....	38
Carga y definición de modelo en OpenGL.....	38
Dibujar modelos en el ciclo de renderizado.....	38
Iluminación.....	40
Módulos extra.....	42

Función que evita colisiones.....	42
Animaciones complejas.....	43
Estructura e instancia de animación.....	43
Animación de monitores.....	44
Animación de armado PC.....	48
Keyframe y animación de sillas.....	54
Keyframe para animación profesor.....	61
Animación de logos en pantalla.....	66
Animaciones adicionales.....	71
Manejo de entrada de teclado.....	75
Espacios estratégicos para patrocinadores.....	80
Gestión de recursos y limpieza.....	81
Liberación de objetos de OpenGL.....	81
Desconexión de callbacks y destrucción de la ventana.....	82
Consideraciones adicionales.....	82
Herramientas de comunicación y trabajo.....	83
Fin de manual técnico.....	84
Soporte y Contacto.....	84
ENLACE DE REPOSITORIO PARA CONSULTA DETALLADA.....	85
REFERENCIAS SIGNIFICATIVAS.....	85

Introducción.

Este Manual Técnico documenta en detalle la arquitectura interna, los componentes y las estructuras de datos del proyecto “Innovación de laboratorio de computación gráfica”, desarrollado por el Equipo 11. Su propósito es servir de referencia completa para:

- Desarrolladores y contribuidores, que necesitan entender el diseño modular del código, las dependencias externas y el flujo de ejecución para mantener o ampliar la aplicación.
- Estudiantes de ingeniería, interesados en profundizar en la implementación de gráficos por computadora con OpenGL 3.3+, y en las técnicas de animación basadas en keyframes e interpolación.
- Profesores y evaluadores, que desean revisar los fundamentos técnicos, la organización de archivos y el uso de librerías clave como GLEW, GLFW, GLM, Assimp y SOIL2.

A lo largo de este documento encontrarás:

1. Objetivo y alcance, definiendo el propósito del proyecto “Campus virtual interactivo” y los límites de lo que cubre este manual.
2. Requerimientos de hardware y software, para garantizar que el entorno cumpla con las especificaciones mínimas y recomendadas.
3. Control de versiones, con instrucciones para clonar el repositorio y contribuir mediante GitHub.
4. Herramientas de modelado, explicando cómo se integran modelos creados en Blender y los generados para demás modelos de componentes del salón, así como su incorporación al código fuente.
5. Arquitectura general del código, mostrando la estructura de carpetas (**include/**, **src/**, **Shader/**, **Models/**), los módulos principales y las dependencias externas .
6. Puntos clave de implementación, entre los que destacan:
 - La configuración de OpenGL y GLFW para inicializar el contexto gráfico.
 - La carga dinámica de modelos 3D mediante la clase **Model** (basada en Assimp).
 - La definición y compilación de shaders (**lighting.vs**, **lighting.frag**) que implementan iluminación direccional, puntual y de foco.

- El bucle de renderizado, que orquesta la captura de entradas, la actualización de animaciones (**Animation()**), el cálculo de **deltaTime** y el envío de uniforms (matrices de transformación, posiciones de luz, explosionFactor, etc.) al GPU.
- El manejo de animaciones de monitores, CPUs y sillas mediante máquinas de estado y **glm::mix**, garantizando transiciones suaves y reversibles .

7. Módulos y funciones detalladas, donde se desglosan:

- Las estructuras **DirLight**, **PointLight** y **SpotLight** y su uso en el shader de fragmentos.
 - La lógica dentro de **Shader::Use()**, **Model::Draw()**, y las rutinas de setup de **VAO/VBO** para lámparas.
 - La creación de instancias de animación (**InstanciaAnimacion**, **SillaAnimada**) y su distribución en filas y columnas dentro del laboratorio.
 - Función animación, donde se tienen los estados de animación para los componentes del laboratorio, además de detallar la animación por Keyframes que realizan las sillas del salón.
8. Buenas prácticas de refactorización, recomendaciones para gestionar la limpieza de recursos (VAO/VBO, texturas) y permitir la escalabilidad futura.

Con este manual, se establecen las bases técnicas y procedimentales necesarias para comprender a fondo cada línea de código y cada recurso utilizado, asegurando que el sistema sea mantenible, extensible y documentado conforme a estándares de ingeniería de software.

Objetivo del proyecto.

Este manual técnico tiene como finalidad ofrecer una referencia exhaustiva y detallada sobre el diseño, la implementación y el mantenimiento del proyecto “Campus virtual interactivo” para el mejoramiento del laboratorio de computación gráfica, en un entorno 3D inmersivo dirigido a estudiantes y profesores de la Facultad de Ingeniería. Sus objetivos específicos son:

1. Definir el alcance funcional del sistema: Nos permite describir las funcionalidades principales como lo son la exploración en primera persona, animaciones de mobiliario y equipos, transición entre salas, control de iluminación y efectos especiales.

2. Orientar a desarrolladores y colaboradores: Se provee un mapa claro de la arquitectura de carpetas y módulos de código, facilitando la incorporación de nuevos desarrolladores al proyecto. Junto con esto Explicamos la interconexión entre clases clave (***Application, Renderer, Model, AnimationManager, Camera***) y cómo añadir funcionalidades o corregir errores sin comprometer la cohesión del sistema.
3. Detallar las tecnologías y bibliotecas empleadas: Se justifica la selección de OpenGL 3.3+ para compatibilidad multisistema y acceso a shaders modernos.
4. Garantizar buenas prácticas de desarrollo: Describir patrones de diseño aplicados (por ejemplo, Factory para creación de instancias de animación, State Machine para fases de transición). Además de sugerir estándares de nomenclatura, gestión de memoria (creación y destrucción de VAO/VBO, texturas y shaders) y organización de "Makefiles" o archivos de proyecto de Visual Studio 2022.
5. Facilitar el mantenimiento y la extensibilidad: Mediante la explicación del uso de Git y GitHub (ramas, pull requests, code reviews) para coordinar el trabajo colaborativo y asegurar la calidad mediante revisiones continuas.
6. Ofrecer referencias de soporte y ampliación: Indicar cómo integrar nuevos formatos de modelo, sistemas de partículas o físicas más complejas, dando un punto de partida para futuras versiones del proyecto.

En conjunto, este manual técnico no solo explica **qué** hace el "Campus virtual interactivo", sino **cómo** lo hace y **por qué** se eligen determinadas soluciones. De este modo, se convierte en una guía de consulta permanente tanto para la evolución del código como para la formación de nuevos integrantes del equipo de desarrollo.

Público objetivo.

Este manual técnico ha sido diseñado para cubrir las necesidades e intereses de tres perfiles principales de usuarios, con el fin de maximizar su utilidad y asegurar que cada grupo obtenga el conocimiento y las herramientas que necesita:

- Estudiantes de la Facultad de Ingeniería: Tiene como interés principal comprender los pilares teóricos y prácticos de la renderización en tiempo real, la gestión de escenas 3D y las técnicas de animación por interpolación y keyframes. Además de tener algunos beneficios como:

- Ejemplos de código comentado que ilustran paso a paso la creación de un contexto OpenGL, la configuración de shaders y la carga dinámica de modelos con Assimp.
 - Explicaciones de patrones de diseño (máquina de estados para animaciones, uso de vectores y matrices con GLM) aplicados en un proyecto real.
- Profesores y tutores de computación gráfica: Se busca evaluar la calidad técnica del proyecto, entender su estructura interna y disponer de material de apoyo para enriquecer sus clases o supervisar trabajos de curso.
 - Patrocinadores y desarrolladores de equipamiento: Valorar el impacto de la propuesta de modernización, entender las especificaciones técnicas necesarias y colaborar en el diseño de soluciones de hardware adaptadas al software.

Cronograma de actividades.

Responsable	Actividades	Inicio	Final	Marzo 17 - 23	Marzo 24- 30	Marzo 31 - Abril 06	Abril 07 - 13	Abril 14 - 20	Abril 21 - 27	Abril 28 - Mayo 04	05/05/2025
Equipo	Reunión y presentación de equipo	19/03/2025	23/03/2025								
Equipo	Borrador propuesta de proyecto.	25/03/2025	30/03/2025								
Miguel	Reunión para análisis de distribución.	05/04/2025	05/04/2025								
Equipo	Entrega propuesta de proyecto.	06/04/2025	06/04/2025								
Miguel	Creación de repositorio y drive.	07/04/2025	07/04/2025								
Diego	Diseño de objetos para laboratorio.	07/04/2025	14/03/2025								
Rodrigo	Modelado de objetos.	14/03/2025	15/03/2025								
Diego	Texturas de modelos.	15/03/2025	16/03/2025								
Rodrigo	Documentación primera parte.	10/04/2025	16/03/2025								
Miguel	Propuestas de fuentes de luz.	16/03/2025	18/03/2025								
Diego	Transición a laboratorio actual.	16/03/2025	20/03/2025								
Rodrigo	Modelado de objetos nuevos.	19/03/2025	21/03/2025								
Miguel	Fuentes de luz en objetos nuevos.	19/03/2025	22/03/2025								
Equipo	Segunda revisión.	22/03/2025	22/03/2025								
Equipo	Entrega de avance.	23/04/2025	23/04/2025								
Rodrigo	Documentación segunda parte.	17/04/2025	26/04/2025								
Diego	Primera animación	24/04/2025	28/04/2025								
Miguel	Segunda animación.	24/04/2025	28/04/2025								
Rodrigo	Manual de usuario.	28/04/2025	04/05/2025								
Diego	Manual técnico.	28/04/2025	04/05/2025								
Equipo	Presentación formal del producto.	05/05/2024	05/05/2024								

Tabla 1. Cronograma de actividades

Requerimientos de hardware y software.

Para asegurar un funcionamiento óptimo de la simulación del laboratorio virtual de computación gráfica, a continuación se detallan y explican en profundidad los recursos mínimos y recomendados tanto de hardware como de software.

Sistema operativo: La aplicación es multiplataforma, por lo que puedes ejecutarla en cualquiera de los siguientes sistemas:

- Windows 10/11, en versiones de 64 bits. Se recomienda mantener el sistema actualizado para contar con los últimos controladores de GPU.
- Linux (Ubuntu 20.04+), en distribuciones basadas en Debian/Ubuntu, puedes instalar las dependencias mediante el gestor apt.
- MacOS 12+. Asegúrate de habilitar el soporte de desarrollo de línea de comandos (Xcode Command Line Tools) para compilar desde terminal.

Hardware mínimo:

Componente.	Especificación mínima.	Por qué es importante.
GPU	Soporte para OpenGL 3.3 o superior.	OpenGL 3.3 introduce shaders, buffers y transformaciones necesarias para el renderizado.
Memoria RAM	8 GB	Permite cargar modelos 3D, texturas y realizar animaciones sin ralentizaciones.
CPU	Dual-core a 2.5 GHz (o equivalente moderno).	Responsable de la lógica del programa, gestión de animaciones y llamadas a OpenGL.
Almacenamiento	10 GB de espacio libre en disco (SSD preferido)	Para compilar, guardar recursos (modelos, texturas) y mantener un rendimiento ágil.

Tabla 2. Requisitos mínimos de hardware.

Nota: Para escenas muy complejas, con múltiples luces y modelos de alta resolución, se recomienda una GPU con al menos 4 GB de VRAM y 16 GB de RAM para mantener 60FPS estables.

Software:

- Compilador C++, con soporte de C++11 o superior.
 1. g++ (GNU Compiler Collection) en Linux/macOS.
 2. Clang en macOS o Linux modernos.
 3. Visual Studio Community / Build Tools en Windows.

Por qué: Las características de C++11 (como auto, Lambda functions y bibliotecas estándar mejoradas) se utilizan ampliamente en las clases de la aplicación.

Se debe considerar las siguientes bibliotecas gráficas y de utilidades:

Biblioteca.	Funcionalidad principal.
GLEW	Gestión de extensiones de OpenGL, necesario para acceder a funciones modernas de la API.
GLFW	Creación de ventanas, gestión de contexto OpenGL y eventos de entrada (teclado, ratón).
GLM	Colección de funciones y tipos matemáticos (vectores, matrices) para transformaciones 3D.
Assimp	Carga de formatos de modelos 3D (.obj, .fbx, etc.) de manera unificada.
SOIL2 o stb_image	Carga de imágenes y texturas en formatos comunes (.png, .jpg, etc.).

Tabla 3. Bibliotecas utilizadas dentro del código.

Con estos elementos correctamente instalados y configurados, tu entorno estará listo para compilar y ejecutar la simulación, garantizando una experiencia fluida y sin sorpresas durante la ejecución.

Instalación de repositorio y control de versiones.

A continuación encontrarás una guía para obtener el código fuente del proyecto, tanto desde línea de comandos como usando GitHub Desktop, así como para contribuir con reportes de errores o mejoras en GitHub.

Prerrequisitos.

Antes de clonar el repositorio, asegúrate de contar con lo siguiente instalado en tu máquina:

- Contar con Git (versión 2.20 o superior) y C++ (compilador compatible con C++11, por ejemplo g++ en Linux/macOS o Visual Studio Community/Build Tools en Windows)
- Conexión a internet para descargar dependencias y sincronizar con GitHub.

Tip: En Windows puedes instalar Git y mediante el instalador de [Chocolatey](#).

Clonación del repositorio con líneas de comando.

1. Abrir una terminal, ya sea en Windows el PowerShell o Git Bash, si utiliza mac OS/Linux abrir la terminal nativa.
2. Navegar a la carpeta donde deseas ubicar el proyecto, **cd /ruta/carpeta/local**
3. Para clonar el repositorio oficial ejecuta el siguiente comando para descargar todos los archivos y el historial de versiones.

URL: <https://github.com/Miguel07FI/Computacion-Grafica-Equipo-11>

Clonación:

```
git clone  
https://github.com/Miguel07FI/Computacion-Grafica-Equipo-11.git  
cd "ubicación local"/Computacion-Grafica-Equipo-11
```

4. Esto creará una carpeta llamada **Computacion-Grafica-Equipo-11** con todo el contenido del proyecto.

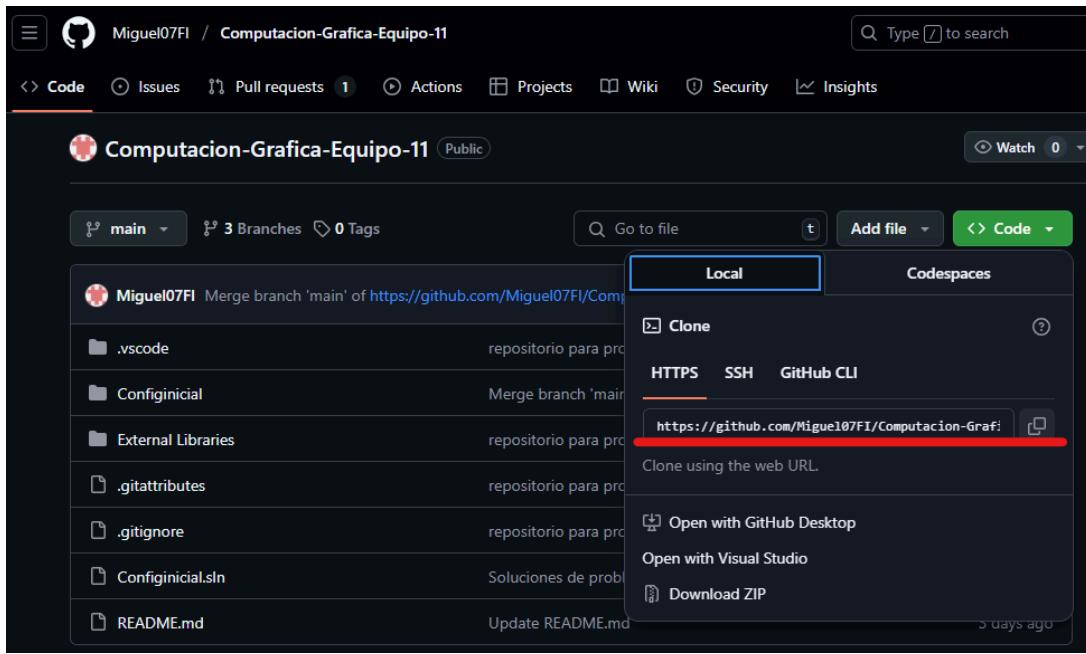


Figura 1. Enlace HTTPS para poder realizar la clonación del repositorio

Clonación del repositorio con GitHub Desktop.

Si se desea utilizar una aplicación de escritorio con interfaz para poder crear y bajar repositorios, se recomienda lo siguiente:

1. Descarga e instala GitHub Desktop desde <https://desktop.github.com/>
2. Inicia GitHub Desktop y haz login con tu cuenta de GitHub.
3. En la esquina superior izquierda, haz clic en File → Clone repository...
4. En la pestaña URL, pega la dirección del repositorio: <https://github.com/Miguel07FI/Computacion-Grafica-Equipo-11>
5. Elige la **ubicación local** donde deseas clonar el proyecto (por ejemplo, tu carpeta de "Proyectos").
6. Haz clic en **Clone**. GitHub Desktop descargará el repositorio y lo mostrará en la lista de tus proyectos.
7. Para acceder a los archivos, abre la carpeta resultante (Computacion-Grafica-Equipo-11) con tu explorador de archivos o directamente con Visual Studio 2022.

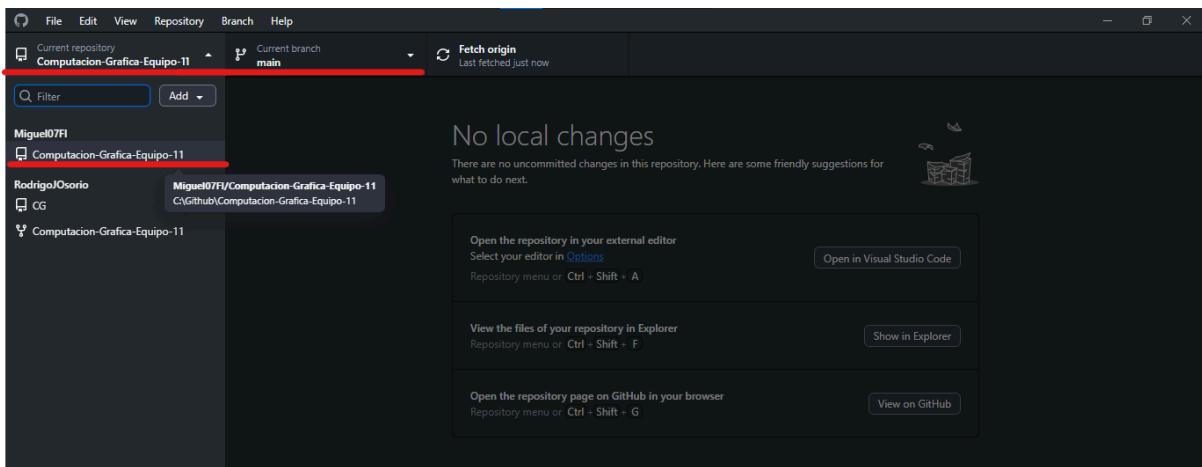


Figura 2. Detalles a verificar para poder trabajar de forma adecuada.

Verificar la estructura de las carpetas.

Dentro de Computacion-Grafica-Equipo-11 deberías ver al menos:

- **.git/**
Carpeta oculta con el historial de versiones y la configuración de Git.
- **Configinicial/, External Libraries/**
Contiene la configuración base, librerías DLL (Windows), modelos y texturas.
- **Models/, Shader/, Debug/, etc.**
Subdirectorios que organizan tus recursos 3D, archivos de sombreado y compilaciones de prueba.
- **Archivos fuente (.cpp, .h)**
El núcleo de la simulación, incluyendo PROYECTO EQUIPO 11.cpp y clases como Model.h, Shader.h, Camera.h.

Si alguno de estos elementos no aparece, comprueba que el comando git clone se haya completado sin errores.

Contribuir y reportar issues.

Para abrir un issues, sigue las indicaciones:

1. Desde tu navegador, ve a la sección **Issues** del repositorio.
2. Haz clic en New issue, describe el problema, indicando sistema operativo, versión de dependencias y pasos para reproducirlo.

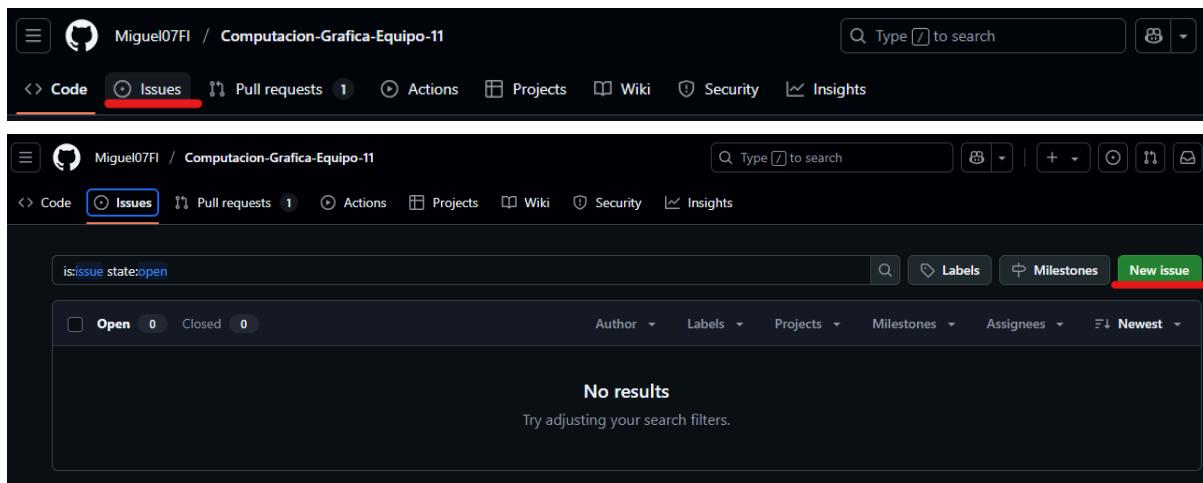


Figura 3. Pestaña de issues para contribuir a la mejora del desarrollo.

Para enviar un **Pull request**:

1. En GitHub Desktop, selecciona Branch → New Branch, nómbrala descriptivamente (por ejemplo, mejora-modelos).
2. Realiza tus cambios y confirma con Commit to <branch>.
3. Haz clic en Push origin para subir la rama a GitHub.
4. En la interfaz web de GitHub, abre un Pull request desde tu rama hacia main, describe tus modificaciones y enviarlo para revisión.

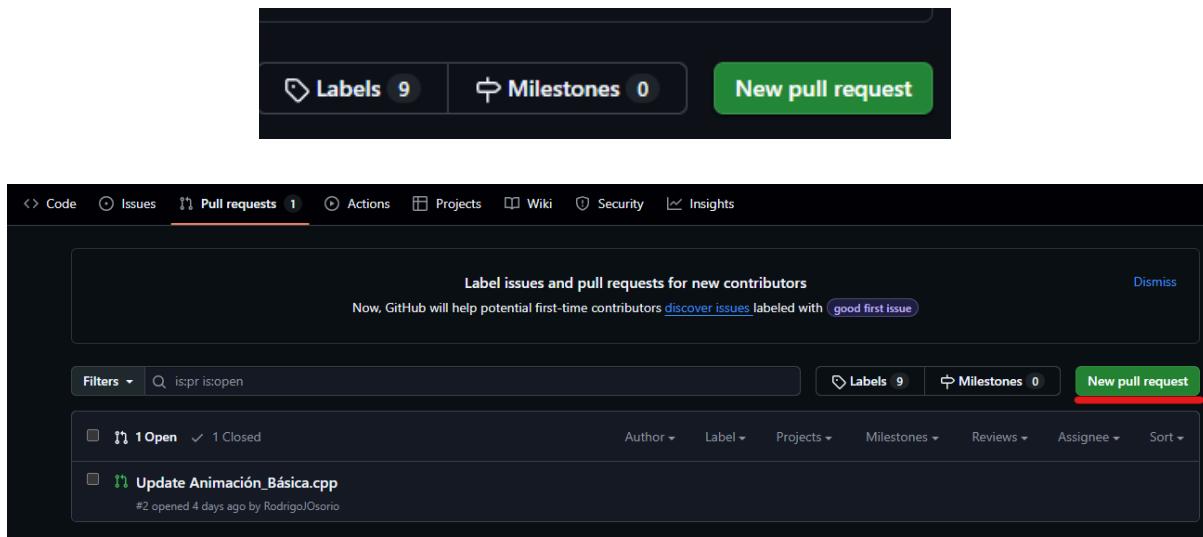


Figura 4. Pestaña Pull Request

Con estos métodos podrás clonar, explorar y colaborar fácilmente en el proyecto, ya sea usando comandos de Git o una interfaz gráfica amigable como GitHub Desktop.

Herramientas de modelado y texturizado.

Para la creación de los activos 3D que componen el “salón actual” del laboratorio, el equipo empleó dos pilares en su flujo de trabajo:

Blender.

Se realiza el modelo para un muñeco que va a estar haciendo un recorrido por el laboratorio de computación gráfica, realizado desde blender para importar su formato .obj y .mtl. Realizando el trabajo de separación de sus extremidades, los brazos, manos, piernas y cabeza.

Su textura fue trabajada a partir de photoshop para que finalmente se pueda obtener el modelo siguiente.

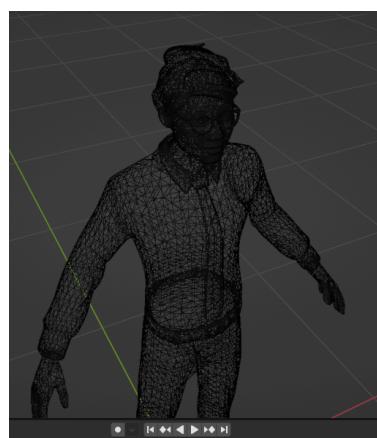


Figura 6. Textura de modelo del muñeco.



Figura 7. Modelo del muñeco.

Modelos para salón actual.

Todos los elementos que conforman el laboratorio "tal como está hoy" se encuentran bajo la carpeta **Models/salon/** y otros subdirectorios relacionados:

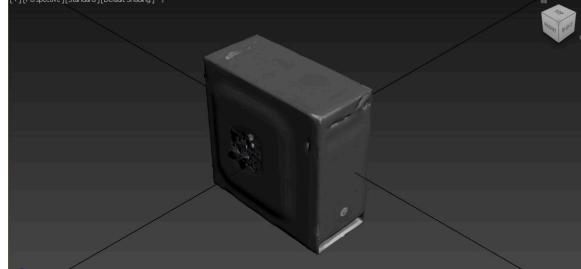
- **Models/salon/salon2.obj + salon2.mtl + texturas asociadas.**
- Modelo completo de la sala: columnas, paredes, suelo y techo, generado en Blender y exportado con materiales múltiples para diferenciar zonas (suelo, paredes, cielo rasante).

A continuación se presenta el mobiliario del laboratorio actual de computación gráfica, así como también para tener un mejor mobiliario y los componentes nuevos para el mejoramiento del laboratorio.

Modelos obtenidos apartir del uso de blender, 3dmax y tripo.com

- <https://www.trip03d.ai/>

(El uso de modelos generados fue de bastante utilidad, pues se obtuvo una base para detallar en las herramientas de software como 3dmax y blender para asegurar una textura y modelos coherentes)

Descripción	Modelo objetivo.
Modelo del gabinete de la PC actual del laboratorio de computación gráfica.	 Figura 8. Modelo de PCs actuales.
Cortina para nuevo modelo de salón, color oscuro.	 Figura 9. Modelo de cortinas remodeladas

Cortina actual del laboratorio de computación gráfica, de color azul, se coloca en las ventanas del salón parte derecha.

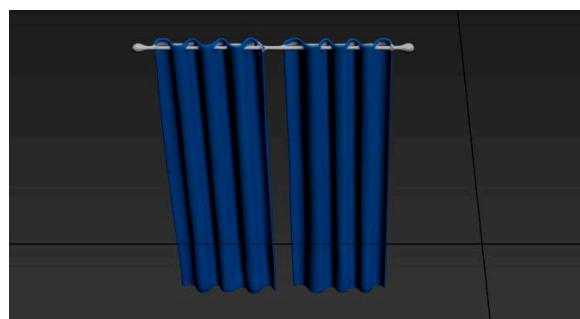


Figura 10. Modelo de cortinas actuales.

Monitor actual del laboratorio de computación gráfica.

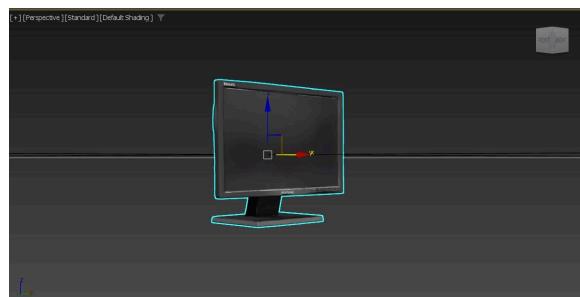


Figura 11. Modelo de monitores actuales.

Teclado actual del laboratorio de computación gráfica.



Figura 12. Modelo de teclados actuales.

Extintor para emergencias ubicado en el salón.

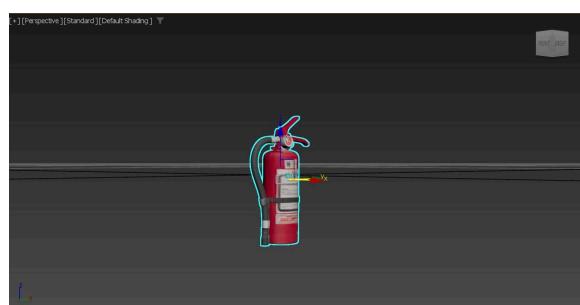
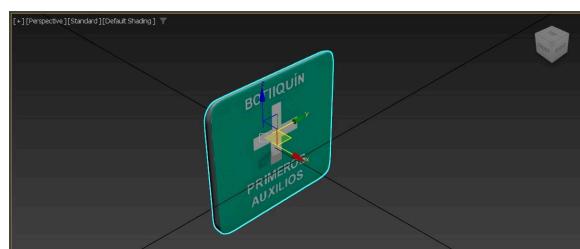
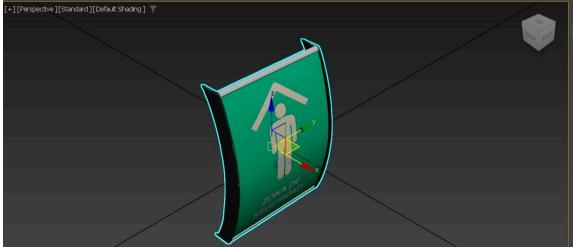
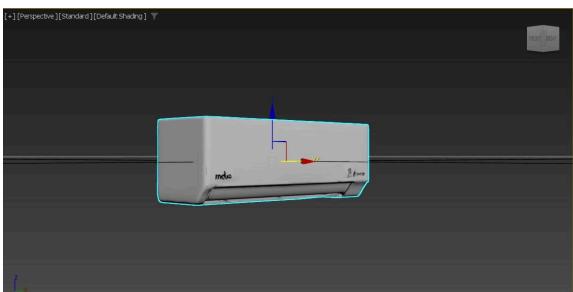


Figura 13. Modelo de extintor actual.

Señalización y el botiquín de primeros auxilios.



	<p>Figura 14. Modelo de señalización botiquín.</p>
Señalización de no introducir comida o bebidas en el laboratorio.	
Señalización de lugar seguro en caso de sismo.	
Aire acondicionado del laboratorio, ubicado en la zona de las ventanas junto a las cortinas.	
Fuente de poder para PC nueva.	

Gabinete para el armado de la nueva PC, con componentes de alta tecnología.



Figura 19. Modelo de gabinete de PC.

Motherboard para instalar en el nuevo gabinete.



Figura 20. Modelo de motherboard.

Tarjetas de memoria RAM, como nuevos componentes para la mejora del PC.

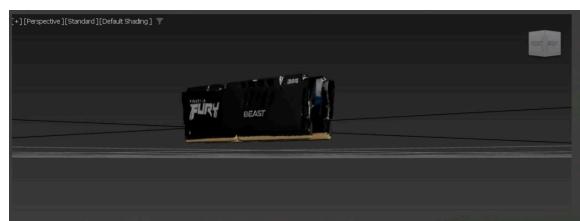


Figura 21. Modelo de tarjetas RAM.

Monitor nuevo de alta tecnología.

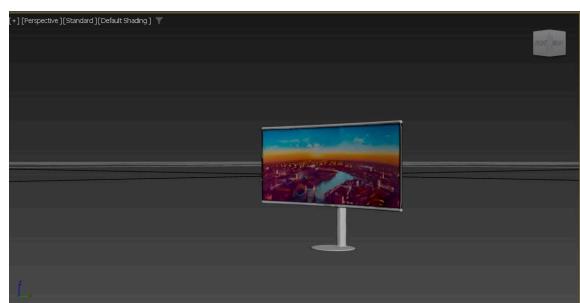


Figura 22. Modelo monitor remodelado.

Teclado mecánico gamer para una mejor respuesta.



Figura 23. Modelo de teclado remodelado.

Tarjeta gráfica de alta capacidad instalada en el nuevo gabinete.



Figura 24. Modelo tarjeta gráfica.

Procesador CPU de última tecnología para tener un mejor rendimiento en los programas del laboratorio.

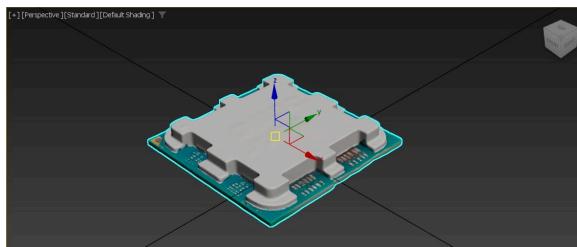


Figura 25. Modelo de procesador.

Señalización y documento para las reglas del laboratorio.

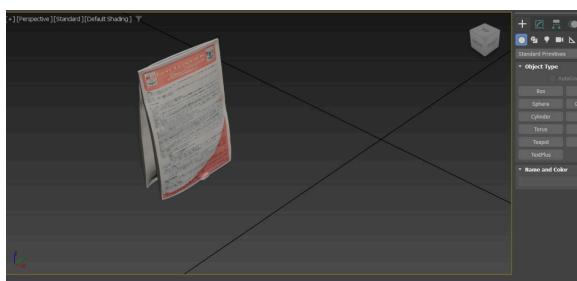


Figura 26. Modelo reglas de laboratorio.

Señalización de prohibido fumar.

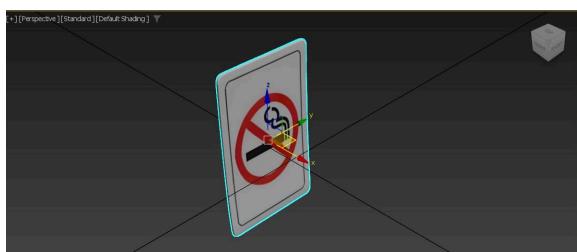


Figura 27. Modelo de señalización prohibido fumar.

Señalización de guardar silencio y no utilizar el celular dentro del laboratorio.

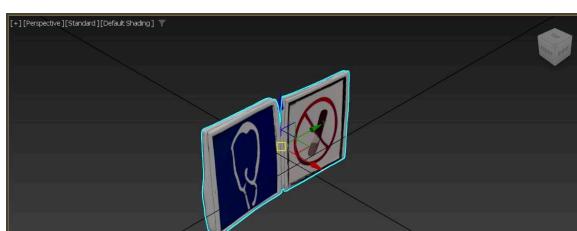
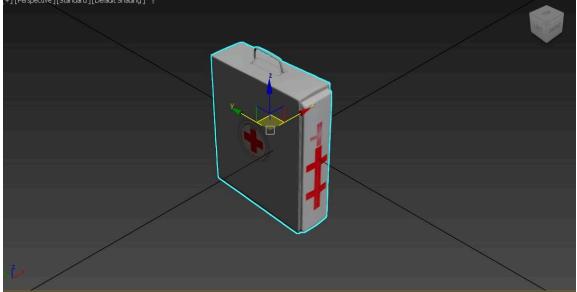
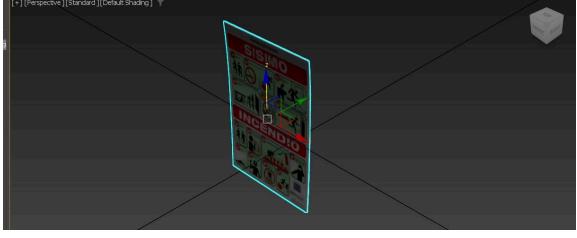
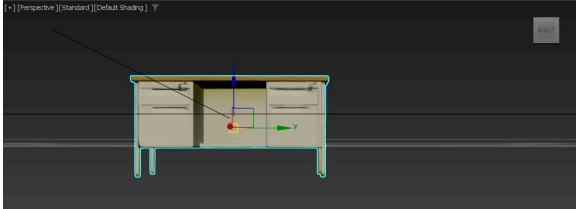
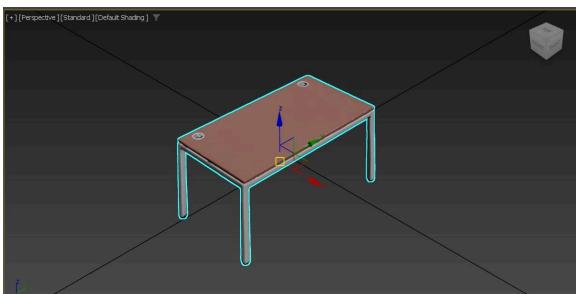


Figura 28. Modelos de señalización

	para guardar silencio y prohibido teléfono.
Botiquín de primeros auxilios.	 <p>Figura 29. Modelo de botiquín.</p>
Señalización para casos de incendio y sismos.	 <p>Figura 30. Modelo de señalización en caso de emergencia.</p>
Escritorio que se encuentra en el fondo del laboratorio sin uso.	 <p>Figura 31. Modelo de escritorio actual.</p>
Mesa arrinconada en el laboratorio.	 <p>Figura 32. Modelo de mesa extra del laboratorio actual.</p>

Mesa de trabajo en el área del profesor, en frente del laboratorio.

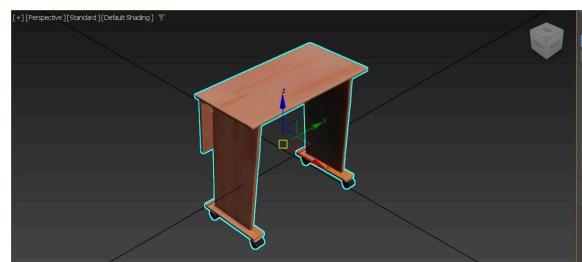


Figura 33. Modelo de mesa de trabajo de profesor.

Pizarrón actual del laboratorio.

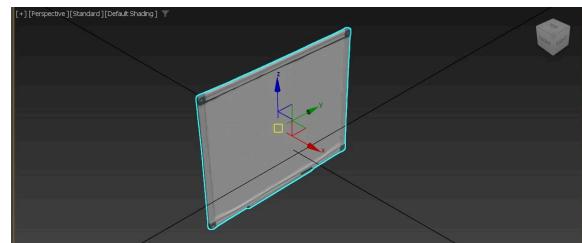


Figura 34. Modelo de pizarrón actual.

Pantalla de alta resolución con touch, conectada a la computadora principal del profesor.

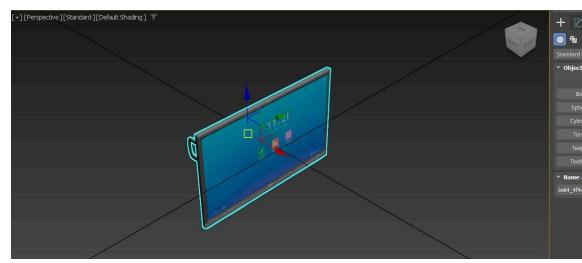


Figura 35. Modelo pantalla táctil de laboratorio remodelado.

Proyector que se encuentra en el actual laboratorio.



Figura 36. Modelo de proyector actual.

Propuesta del modelo para ventana actual del laboratorio.

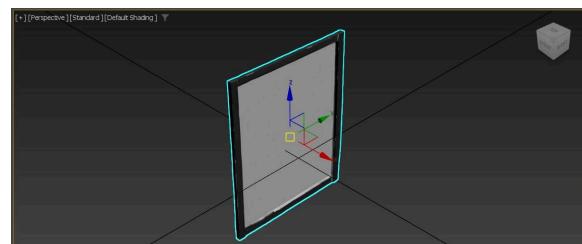


Figura 37. Modelo de ventana del laboratorio.

Rack para servidor del laboratorio.

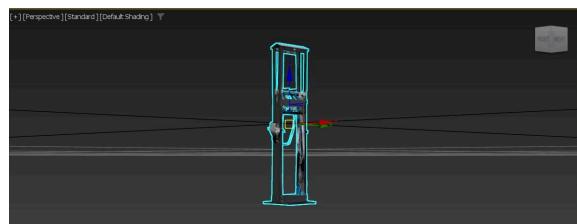


Figura 38. Modelo de rack actual de laboratorio.

Silla actual del laboratorio.

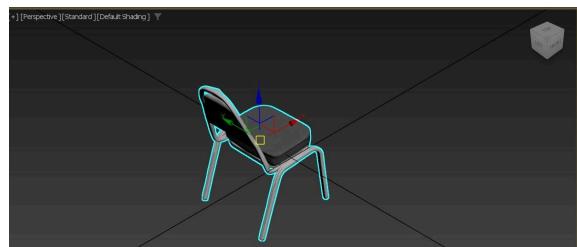


Figura 39. Modelo de sillas actuales.

Propuesta de nueva silla para el laboratorio.



Figura 40. Modelo de sillas remodeladas.

Propuesta para mesa de trabajo para estudiantes del nuevo laboratorio.

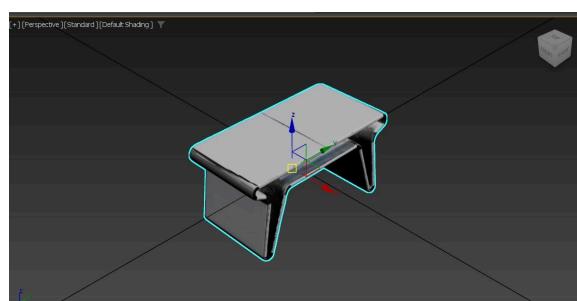


Figura 41. Modelo de mesas remodeladas.

Mesa de trabajo para estudiantes del laboratorio, donde se encuentran los equipos de cómpupto actuales.

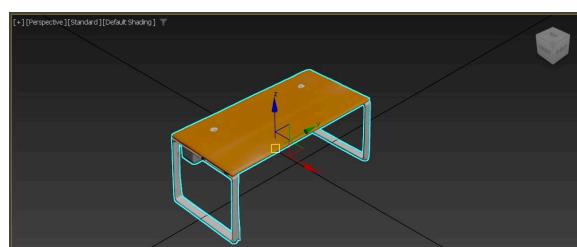


Figura 42. Modelo de mesas actuales.

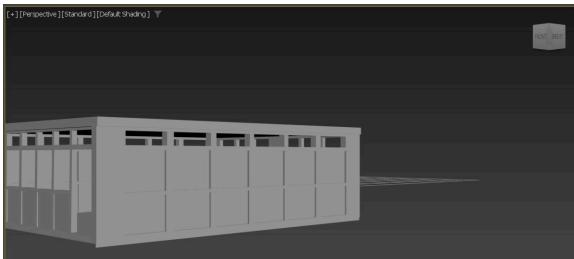
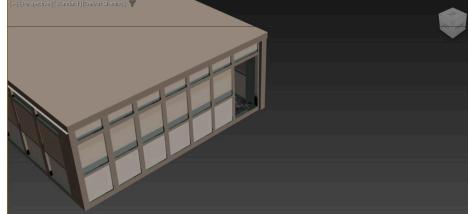
Estructura de paredes y ventanas del laboratorio de computación gráfica.	
Estructura y techo del laboratorio.	
Modelo de profesor para animación por keyframes.	
Logos para animación en pantalla nueva.	

Tabla 4. Modelos utilizados.

Cada carpeta contiene, al menos, un archivo **.obj** (malla), un **.mtl** (materiales con rutas a texturas) y las imágenes de textura correspondientes. Gracias a **Assimp** en **Model.h**, estos activos se importan automáticamente al programa, respetando coordenadas de UV y configuraciones de material definidas en Blender.

Arquitectura general código fuente.

El proyecto “Campus virtual interactivo” sigue una estructura modular dividida en fases claras que separan la inicialización, la carga de recursos, el manejo de entrada, la lógica de animación y el renderizado. Esta organización facilita el mantenimiento, la extensión y el debugging.

Estructura de archivos y módulos.

Tras clonar o descargar el repositorio, comprueba que tu carpeta **ProyectoGrafica/** contenga al menos lo siguiente:

```
ProyectoGrafica/
├── .git
├── .vs
└── .vscode    ← Configuración local de Visual Studio
├── Configinicial
│   ├── Debug    ← DLLs y librerías compiladas
│   └── Models
│       ├── Modelos necesarios para laboratorio actual y nuevo.
│       │   cada carpeta contiene ` `.obj` , ` `.mtl` y texturas.
│       ├── Shader
│       ├── Shader_
│       ├── SOIL2
│       ├── assimp-vc140-mt.dll
│       ├── Camera.h
│       ├── Mesh.h
│       ├── Model.h
│       ├── Shader.h
│       ├── stb_image.h
│       ├── Configinicial.vcxproj
│       ├── Configinicial.vcxproj.filters
│       ├── Configinicial.vcxproj.user
│       ├── glew32.dll
│       └── PROYECTO EQUIPO 11.cpp
└── Debug      ← Carpeta de salida tras compilar
└── External Libraries
```

```
| .gitattributes  
| .gitignore  
|- ConfigInicial.sln ← Solución de Visual Studio 2022
```

Si faltara alguno de estos elementos, revisa que la clonación (*git clone ...*) se completará sin errores.

Puntos clave de código fuente.

El proyecto sigue una arquitectura modular estructurada en las siguientes fases:

1. Inicialización.

Se realiza la configuración e inicialización de las bibliotecas GLFW, GLM y GLEW, para manejo de las extensiones necesarias de OpenGL. Además se tiene la creación de:

- GLFW: se invoca `glfwInit()` y se configuran las versiones de contexto OpenGL.
- La ventana se crea con `glfwCreateWindow(width, height, title, ...)`.
- Registro de **Callback** de teclado llamado por **KeyCallback** y ratón con **MouseCallback** gestionar interacciones asíncronas.
- GLEW: tras crear el contexto, `glewExperimental = GL_TRUE;` `glewInit()`; carga los punteros a funciones modernas de OpenGL.
- Estados globales: `glEnable(GL_DEPTH_TEST);` habilita el test de profundidad, y opcionalmente **GL_BLEND**, **GL_CULL_FACE** según necesidad.

2. Carga de recursos.

Se encarga de la compilación y enlace de shaders Shader **lightingShader**, Shader **lampShader**. También de la importación de modelos con **Model::Model(path)** que se tiene en la carpeta **Models/**, usando **Assimp::Importer** en **Model::Model(path)**, se cargan mallas, materiales y texturas desde Models/. Cada modelo crea sus VAO/VBO/ EBO y asigna atributos de vértice (posición, normal, UV).

La configuración de buffers para la creación de un cubo skybox o lámpara prueba con VAO/VBO, dedicados para demostrar el pipeline de vértices sin necesitar un modelo externo.

Por otro lado, tenemos la instanciación de animaciones con la construcción de vectores (`std::vector<InstanciaAnimacion>`

animaciones; std::vector<SillaAnimada> sillas;) con las propiedades iniciales (posiciones, escalas y keyframes) de cada objeto animable.

3. Gestión de entradas y cámaras.

Las siguientes funciones ayudan a las entradas que necesitamos para registrar teclas y cómo manipular el entorno virtual de nuestra ventana:

- **KeyCallback:** Cada vez que se pulsa o suelta una tecla, se actualiza **keys[key] = true/false**. Estas banderas activan animaciones (explosionActive, assemblePC).
- **DoMovement():** Lee el estado de **keys** para mover la cámara con **camera.ProcessKeyboard(direction, deltaTime)** usando W,A,S,D o flechas. También ajusta las luces puntuales y el foco del proyector con teclas como T/G/Y/H/U/J.
- **MouseCallback():** Captura desplazamientos del ratón y llama a **camera.ProcessMouseMovement(xoffset, yoffset)** para rotar la vista.

4. Ciclo de renderizado para la creación de modelos y sus animaciones.

```
while (!glfwWindowShouldClose(window)) {  
    // 1. Cálculo de deltaTime para sincronizar animaciones  
    currentTime = glfwGetTime();  
    deltaTime = currentTime - lastFrame;  
    lastFrame = currentTime; //Manejo de eventos y Lógica  
    glfwPollEvents();  
    DoMovement();  
    Animation();  
    glClearColor(0.1f, 0.1f, 0.1f, 1.0f); //Limpieza de buffers  
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);  
    // 4. Renderizado con LightingShader  
    lightingShader.Use();  
    // - Envío de uniforms: viewPos, matrices de proyección y  
    // vista, propiedades de luces  
    // - Dibujado de cada modelo:  
    for (auto &model : modelos) {  
        model.Draw(lightingShader); }  
    // 5. Renderizado de Lámparas  
    lampShader.Use();  
    for (auto &lightPos : pointLightPositions) {  
        DrawLamp(lightPos); } // 6. Swap buffers  
    glfwSwapBuffers(window); }
```

El bucle principal se basa en **While** (`!glfwWindowShouldClose`) para realizar lo siguiente:

- Cálculo de **deltaTime** utilizado para sincronizar las animaciones por cada frame en tiempo de ejecución.
- Implementación de funciones `glfwPollEvents()`, `DoMovement()`, `Animation()`, que se encargan de la actualización de los estados que siguen las animaciones del laboratorio.
- Limpieza de buffers con `glClear`.
- Envío de la información y enlace a shaders con **Bind** y los **Uniforms**, además de contemplar las matrices para materiales e iluminación.

En la parte de renderizado tenemos las instancias:

- Iteración sobre animaciones: `Update()`, `Draw()` con transformaciones propias.
- Iteración sobre sillas: actualización **keyframe** y `Draw()`.
- Dibujo de Com, Com2, me, con animaciones de explosión y aparición.
- Renderizado de lámparas con `lampShader` y `glfwSwapBuffers()`.

5. Finalización y liberación de recursos.

La destrucción de `glDeleteVertexArrays`, `glDeleteBuffers` para cada **VAO/VBO/EBO**. y texturas de **Model**, nos permite liberar recursos del ciclo de renderizado con `glfwTerminate()`. Y la destrucción de texturas gestionadas por **Model**.

Finalmente, también `glfwTerminate()` cierra la ventana y libera el contexto gráfico.

Módulos clave.

Shader lighting.frag

Este shader de fragmentos implementa un modelo de iluminación basado en tres tipos de luces:

1. **Direccional (DirLight)**: luz con dirección fija que afecta por igual a todos los fragmentos, como el sol.

2. **Puntual (PointLight):** luz que emana de un punto en el espacio y decrece con la distancia, como una bombilla.
3. **Focal (SpotLight):** luz puntual que emite en un cono, como un reflector.

Para cada uno de estos, se calcula el componente ambiental, difuso y especular usando el modelo de Phong:

- **Ambiental:** iluminación global tenue.
- **Difuso:** depende del ángulo entre la normal y la dirección de la luz $\max(\text{dot}(\text{normal}, \text{lightDir}), 0.0)$.
- **Especular:** brillos de alta intensidad que dependen de la dirección de visión, la normal y la dirección de la luz reflejada, elevado al **shininess** del material.

El shader admite un array de hasta **NUMBER_OF_POINT_LIGHTS** luces puntuales y una única luz direccional y una focal. Además, se puede descartar el fragmento (hacerlo totalmente transparente) si su componente alfa es muy bajo y la bandera **transparency** está activada.

Uniforms y estructuras.

1. `#define NUMBER_OF_POINT_LIGHTS 6`: Define en tiempo de compilación cuántas luces puntuales cabe en el array **pointLights[]**. Cambiar este valor es para obtener nuestras luces con focos en sus posiciones sobre el techo.
 2. `struct Material`: Se implementan sampler2D diffuse y sampler2D specular que manejan dos texturas independientes. La **diffuse** aporta el color base de la superficie y la **specular** controla dónde y con qué intensidad se muestran los brillos.
 3. `struct DirLight`: Se utiliza vec3 direction como vector unitario que indica de dónde viene la luz. También **vec3 ambient, diffuse, specular** para colores o intensidades para cada componente. Se suelen pasar desde la CPU según consulta de la escena.
 4. `struct PointLight`: Tenemos **const, linear, quadratic**: parámetros de atenuación según la fórmula $\text{attenuation} = \frac{1}{c + ld + qd^2}$
- Ambient, diffuse, specular:** análogos a DirLight, pero modulados por atenuación.
5. *Entradas (in) y salidas (out):* **FragPos, Normal, TexCoords** vienen del **vertex** shader, interpolados por fragmento. Como salida tenemos **color** el resultado final que pinta la pantalla.

6. struct SpotLight: Añade **cutOff** y **outerCutOff** cosenos de los ángulos interior y exterior del cono.

```
#version 330 core
#define NUMBER_OF_POINT_LIGHTS 6
struct Material { // Propiedades del material
    sampler2D diffuse;      // Mapa de textura difusa
    sampler2D specular;     // Mapa de textura especular
    float shininess;        // Exponente de brillo para el cálculo
especular };
struct DirLight { // Luz direccional
    vec3 direction;         // Dirección de La Luz
    vec3 ambient;            // Color ambiental
    vec3 diffuse;             // Color difuso
    vec3 specular;            // Color especular };
struct PointLight { // Luz puntual
    vec3 position;           // Posición en el espacio
    float constant;          // Término constante de atenuación
    float linear;             // Término Lineal de atenuación
    float quadratic;          // Término cuadrático de atenuación
    vec3 ambient;
    vec3 diffuse;
    vec3 specular; };
struct SpotLight { // Luz focal (spotlight)
    vec3 position;
    vec3 direction;          // Eje del cono de la Luz
    float cutOff;             // Ángulo interior del cono (coseno)
    float outerCutOff; // Ángulo exterior para suavizado (coseno)
    float constant;
    float linear;
    float quadratic;
    vec3 ambient;
    vec3 diffuse;
    vec3 specular; };
// Entradas interpoladas desde el vertex shader
in vec3 FragPos;           // Posición del fragmento en espacio mundo
in vec3 Normal;              // Normal interpolada
in vec2 TexCoords;           // Coordenadas de textura
// Salida del fragment shader
out vec4 color; // Uniforms de escena
uniform vec3 viewPos; // Posición de La cámara
uniform DirLight dirLight; // Luz direccional
uniform PointLight pointLights[NUMBER_OF_POINT_LIGHTS]; // Array de luces
puntuales
uniform SpotLight spotLight; // Luz focal
uniform Material material; // Propiedades del material
uniform int transparency; // Flag para descartar fragmentos muy
```

transparentes

Función main.

Se realiza una normalización para que las operaciones **dot** y **reflect** asumen vectores con longitud 1 para generar resultados correctos y predecibles.

Para obtener una iluminación se inicializa result con la contribución global del "sol" de la escena, de tal forma que itera sobre todas las luces puntuales, acumulando su aporte. Finalmente el **Canal alfa** se toma del canal rojo de la textura difusa (suponiendo un mapa de opacidad en R).

```
void main() { // Normalizar normal y dirección de visión
    vec3 norm = normalize(Normal);
    vec3 viewDir = normalize(viewPos - FragPos);
    // 1) Iluminación direccional
    vec3 result = CalcDirLight(dirLight, norm, viewDir);
    // 2) Sumar contribuciones de cada Luz puntual
    for (int i = 0; i < NUMBER_OF_POINT_LIGHTS; i++)
        result += CalcPointLight(pointLights[i], norm, FragPos,
viewDir);
    // 3) Añadir Luz focal
    result += CalcSpotLight(spotLight, norm, FragPos, viewDir);
    // Componer color final: rgb = result, alfa = del mapa difuso
    color = vec4(result, texture(material.diffuse, TexCoords).r);
    // Descarta fragmentos con alpha < 0.1 si transparency está activado
    if (color.a < 0.1 && transparency == 1)
        discard;}
```

Cálculos de iluminación.

Luz Direccional.

Para la luz direccional light.direction suele apuntar hacia la fuente, por eso se invierte con -light.direction. Por otra parte, **material.shininess** controla el tamaño del brillo especular: valores altos → puntos de luz pequeños y concentrados.

Luz puntual.

Mientras que en la luz puntual la atenuación simula la pérdida de intensidad con la distancia, usando los coeficientes **constant**, **linear** y **quadratic**.

La atenuación se calcula mediante:

$$\text{attenuation} = \frac{1}{c + ld + qd^2}$$

Controla cómo disminuye la intensidad con la distancia d . Además va a multiplicar cada componente (ambiental, difusa y especular).

Luz Focal.

En esta luz **cutOff** y **outerCutOff** están en cosenos de ángulo; el suavizado evita un borde de sombra duro.

Se realiza el cálculo del coseno del ángulo entre el eje del foco y la dirección al fragmento: $\theta = L \cdot light.direction$

Define un rango entre **outerCutOff** y **cutOff**, para después multiplicar **rattenuation * intensity** para suavizar bordes.

```
// 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 );
    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 );
    // Diffuse shading
    float diff = max( dot( normal, lightDir ), 0.0 );
    vec3 reflectDir = reflect( -lightDir, normal ); // Specular shading
    float spec = pow( max( dot( viewDir, reflectDir ), 0.0 ),
material.shininess );
    // Attenuation
    float distance = length( light.position - fragPos );
    float attenuation = 1.0f / ( light.constant + light.linear *
distance + light.quadratic * ( distance * distance ) );
    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;
    diffuse *= attenuation;
    specular *= attenuation;
    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 );
// 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 epsilon = light.cutOff - light.outerCutOff;
float intensity = clamp( ( theta - light.outerCutOff ) / epsilon,
0.0, 1.0 );
    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;
    return ( ambient + diffuse + specular ); }

```

Shader lighting.vs

Este *vertex shader* prepara cada vértice para el *fragment shader* de iluminación, realizando los siguientes pasos:

1. **Explosión de malla:** desplaza cada vértice a lo largo de su normal, controlado por **explosionFactor**.

2. **Transformación espacial:** convierte la posición “explotada” de espacio local a coordenadas de clip mediante las matrices **model**, **view** y **projection**.

3. Cálculo de atributos de vértice **VBO** para el **fragment shader**:

- FragPos: posición en espacio mundo, para cálculos de iluminación y atenuación, teniendo como componentes X, Y, Z.
- Normal: normal transformada adecuadamente por la matriz inversa-traspuesta del modelo.
- TexCoords: coordenadas de textura sin modificar.

Los **uniforms** que se utilizan son para:

- **model**: traslada, rota y escala vértices desde espacio local a mundo.
- **view**: posiciona y orienta la cámara.
- **projection**: aplica la proyección en clip (por ejemplo perspectiva).
- **explosionFactor**: escalar el desplazamiento de vértices a lo largo de la normal para un efecto de “explotar” la malla.

```
#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;
uniform float explosionFactor;
void main() {
    vec3 explodedPosition = position + normal * explosionFactor;
    // Usa la posición explosionada para todo
    vec4 worldPosition = model * vec4(explodedPosition, 1.0);
    FragPos = vec3(worldPosition);
    gl_Position = projection * view * worldPosition;
    Normal = mat3(transpose(inverse(model))) * normal;
    TexCoords = texCoords; }
```

Variables globales.

Variables para animación del monitor.

Tenemos las siguientes variables globales de animación "X", y control de escena.

```
float globalTranslationYInicial = 7.0f;
float globalTranslationYFinal = 0.5f;
bool animacionXTerminada = false;
glm::vec3 globalTranslationX = glm::vec3(0.0f,
globalTranslationYInicial, -25.8f);
glm::vec3 globalScaleX = glm::vec3(4.0f, 4.0f, 4.0f);
float globalRotationX = 90.0f;
bool bajandoY = false;
float velocidadBajadaY = 4.0f; //Controla la velocidad de descenso en Y
bool reduciendoEscala = false;
float escalaTemporal = 4.0f;
```

Dónde se define la transformación global inicial (posición, escala y rotación) de todo el grupo "X" y flags para controlar si está "bajando" o "reduciendo escala" durante la animación.

Variables de animación específica de sillas "si" y "sn".

Las siguientes variables controlan fase por fase la animación de intercambio entre dos sillas, su posición y escalado inicial.

```
bool animSiActive = true; // La silla original aparece al inicio
bool animSnActive = false; // La silla "nueva" espera para animarse
bool siMovingOut = false, siShrinking = false;
bool snAppearing = false, snMovingIn = false;
glm::vec3 siPos = glm::vec3(5.5f, -1.5f, -25.0f);
glm::vec3 snPos = glm::vec3(9.0f, -1.0f, -15.0f);
float siScale = 4.0f;
float snScale = 0.0f; // Empieza invisible (escala 0)
```

Prototipos de callbacks y funciones de actualización.

Se hace la declaración anticipada de las funciones encargadas de procesar entrada de usuario (**KeyCallback**, **MouseCallback**), mover la cámara/objetos (**DoMovement**) y ejecutar la lógica de animaciones por frame (**Animation**).

```
void KeyCallback(GLFWwindow* window, int key, int scancode, int action,
int mode);
void MouseCallback(GLFWwindow* window, double xPos, double yPos);
void DoMovement();
void Animation();
```

Configuración de ventana y cámara.

Con **WINDOW** se definen las dimensiones fijas para el **framebuffer** y variables para capturar el tamaño real de la ventana.

Para la **cámara** se instala en la clase **Camera**, junto a variables para el cursor y seguimiento de movimiento.

```
const GLuint WIDTH = 1920, HEIGHT = 1080;
int SCREEN_WIDTH, SCREEN_HEIGHT;
Camera camera(glm::vec3(0.0f, 0.0f, 3.0f));
GLfloat lastX = WIDTH/2.0, lastY = HEIGHT/2.0;
bool keys[1024];           // Array de estados de teclas
bool firstMouse = true;
```

Luces y animaciones complementarias.

La luz puntual (**lightPos**, **active**) y flag de explosión global.

```
glm::vec3 lightPos(0.0f, 0.0f, 0.0f);
bool active;
bool explosionActive = false;
```

Límites y control de animación monitores.

Activación y límites para el movimiento de las transformaciones durante el cambio de los monitores mediante la explosión.

```
// Control de la animación
bool animacionActivada = false; // Animación inicialmente desactivada
float limiteInferior = 0.0f;
float limiteSuperior = 1.7;
float explosionFactor = 0.0f;
float implosionFactor = 0.0f;
bool implosionActive = false;
```

Visibilidad y estado de modelos individuales para gabinete.

Cada uno de los modelos para el armado de la nueva computadora durante su animación se contempla que cada uno tiene flags de visibilidad (...**Visible**), de animación activa (...**AnimActive**), fases (...**InPhase1**, ...**InPhase2**), parámetros de escala (...**ScaleFactor**), rotación (...**Rotation**, ...**TargetRotation**) y offsets (...**OffsetX**) para coordinar fases secuenciales.

```
// Control de visibilidad y animación de modelos
bool model1Visible = true;
bool model2Visible = false;
```

```

bool model2Appearing = false;
float model2ScaleFactor = 0.0f;
float model2Rotation = 0.0f;
float model2TargetRotation = 1080.0f; // 3 vueltas completas
const float escalaX = 2.0f;
const float escalaY = 3.2f;
const float escalaZ = 4.0f;
bool gaExplosionActive = false; // Para modelo 'ga'
bool gaVisible = true;
float gaExplosionFactor = 1.0f;
bool gaInflating = false;
bool gaContracting = false;
bool ganAppearing = false;
float ganRotation = 0.0f;
float ganTargetRotation = 270.0f;
float gOffsetX = -0.5f; // Agregar variables para animación del modelo g
bool gMovingOut = false;
bool gReturning = false;
bool gAnimActive = false;
bool gVisible = false; // Para modelo 'g'
bool gAppearing = false;
float gScaleFactor = 0.0f;
float gRotation = 0.0f;
float gTargetRotation = 180.0f;
bool grVisible = false; // Variables para modelo 'gr'
bool grAppearing = false;
float grScaleFactor = 0.0f;
float grRotation = 0.0f;
float grTargetRotation = 360.0f;
float grOffsetX = -0.5f;
bool grMovingOut = false;
bool grReturning = false;
bool grAnimActive = false;
bool ramVisible = false;
bool ramAppearing = false;
float ramScaleFactor = 0.0f;
bool grAppearingScale = false; // Ya existe `grAppearing` pero lo usamos para rotación, así que usamos otro para la escala
bool ramAnimActive = false;
bool ramMovingOut = false;
bool ramReturning = false;
float ramOffsetX = -0.5f;
static bool pasoExtraGr = false;
static bool pasoExtraRam = false;
bool animacionXCompleta = false;
bool gInPhase1 = false, gInPhase2 = false; // Estados para X

```

```

bool grInPhase1 = false, grInPhase2 = false;
bool ramInPhase1 = false, ramInPhase2 = false;
float ramRotation = 0.0f;
float ramTargetRotation = 360.0f;
bool prVisible = false;
bool prAnimActive = false;
bool prInPhase1 = false, prInPhase2 = false;
float prScaleFactor = 0.0f;
float prRotation = 0.0f;
float prOffsetX = -0.5f;
bool fuVisible = false; // Variables para modelo 'fu'
bool fuAnimActive = false;
bool fuInPhase1 = false, fuInPhase2 = false;
float fuScaleFactor = 0.0f;
float fuRotation = 0.0f;
float fuOffsetX = -0.5f;
float fuTargetRotation = 180.0f;
bool ganFinished = false;
bool gaAntesDeX = false;
bool ganVisible = false; // Antes estaba en true
float ganScaleFactor = 1.0f;
bool ganDisappearing = false;

```

Actualización global del laboratorio.

Este bloque agrupa una serie de flags y factores de escala que se usan para controlar, de forma global, la aparición, desaparición y el tamaño de dos pares de objetos ("pi" / "n" y dos "sillas especiales") antes de entrar en las animaciones más complejas del laboratorio.

```

//////// VARIABLES GLOBALES PARA ACTUALIZACIÓN GLOBAL////////
bool piVisible = true;
bool nVisible = false;
bool piMinimizando = false;
float piScaleFactor = 1.0f;
bool nApareciendo = false;
float nScaleFactor = 0.0f;
bool sillaEspecial1Minimizando = false;
bool sillaEspecial2Minimizando = false;
float escalaSillaEspecial1 = 4.0f;
float escalaSillaEspecial2 = 4.0f;
bool sillaEspecial1Visible = true;
bool sillaEspecial2Visible = true;

```

Variables de iluminación.

Se declaran las siguientes posiciones de las luces.

```
glm::vec3 pointLightPositions[] = { ///////////POSICIONES DE LUCES
    glm::vec3(0.0f, 8.3f, -55.0f),
    glm::vec3(-10.0f, 8.3f, -55.0f),
    glm::vec3(10.0f, 8.3f, -55.0f),
    glm::vec3(-10.0f, 8.3f, -30.0f),
    glm::vec3(0.0f, 8.3f, -30.0f),
    glm::vec3(10.0f, 8.3f, -30.0f) };
```

Carga y definición de modelo en OpenGL.

La carga y declaración de los modelos que ocupamos para el laboratorio virtual de computación gráfica, se realiza dentro de la instancia del inicio de la función principal, éstas quedan listas para su renderizado posterior en el bucle principal.

```
//models
Model Com((char*)"Models/Com/1.obj");
Model Com2((char*)"Models/Comp2/2.obj");
Model me((char*)"Models/Mes/me.obj");
Model ga((char*)"Models/Ga/ga.obj");
Model gan((char*)"Models/GabN/gan.obj");
Model g((char*)"Models/Graf/g.obj");
Model gr((char*)"Models/gr/gr.obj");
Model ram((char*)"Models/ram/ram.obj");
Model pr((char*)"Models/pr/pr.obj");
Model fu((char*)"Models/fu/fu.obj");
Model si((char*)"Models/silla/si.obj");
Model sn((char*)"Models/sn/sn.obj");
Model sal((char*)"Models/salon/salon2.obj");
...
```

Dibujar modelos en el ciclo de renderizado.

Dentro del bucle principal **while (!glfwWindowShouldClose(window))**, y tras configurar shaders y matrices (**view**, **projection**), cada modelo 3D se dibuja siguiendo un patrón muy similar que consta de tres etapas esenciales: **cálculo de la matriz de modelo**, configuración de los **uniforms** en el shader y ejecución del método **Draw()** del objeto. Entonces se sigue:

1. Se calcula la matriz **model** (traslación, rotación, escala) según la lógica de animación.
2. Se actualiza el **uniform** **model** en el shader:

3. Se llama a **model.Draw(shader)**.

4. Para las lámparas (luces puntuales) se usa el **lampShader**.

Para el modelo del salón se utiliza el obj y mtl creados a partir de blender, para tomar sus texturas correspondientes también de la carpeta **Models**.

Al seguir esta estructura tenemos como ejemplo la siguiente implementación de los modelos en el código fuente:

```
// ----- PIZARRON -----
    if (piVisible){
        glm::mat4 modelPi = glm::mat4(1.0f);
        modelPi = glm::translate(modelPi, glm::vec3(-18.0f,
3.5f, -40.0f));
        modelPi = glm::rotate(modelPi, glm::radians(180.0f),
glm::vec3(0.0f, 1.0f, 0.0f));
        modelPi = glm::scale(modelPi, glm::vec3(10.5f, 8.0f,
10.5f) * piScaleFactor);
        glUniform1f(glGetUniformLocation(lightingShader.Program,
"explosionFactor"), 0.0f);
        glUniform1i(glGetUniformLocation(lightingShader.Program,
"transparency"), 0);
        glUniformMatrix4fv(glGetUniformLocation(lightingShader.Program,
"model"), 1, GL_FALSE, glm::value_ptr(modelPi));
        pi.Draw(lightingShader); }
    if (nVisible) // PIZARRON NUEVO {
        glm::mat4 modelN = glm::mat4(1.0f);
        modelN = glm::translate(modelN, glm::vec3(-18.0f,
3.5f, -40.0f));
        modelN = glm::rotate(modelN, glm::radians(0.0f),
glm::vec3(0.0f, 1.0f, 0.0f));
        modelN = glm::scale(modelN, glm::vec3(5.5f, 15.0f,
17.5f) * nScaleFactor); // escala animada
        glUniform1f(glGetUniformLocation(lightingShader.Program,
"explosionFactor"), 0.0f);
        glUniform1i(glGetUniformLocation(lightingShader.Program,
"transparency"), 0);
        glUniformMatrix4fv(glGetUniformLocation(lightingShader.Program,
"model"), 1, GL_FALSE, glm::value_ptr(modelN));
        N.Draw(lightingShader); }
```

Este flujo asegura que cada modelo se renderice con su transformación y material correspondientes en cada frame.

Además de tener el orden del laboratorio de computación gráfica, a lo más cercano a la realidad.

Illuminación.

Como se mencionó en la parte del **Shader lighting.frag** necesitamos tener en cuenta la definición de las 6 luces puntuales para que sean colocadas en su posición en el techo del salón, dónde cada hilera será manipulada desde teclado para poder apagar y prender dichas luces.

```
#version 330 core
#define NUMBER_OF_POINT_LIGHTS 6
```

Dentro del código fuente PROYECTO EQUIPO 11.cpp la definición de las luces se realiza a partir del dibujo de las lámparas en el techo en cada posición de luz puntual, para visualizar dónde están colocadas en la escena.

Nos aseguramos que el modelo “LAM” no sufra explosiones ni transparencias. Dentro del ciclo manejamos las posiciones de las luces mediante un array **pointLightPositions[]** con las coordenadas donde se sitúan cada luz puntual.

- Translate: Sitúa el origen del modelo donde está la luz puntual en el techo.
- Rotate: Ajusta la orientación.
- Scale: Adapta el tamaño para que sea visible y proporcione una referencia del área iluminada.

```
lampShader.Use(); // Configuración del shader de Lámparas
modelLoc = glGetUniformLocation(lampShader.Program,
"model");
viewLoc = glGetUniformLocation(lampShader.Program, "view");
projLoc = glGetUniformLocation(lampShader.Program,
"projection");
glUniformMatrix4fv(viewLoc, 1, GL_FALSE,
glm::value_ptr(view));
glUniformMatrix4fv(projLoc, 1, GL_FALSE,
glm::value_ptr(projection));
 glBindVertexArray(VAO); // Los cubitos usan su propio VAO
for (int i = 0; i < 6; ++i) {
    glm::mat4 model = glm::mat4(1.0f);
    model = glm::translate(model, pointLightPositions[i]);
    model = glm::scale(model, glm::vec3(0.2f)); //Tamaño
    glUniformMatrix4fv(modelLoc, 1, GL_FALSE,
glm::value_ptr(model));
    glDrawArrays(GL_TRIANGLES, 0, 36); }
lightingShader.Use();
glUniform1f(glGetUniformLocation(lightingShader.Program,
"explosionFactor"), 0.0f);
glUniform1i(glGetUniformLocation(lightingShader.Program,
```

```

"transparency"), 0);
    modelLoc = glGetUniformLocation(lightingShader.Program,
"model");
    for (int i = 0; i < 6; ++i) {
        glm::mat4 model = glm::mat4(1.0f);
        model = glm::translate(model, pointLightPositions[i]);
        // (ángulo en radianes)
        model = glm::rotate(model, glm::radians(90.0f),
glm::vec3(1.0f, 0.0f, 0.0f));
        model = glm::rotate(model, glm::radians(90.0f),
glm::vec3(0.0f, 1.0f, 0.0f));
        model = glm::scale(model, glm::vec3(10.5f, 10.0f,
5.5f)); // Escalado final
        glUniformMatrix4fv(modelLoc, 1, GL_FALSE,
glm::value_ptr(model));
        LAM.Draw(lightingShader); }
    glBindVertexArray(0); // Libera el VAO

```

Illuminación.	Imagen.
Luz proyector y con las 6 luces puntuales del laboratorio.	
Solo luz reflectora del proyector.	

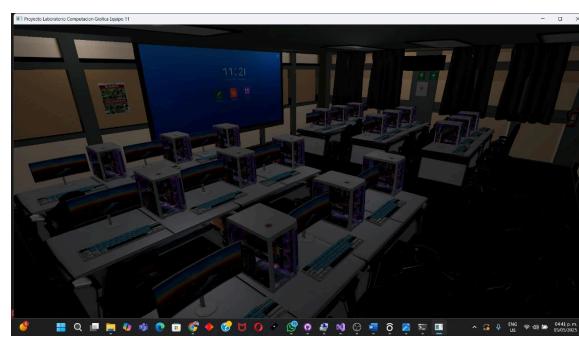
Luces puntuales del nuevo salón, solo luces de la primera hilera.	
Luces apagadas para el salón nuevo del laboratorio de computación gráfica.	

Tabla 5. Tipos de iluminación aplicados.

Módulos extra.

Función que evita colisiones.

Esta función principalmente realiza la comprobación de solapamientos en el plano de suelo (eje XZ) entre la nueva posición propuesta para una silla y las posiciones actuales de las demás sillas (tanto la versión antigua como la nueva), ignorando la propia silla, y devuelve **true** si alguna queda a menos de un umbral de separación (por ejemplo, 2 unidades), evitando así que dos animaciones de sillas coincidan en el mismo espacio.

Su funcionamiento está compuesto por una rutina de detección, contemplando lo siguiente:

1. Función: Dónde **nuevaPos** la posición candidata (X, Y, Z) a la que la silla en movimiento (**actual**) quiere desplazarse. Para que **actual** haga referencia a la propia silla que se está animando, para no compararla consigo misma.

2. Recorre todas las sillas: Con el ciclo **for** se asume un vector global **std::vector<SillaAnimada> silllas**; que contiene todas las sillas animadas. Para que en el bucle se examine cada instancia de **silla** para saber si existe una colisión en la posición propuesta.
3. Ignora la comparación de sí misma:
4. Comprobar versiones de posición:
5. Resultado: Si cualquier otra silla, ya sea en su versión vieja o nueva, está a menos de 2.0 unidades de la posición propuesta, la función devuelve **true** (colisión detectada). Por otro lado, si el bucle termina sin encontrar colisiones, devuelve **false** (movimiento seguro).

```
// Rutina para detección de colisiones entre animaciones
bool hayColision(glm::vec3 nuevaPos, const SillaAnimada& actual)
{
    for (const auto& silla : silllas)
    {
        if (&silla == &actual) continue; // Ignora la propia silla
        if (silla.siVisible &&
            glm::distance(glm::vec2(silla.siPos.x, silla.siPos.z),
            glm::vec2(nuevaPos.x, nuevaPos.z)) < 2.0f)
            return true;
        if (silla.snVisible &&
            glm::distance(glm::vec2(silla.snPos.x, silla.snPos.z),
            glm::vec2(nuevaPos.x, nuevaPos.z)) < 2.0f)
            return true;
    }
    return false;
}
```

Animaciones complejas.

En la siguiente sección se detalla la creación y funciones de las animaciones que se implementan para el cambio de mobiliario del laboratorio de computación gráfica.

Estructura e instancia de animación.

Bloque de creación de instancias de animación para PC's y transiciones de sillas, se define dentro de la función **Main** antes del ciclo principal de renderizado.

Se agregan múltiples objetos animados ("PC's") al vector 'animaciones', cada uno con posiciones distribuidas en filas y columnas, usando la estructura **InstanciaAnimacion**.

Las tres últimas instancias marcan objetos estáticos (sin translación ni escala dinámica), activando solo animaciones internas. De manera similar, el vector 'sillas' se rellena con posiciones para cada silla animada, creando un patrón uniforme en el espacio 3D.

Esta configuración define el layout inicial de todos los elementos que luego se procesarán en el ciclo de renderizado y animación. Se manda a llamar las características de acuerdo a la posición.

```
// ----- DIBUJOS DE PC's (Instancias Animación)
// Fila 1: tres PCs con efecto GA visible al inicio
animaciones.emplace_back(glm::vec3( 3.5f,  0.5f, -25.8f));
animaciones.emplace_back(glm::vec3( 3.5f,  0.5f, -30.3f));
animaciones.emplace_back(glm::vec3( 3.5f,  0.5f, -34.8f));
// Fila 2: columna izquierda
en animaciones.emplace_back(glm::vec3(-3.5f,  0.5f, -25.8f));
animaciones.emplace_back(glm::vec3(-3.5f,  0.5f, -30.3f));
animaciones.emplace_back(glm::vec3(-3.5f,  0.5f, -34.8f));
// Fila 3: columna más a la izquierda
en animaciones.emplace_back(glm::vec3(-10.5f,  0.5f, -25.8f));
animaciones.emplace_back(glm::vec3(-10.5f,  0.5f, -30.3f));
animaciones.emplace_back(glm::vec3(-10.5f,  0.5f, -34.8f));
// Fila 4: segunda sección de PCs más atrás
animaciones.emplace_back(glm::vec3( 3.5f,  0.5f, -48.5f));
animaciones.emplace_back(glm::vec3( 3.5f,  0.5f, -56.5f));
...
// Instancias estáticas (solo animaciones internas sin mover/escala externa)
animaciones.emplace_back(glm::vec3( 4.0f, -2.3f, -59.0f), true);
animaciones.emplace_back(glm::vec3(-3.0f, -2.3f, -59.0f), true);
animaciones.emplace_back(glm::vec3(-10.0f, -2.3f, -59.0f), true);
// ----- DIBUJOS DE SILLAS (Instancias SillaAnimada) -----
// Bloques de sillas en tres filas frontales
sillas.emplace_back(glm::vec3( 7.0f, -1.5f, -28.0f));
sillas.emplace_back(glm::vec3( 7.0f, -1.5f, -23.0f));...
// Bloques de sillas en segunda sección más atrás
sillas.emplace_back(glm::vec3( 7.0f, -1.5f, -58.5f));...
```

Animación de monitores.

La animación de las pantallas es un ciclo de **explosión** (encogimiento y elevación del modelo 1), seguido de **aparición** (crecimiento giratorio del modelo 2), y opcionalmente implosión para volver al estado original.

Todos los pasos usan interpolación lineal basada en **deltaTime** y flags booleanos que determinan qué fases están activas en cada momento.

```

bool model1Visible = true; Control de visibilidad y animación de modelos
bool model2Visible = false;
bool model2Appearing = false;
float model2ScaleFactor = 0.0f;
float model2Rotation = 0.0f;
float model2TargetRotation = 1080.0f; // 3 vueltas completas
const float escalaX = 2.0f;
const float escalaY = 3.2f;
const float escalaZ = 4.0f;

```

En el ciclo de renderizado se posiciona cada pantalla (**posicionesPantallas**) en el mundo, escalando 2.5x y haciendo una rotación de 90° alrededor del eje Y para orientarla correctamente.

Para **Model 1**, se hace la animación de explosión si la **condición** es sólo mientras **model1Visible** sea **true** y **explosionFactor < 1.7**.

- **Escala:** **explosionScale** decrece linealmente de 1.0 a 0.0 conforme **explosionFactor** va de 0.0 a 1.7.
- **Desplazamiento vertical:** la pantalla “sube” ligeramente para simular que se aleja hacia arriba a medida que se encoge.
- **Rotación fija de 270°** para mantener la orientación deseada durante la animación.



Figura 51. Animación de explosión de monitores.

Para **Model 2**, se tiene la animación de aparición donde:

- **Condición:** **model2Visible == true**.

- **Rotación:** va aumentando hasta un ángulo objetivo (*model2TargetRotation*).
- **Escala:** de 0 a 1 conforme progresó la fase de aparición.
- Se desactiva cualquier efecto de explosión o transparencia para este modelo.

```
/////////-PANTALLAS-----///////////
    for (const auto& pos : posicionesPantallas)
    { // 1. Transformación base del "grupo" de la pantalla
        glm::mat4 groupTransform = glm::mat4(1.0f);
        groupTransform = glm::translate(groupTransform, pos);
        groupTransform = glm::scale(groupTransform,
glm::vec3(2.5f));
        groupTransform = glm::rotate(groupTransform,
glm::radians(90.0f), glm::vec3(0.0f, 1.0f, 0.0f));
        // MODEL 1 - Explosión Animación
        if (model1Visible && explosionFactor < 1.7f){
            glm::mat4 explodedModel = glm::mat4(1.0f);
            float explosionScale=1.0f-explosionFactor/1.7f;
            float baseOffset = 0.8f;
            float verticalCompensate=baseOffset*(1.0f-
explosionScale);
            explodedModel = glm::translate(explodedModel,
glm::vec3(0.0f, verticalCompensate, 0.0f));
            explodedModel = glm::scale(explodedModel,
glm::vec3(explosionScale));
            explodedModel = glm::rotate(explodedModel,
glm::radians(270.0f), glm::vec3(0.0f, 1.0f, 0.0f));
            glm::mat4 finalModel1 = groupTransform *
explodedModel;
            glUniformMatrix4fv(modelLoc, 1, GL_FALSE,
glm::value_ptr(finalModel1));
            Com.Draw(lightingShader); }
        if (model2Visible) // MODEL 2 - Aparición animada {
            glm::mat4 model2 = glm::mat4(1.0f);
            model2 = glm::rotate(model2,
glm::radians(model2Rotation), glm::vec3(0.0f, 1.0f, 0.0f));
            model2 = glm::scale(model2,
glm::vec3(model2ScaleFactor));
            glm::mat4 finalModel2 = groupTransform * model2;
            glUniform1f(glGetUniformLocation(lightingShader.Program,
"explosionFactor"), 0.0f);
            glUniform1i(glGetUniformLocation(lightingShader.Program,
"transparency"), 0);
            glUniformMatrix4fv(modelLoc, 1, GL_FALSE,
```

```

glm::value_ptr(finalModel2));
Com2.Draw(lightingShader); } }

```

La lógica de actualización en **void Animation()** se basa en los siguientes aspectos:

1. Explosión: Donde se incrementa **explosionFactor** hasta 1.7, para que al llegar, oculta el primer modelo y prepara el segundo para aparecer.
2. Aparición: Aumenta el **model2ScaleFactor** de 0→1 y rota 360° por segundo hasta llegar al ángulo meta. Después desactiva la animación cuando alcanza el objetivo.
3. Implosión: Con la tecla **M** se “rebobina” la explosión: resetea todos los factores y visibilidades a su estado inicial.

```

if (explosionActive && explosionFactor < 1.7f) { 1) Explosión (tecla N)
    explosionFactor += deltaTime * 0.2f; // velocidad de contracción
    if (explosionFactor >= 1.7f) { // a) Finalizar explosión
        explosionFactor = 1.7f;
        explosionActive = false;
        model1Visible = false; // ocultar model1
        model2Visible = true; // preparar model2
        model2Appearing = true;
        model2ScaleFactor= 0.0f;
        model2Rotation = 0.0f; } }

if (model2Appearing) { // 2) Aparición de model2
    model2ScaleFactor += deltaTime * 1.0f; // escala de 0 → 1
    if (model2ScaleFactor > 1.0f) model2ScaleFactor = 1.0f;
    model2Rotation += deltaTime * 360.0f; // rotación continua
    if (model2Rotation >= model2TargetRotation) {
        model2Rotation = model2TargetRotation;
        model2Appearing = false; // fase concluida } }

if (implosionActive) { // 3) Implosión (tecla M)
    implosionFactor += deltaTime * 2.0f; velocidad de expansión inversa
    if (implosionFactor >= 1.7f) { // a) Restaurar estado inicial
        implosionFactor = 0.0f;
        explosionFactor = 0.0f;
        implosionActive = false;
        model1Visible = true;
        model2Visible = false;
        model2Appearing = false;
        model2ScaleFactor= 0.0f;
        model2Rotation = 0.0f; } }

```

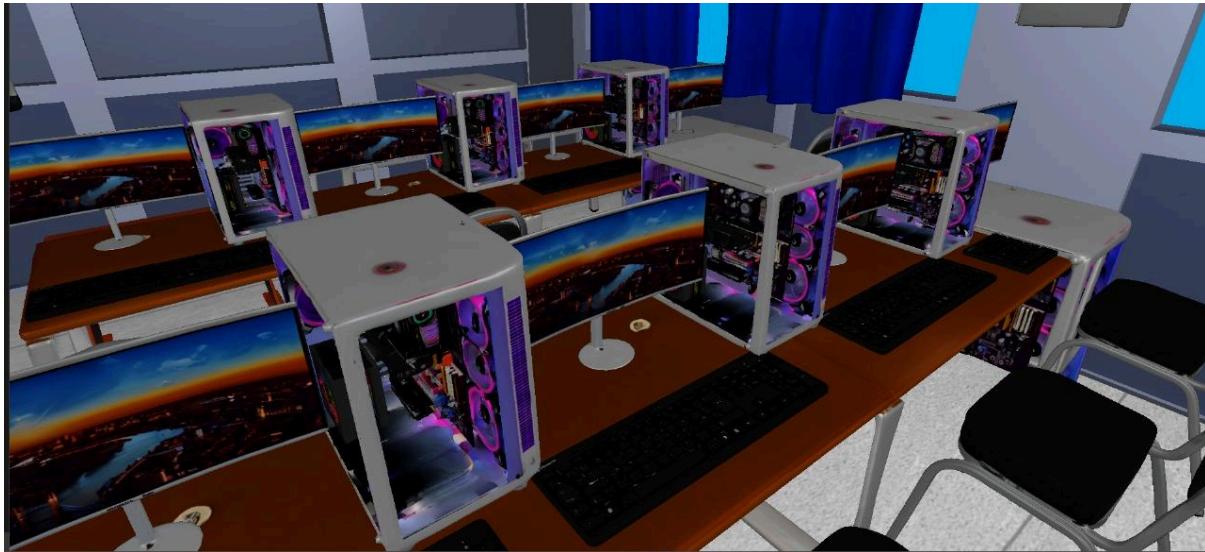


Figura 52. PC remodeladas posterior a la animación de ensamblaje.

Animación de armado PC.

La animación de “armado” de la PC se estructura en varias etapas encadenadas, cada una controlada por flags y valores de transformación que definen aparición, rotación, desplazamiento y finalmente caída y escalado globales.

Los parámetros globales iniciales definen la posición elevada, escala y rotación de todo el conjunto de componentes, así como las velocidades de bajada y de reducción de escala al final.

- Inicio: todo el conjunto está 7 unidades por encima del suelo y escalado 4x.
- Final: tras animar cada componente, se activa **reduciendoEscala** para volver a escala 1x y luego **bajandoY** para mover el grupo hasta **y=0.5**.

```
glm::vec3 globalTranslationX = {0, 7, -25.8};
glm::vec3 globalScaleX      = {4, 4, 4};
float    globalRotationX   = 90.0f;
bool    bajandoY          = false;
float  velocidadBajadaY = 4.0f;                      // Ajustable
bool    reduciendoEscala = false;
float  escalaTemporal   = 4.0f;
```

Con la fase **GA** se tiene la explosión inicial donde se muestra un modelo “ga” (caja madre) que explota (se contrae) al pulsar la tecla adecuada.

En la parte de void Animation, si **gaContracting** está activo, **gaExplosionFactor** va a decrementar a velocidad fija. Por otro lado, al llegar a 0, se oculta “ga”

(*gaVisible=false*) y se disparan los flags para la siguiente fase (*ganVisible=true*, *ganAppearing=true*).

```
bool gaVisible      = true;
bool gaExplosionActive = false;
float gaExplosionFactor= 1.0f;
bool gaInflating    = false;
bool gaContracting = false;
```

De la misma forma que en la sección de variables globales se tienen flags y variables por **Submodelo**.

Para cada componente del PC (ga, gan, g, gr, ram, pr, fu) existen variables globales (y equivalentes dentro de cada *InstanciaAnimacion*) que siguen el mismo patrón:

Modelo	Visibilidad	Activación	Fase 1	Fase 2	Parámetros clave
ga (intro)	gaVisible	gaContracting	—	—	gaExplosion Factor
gan (gabinete)	ganVisible	ganAppearing	escala 0→1	rot 0→270°	ganScaleFactor, ganRotation
g (GPU)	gVisible	gAnimActive	escala 0→1 + rot 0→180°	slide -0.5→0 X	gScaleFactor, gRotation, gOffsetX
gr (gráfica)	grVisible	grAnimActive	escala 0→1 + rot 0→360°	slide -0.5→0 X	iguales a "g" con gr*
ram	ramVisible	ramAnimActive	escala 0→1 + rot 0→360°	slide -0.5→0 X	ram*
pr (CPU)	prVisible	prAnimActive	escala 0→1 + rot 0→360°	slide -0.5→0 X	pr*
fu (fuente)	fuVisible	fuAnimActive	escala 0→1 + rot 0→180°	slide -0.5→0 X	fu*

Tabla 6. Modelos y comportamiento animado.

- Visibilidad (***Visible**) indica si se dibuja el modelo.
- Activación (***Appearing ó *AnimActive**) inicia la lógica de animación.

- Fase 1: incremento de escala y giro simultáneos hasta alcanzar el valor objetivo.
- Fase 2: desplazamiento en X desde fuera (**offsetX = -0.5**) hasta posición final (0).

Cada vez que una fase 1 acaba (**rotación ≥ objetivo**), se desactiva esa bandera y se activa la fase 2. Cuando la fase 2 acaba (**offsetX ≥ 0**), se pasa al siguiente modelo en la cadena.

La secuencia en **Animation()**, dentro del bucle de actualización, para cada **InstanciaAnimacion** de tu vector animaciones:

1. GA: si **gaContracting**, decrementa **gaExplosionFactor**. Al llegar a 0: **gaVisible = false** y **ganVisible = true**, **ganAppearing = true**.
2. GAN: mientras **ganAppearing**: Aumenta **ganScaleFactor** (0→1) y **ganRotation** (0→270°), y Al cumplirse ambos, **ganFinished = true**.
3. G: si **ganFinished** activa **gAnimActive** y **gInPhase1**.
 - Fase 1: escala 0→1 y rotación 0→180° → al terminar, **gInPhase2**.
 - Fase 2: offsetX -0.5→0 → al terminar, activa **grAnimActive**.
4. GR: misma lógica que "G" pero con rotación 360° → luego activa **ramAnimActive**.
5. RAM: idéntico a "GR" → luego activa **prAnimActive**.
6. PR: idéntico a "G/GR" con rot 360° → luego activa **fuAnimActive**.
7. FU: escala 0→1 + rot 0→180° → luego offsetX -0.5→0 → al terminar, **reduciendoEscala = true**.
8. Escalado global (**reduciendoEscala**): instancia.escala se mezcla hasta (2,2,2).
9. Descenso (**bajandoY**): **instancia.translationY** baja de 7 → 0.5.

```
-----ANIMACIÓN DE PC'S-----///
for (InstanciaAnimacion& instancia : animaciones) {
    if (instancia.animacionEstatica && !instancia.gaContracting
&& !instancia.ganAppearing && !instancia.gVisible) {
        continue; }
    if (instancia.gaVisible && instancia.gaContracting)
    {
        ...
        // Solo si estaba arriba, sube a Y=7.0 para animar
        if (!instancia.animacionEstatica)
```

```

        instancia.translationY =
globalTranslationYInicial;
    }
}
if (instancia.posicion.x == 4.0f) {
    // Desactiva el cambio de escala y bajada
    instancia.reduciendoEscala = false;
    instancia.bajandoY = false; }
if (instancia.ganAppearing) { // Aparece GAN
    if (instancia.ganScaleFactor < 1.0f) {
        instancia.ganScaleFactor += deltaTime * 0.5f;
    ...
// Activar G después de GAN
if (instancia.ganFinished && !instancia.gAnimActive) {
    instancia.gVisible = true;
    instancia.gScaleFactor = 0.0f;
    instancia.gRotation = 0.0f;
    instancia.gOffsetX = -0.5f;
    instancia.gAnimActive = true;
    instancia.gInPhase1 = true;
    instancia.ganFinished = false; }
if (instancia.gAnimActive) { // Animación de G
    if (instancia.gInPhase1) {
        instancia.gScaleFactor += deltaTime * 1.5f;
        if (instancia.gScaleFactor > 1.0f)
instancia.gScaleFactor = 1.0f;
        instancia.gRotation += deltaTime * 180.0f;
    ...
// Activar GR
    instancia.grVisible = true;
    instancia.grScaleFactor = 0.0f;
    instancia.grRotation = 0.0f;
    instancia.grOffsetX = -0.5f;
    instancia.grAnimActive = true;
    instancia.grInPhase1 = true; } } }
if (instancia.grAnimActive) { // Animación de GR
    if (instancia.grInPhase1) {
        instancia.grScaleFactor += deltaTime * 1.5f;
        if (instancia.grScaleFactor > 1.0f)
instancia.grScaleFactor = 1.0f;
        instancia.grRotation += deltaTime * 180.0f;
        if (instancia.grRotation >=
instancia.grTargetRotation) {
            instancia.grRotation =
instancia.grTargetRotation;
            instancia.grInPhase1 = false;
            instancia.grInPhase2 = true; } } }

```

```

        else if (instancia.grInPhase2) {
            instancia.grOffsetX += deltaTime * 1.0f;
            ...
            instancia.ramVisible = true; // Activar RAM
            instancia.ramScaleFactor = 0.0f;
            instancia.ramRotation = 0.0f;
            instancia.ramOffsetX = -0.5f;
            instancia.ramAnimActive = true;
            instancia.ramInPhase1 = true; } } }

        if (instancia.ramAnimActive) { // RAM
            if (instancia.ramInPhase1) {
                instancia.ramScaleFactor += deltaTime * 1.5f;
                if (instancia.ramScaleFactor > 1.0f)
instancia.ramScaleFactor = 1.0f;
                instancia.ramRotation += deltaTime * 180.0f;
                if (instancia.ramRotation >=
instancia.ramTargetRotation) {
                    instancia.ramRotation =
instancia.ramTargetRotation;
                    instancia.ramInPhase1 = false;
                    instancia.ramInPhase2 = true; } }

            ...
            if (instancia.prAnimActive) { // PR
                if (instancia.prInPhase1) {
                    instancia.prScaleFactor += deltaTime * 1.5f;
                    if (instancia.prScaleFactor > 1.0f)
instancia.prScaleFactor = 1.0f;
                    instancia.prRotation += deltaTime * 180.0f;
                    if (instancia.prRotation >= 360.0f) {
                        instancia.prRotation = 360.0f;
                        ...

```

Cuando todas las instancias han completado estos pasos, puedes marcar **animacionXCompleta = true** para suspender más actualizaciones.

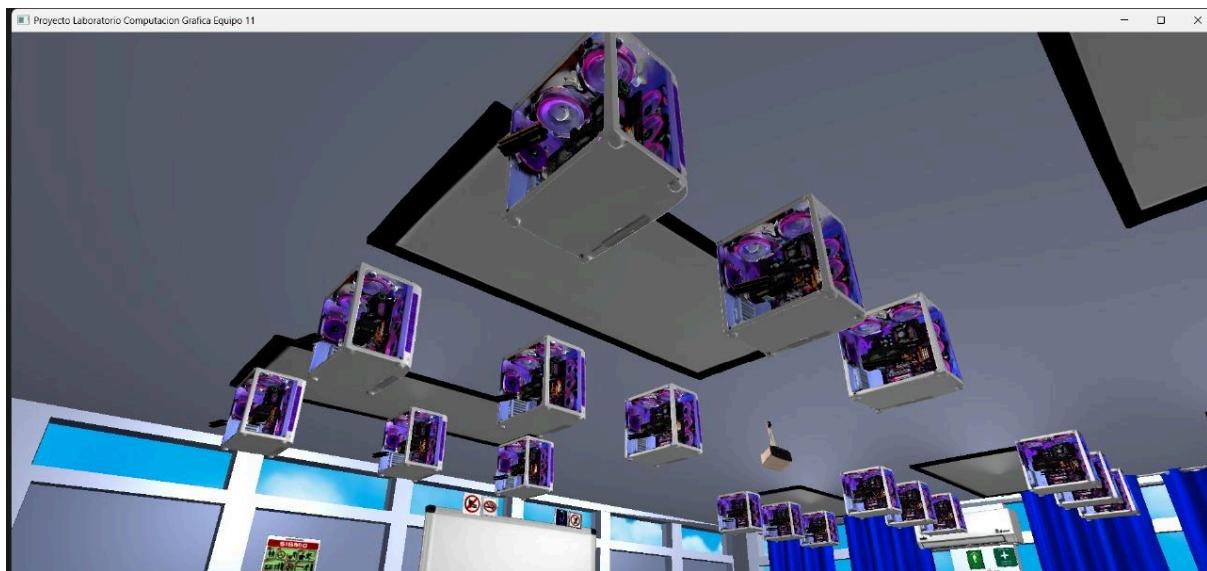


Figura 53. Animación de ensamblaje de PC

¿Por qué funciona así?

- Encadenamiento: cada submodelo sólo empieza cuando el anterior ha terminado, gracias a los flags ***Finished**, ***InPhase2** y el modo “estático” (**animacionEstatica**) que omite instancias ya montadas.
- Interpolación continua: todas las transiciones usan **deltaTime** para mantenerse independientes de la tasa de frames.
- Modularidad: mover cada conjunto de variables dentro de **InstanciaAnimacion** permite instanciar múltiples PCs que se animen en paralelo o en secuencia, sin colisiones de estado global.



Figura 54. Modelos colocados en su orden adecuado.

Keyframe y animación de sillas.

Para orquestar la transición entre la silla original (**si**) y la “nueva” (**sn**), se usa un conjunto de flags y transformaciones que controlan cuatro fases:

1. Inicio se realiza con **animSiActive = true**, donde la silla original (**si**) se dibuja y puede moverse. **siPos** y **siScale** definen su ubicación y tamaño al empezar.
2. Fase “Mover hacia fuera” (**siMovingOut**), se activa cuando quieras desplazar **si** fuera de la vista (por ejemplo, al pulsar ‘P’). Cada frame, en **Animation()**, incrementas o decrementas **siPos** a lo largo de uno de los ejes usando **deltaTime**, hasta que la silla sale del “cuadro”. Al acabar, desactivas **siMovingOut** y activas **siShrinking**.
3. Fase “Encoger” (**siShrinking**): Con **siShrinking = true**, cada frame se reduce **siScale de 4.0f → 0.0f** a velocidad controlada (**deltaTime * velocidad**). Esto da la sensación de que la silla “se aleja” o desaparece en el espacio. Al llegar a escala 0, fijas **siScale = 0** y **siShrinking = false**, y entonces **animSiActive = false**, y activas **animSnActive = true** y **snAppearing = true**.
4. Fase “Aparición” de la silla nueva (**snAppearing**): Con **snAppearing = true**, incrementas **snScale de 0.0f → 4.0f** cada frame, esto hace que la silla “aparezca” creciendo desde un punto invisibles. Al alcanzar **snScale = 4.0f**, fijas **snScale = 4.0f**, **snAppearing = false** y activas **snMovingIn**.
5. Fase “Mover hacia dentro” (**snMovingIn**): Ahora se desplaza **snPos** desde una posición inicial hacia su destino final (**glm::vec3(9.0f, -1.0f, -15.0f)**) interpolando posiciones con **deltaTime**. Cuando **snPos** coincide con el destino, desactivas **snMovingIn**. La animación queda completa, y **animSnActive = false**.

Para el uso e implementación del keyframe que controla la animación y las variables de las sillas, tenemos que:

- **time**: marca el instante clave de la animación.
- **position**: posición objetivo de la silla en ese keyframe.
- **scale**: escala objetivo (cómo de grande o pequeña se dibuja) en ese keyframe.
- Con varios Keyframe ordenados por time, se define la trayectoria y el “crecer/encoger” de la silla a lo largo del tiempo.

```

struct Keyframe {
    float time;           // Momento relativo (en segundos) desde el inicio
    glm::vec3 position; // Posición de la silla en ese instante
    float scale;         // Factor de escala de la silla en ese instante
};

```

Para la estructura **SillaKeyframeAnimation** se tiene que:

- **keyframes**: secuencia temporal de Keyframe que define la animación.
- **currentTime**: acumula **deltaTime** cada frame.
- **currentIndex**: señala el par de **keyframes** entre los que interpolar.
- **active**: solo si es true, **update()** avanzará y se recalculará **interpolatedPosition/Scale**.
- **start()**: reinicia la animación desde cero.
- **update(deltaTime)**:
 1. Avanza **currentTime**.
 2. Determina en qué segmento de **keyframes** estamos (busca el siguiente time).
 3. Si estamos al final, desactiva **active**.
 4. Interpolación lineal (**glm::mix**) de posición y escala entre los dos keyframes actuales.

```

struct SillaKeyframeAnimation {
    std::vector<Keyframe> keyframes;           // Lista de momentos clave
    float currentTime = 0.0f;                     // Tiempo transcurrido desde el
    inicio
    bool active = false;                         // ¿Está animando ahora?
    int currentIndex = 0;                         // Índice del keyframe anterior
    al tiempo actual
    glm::vec3 interpolatedPosition;             // Resultado de la interpolación
    cada frame
    float interpolatedScale;                    // Resultado de la interpolación
    cada frame
    void start() {
        currentTime = 0.0f;
        currentIndex = 0;
        active = true;
    }
    void update(float deltaTime) {
        if (!active || keyframes.size() < 2) return;
    }
};

```

```

currentTime += deltaTime;
// Avanzar al segmento correcto de keyframes
while (currentIndex < (int)keyframes.size() - 1 &&
       currentTime > keyframes[currentIndex + 1].time) {
    currentIndex++;
}
// Si ya llegamos al último keyframe, terminar
if (currentIndex >= (int)keyframes.size() - 1) {
    active = false;
    return;
}
// Interpolan entre keyframes[currentIndex] y [currentIndex+1]
Keyframe& kf1 = keyframes[currentIndex];
Keyframe& kf2 = keyframes[currentIndex + 1];
float localTime = currentTime - kf1.time;
float duration = kf2.time - kf1.time;
float t = localTime / duration; // Normalizado
interpolatedPosition = glm::mix(kf1.position, kf2.position, t);
interpolatedScale = glm::mix(kf1.scale, kf2.scale, t);
}
};


```

En la estructura **SillaAnimada**, las variables realizan:

- **siPos / siScale**: posición y escala actuales de la silla original.
- **snPos / snScale**: posición y escala de la nueva silla.
- **siVisible, snVisible**: controlan si se dibujan.
- **siMoving, siShrinking, snAppearing, snReturning**: flags de fase interna, usadas en el bucle Animation() para avanzar la lógica.
- **animación**: la instancia de **SillaKeyframeAnimation** usada para interpolar movimientos complejos.

Como se define desde las **INSTANCIAS**, se colocan las sillas en posiciones fijas y luego en cada frame, dibuja la “silla original” (**si**) o la “silla nueva” (**sn**) según sus flags de visibilidad y sus parámetros de animación.

- **emplace_back(glm::vec3(x,y,z))** construye “in-place” cada **SillaAnimada**, usando su constructor que fija **siPos = start, siScale = 4.0f, snPos fuera de cámara y snScale = 0.0f**.
- El patrón de coordenadas crea una rejilla de sillas en las tres columnas X = {7, 0, -7.5} y varias filas de Z, tanto cerca (-23...-33) como más lejos (-46...-58).

```

sillas.emplace_back(glm::vec3( 7.0f, -1.5f, -28.0f));
sillas.emplace_back(glm::vec3( 7.0f, -1.5f, -23.0f));

```

```
sillas.emplace_back(glm::vec3( 7.0f, -1.5f, -33.0f));
// ...y así sucesivamente para cubrir tres filas y tres columnas,
// luego otras tres filas más abajo (más negativo Z), etc.
```

El dibujo de Renderizado dentro del bucle principal, realiza:

1. Transformaciones:
 - **Translate:** mueve el modelo a la posición actual (**siPos o snPos**).
 - **Rotate:** gira 180° en Y para alinear la silla con tu sistema de coordenadas.
 - **Scale:** aplica el factor dinámico (**siScale** decreciente durante el “encogimiento”, **snScale** creciente durante la aparición).
2. Flags de visibilidad: **siVisible** controla si se dibuja la silla original. **snVisible** hace lo propio con la silla nueva.
3. Parámetros de animación: Para la silla original, además pasas al shader **explosionFactor** (si la silla “explota” o se contrae). La silla nueva no usa ese parámetro; simplemente se dibuja con su modelo (**sn.Draw**).

```
for (const auto& silla : sillas) {
    // 2.1 Dibujo de la silla original ('si')
    if (silla.siVisible) {
        glm::mat4 modelSi = glm::mat4(1.0f);
        modelSi = glm::translate(modelSi, silla.siPos);
        modelSi = glm::rotate(modelSi, glm::radians(180.0f),
                              glm::vec3(0.0f, 1.0f, 0.0f));
        modelSi = glm::scale(modelSi, glm::vec3(silla.siScale));
        glUniformMatrix4fv(modelLoc, 1, GL_FALSE,
                           glm::value_ptr(modelSi));
        // Pasa el factor de "explosión" si aplicaste ese efecto
        glUniform1f(glGetUniformLocation(lightingShader.Program,
                                         "explosionFactor"),
                    silla.siExplosionFactor);
        si.Draw(lightingShader);
    }
    // 2.2 Dibujo de la silla nueva ('sn')
    if (silla.snVisible) {
        glm::mat4 modelSn = glm::mat4(1.0f);
        modelSn = glm::translate(modelSn, silla.snPos);
        modelSn = glm::rotate(modelSn, glm::radians(180.0f),
                              glm::vec3(0.0f, 1.0f, 0.0f));
        modelSn = glm::scale(modelSn, glm::vec3(silla.snScale));
        glUniformMatrix4fv(modelLoc, 1, GL_FALSE,
                           glm::value_ptr(modelSn));
    }
}
```

```

        sn.Draw(lightingShader);
    }
}

```

Dentro de la función ***Animation()*** cada silla recorre estas etapas secuenciales, controladas por el **enum FaseAnimSilla** y un **switch** sobre **silla.fase**:

1. ***EscalandoAntesDeMover***: Se reduce la escala inicial grande (4.0f) de la silla original hasta su tamaño "real" (1.0f). Esto se hace con **glm::mix** ya que interpola linealmente (**t = deltaTime**) cada frame y pasa a la fase ***SubirAntesDeMover***.
2. ***SubirAntesDeMover***: Se eleva la silla original 4 unidades en Y (simular "levantarla" antes de desplazarla). Se hace una interpolación de la componente Y con **glm::mix**. Su transición se hace al alcanzar la altura deseada, configuras dos keyframes (desde la posición elevada hasta **targetPos**) y arrancas el **SillaKeyframeAnimation**, cambiando a ***MoverASalida***.
3. ***MoverASalida***: se desplaza la silla desde la posición elevada hasta la "salida" (**targetPos**) usando keyframes. Ya que en cada frame se llama a **animacion.update**, se lee **interpolatedPosition** y actualiza **siPos**. La transición se hace cuando la animación keyframe termina (**active == false**), activas el efecto de "explosión" de la silla y pasa a ***DesaparecerSi***.
4. ***DesaparecerSi***: Simula que la silla original "explota" y se encoge hasta desaparecer, se incrementa **siExplosionFactor** para controlar el shader de explosión, al final se hace la interpolación **siScale** hacia 0.
5. ***AparecerSn***: Hace "crecer" la silla nueva desde escala 0→1 de forma rápida (**t = deltaTime * 2**), para que al llegar a escala 1, se configuran los keyframes para que **snPos** se desplace desde la salida hasta encima de la posición original (elevada), y se cambia a ***SnMoverAOriginal***.
6. ***SnMoverAOriginal***: Se desplaza **sn** desde la salida hasta la posición original elevada. Su transición se realiza cuando acaban los keyframes, pasa a ***SnBajarFinal***.
7. ***SnBajarFinal***: Baja la silla nueva 4 unidades en Y (desde la altura elevada hasta el suelo), se realiza con la interpolación de la componente Y con velocidad de 1.5. Al llegar al suelo, se cambia a la fase ***SnEscalarOriginal***.
8. ***SnEscalarOriginal***: Se ajusta la escala final de la silla nueva a 4.0f, al finalizar se marca la animación como **Completa** y no se procesan más fases.

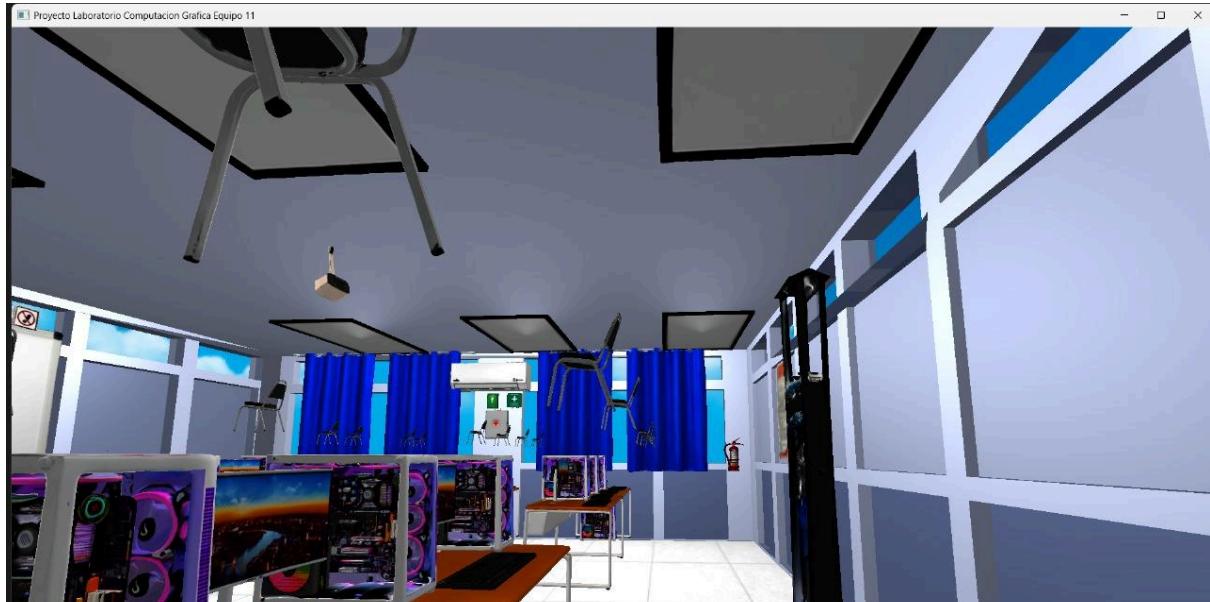


Figura 55. Animación de sillas actuales.



Figura 56. Animación de sillas remodeladas.

```
// Fase EscalandoAntesDeMover
silla.siScale = glm::mix(silla.siScale, 1.0f, deltaTime * 1.0f);
if (fabs(silla.siScale - 1.0f) < 0.05f) {
    silla.siScale = 1.0f;
    silla.fase    = SubirAntesDeMover;  }
// Fase SubirAntesDeMover
silla.siPos.y = glm::mix(silla.siPos.y, silla.siPosOriginal.y + 4.0f,
deltaTime * 1.0f);
if (fabs(silla.siPos.y - (silla.siPosOriginal.y + 4.0f)) < 0.05f) {
    silla.siPos.y = silla.siPosOriginal.y + 4.0f;
    silla.animacion.keyframes = { // Preparar keyframes para mover
La silla hacia la salida
```

```

        {0.0f, silla.siPos, 1.0f},
        {2.0f, silla.targetPos, 1.0f}  };
    silla.animacion.start();
    silla.fase = MoverASalida; }
// Fase MoverASalida
if (silla.animacion.active) {
    silla.animacion.update(deltaTime);
    silla.siPos = silla.animacion.interpolatedPosition; }
else {
    silla.siExplotaComoModel1 = true;
    silla.siExplosionFactor = 0.0f;
    silla.fase = DesaparecerSi; }
// Fase DesaparecerSi
if (silla.siExplosionFactor < 1.7f) {
    silla.siExplosionFactor += deltaTime * 0.3f; }
silla.siScale = glm::mix(silla.siScale, 0.0f, deltaTime * 1.0f);
if (silla.siExplosionFactor >= 1.7f && silla.siScale <= 0.05f) {
    // Ocultar si y preparar sn
    silla.siVisible = false;
    silla.siExplosionFactor = 0.0f;
    silla.siScale = 0.0f;
    silla.snVisible = true;
    silla.snScale = 0.0f;
    silla.snPos = silla.targetPos;
    // Definir keyframes para subida de la nueva silla
    silla.animacion.keyframes = {
        {0.0f, silla.snPos, 1.0f},
        {3.0f, silla.siPosOriginal + glm::vec3(0,4,0), 1.0f}  };
    silla.animacion.start();
    silla.fase = AparecerSn; }
// Fase AparecerSn
silla.snScale = glm::mix(silla.snScale, 1.0f, deltaTime * 2.0f);
if (fabs(silla.snScale - 1.0f) < 0.05f) {
    silla.snScale = 1.0f;
    // Configurar keyframes para mover sn a su posición original elevada
    silla.animacion.keyframes = {
        {0.0f, silla.snPos, 1.0f},
        {3.0f, silla.siPosOriginal + glm::vec3(0,4,0), 1.0f}  };
    silla.animacion.start();
    silla.fase = SnMoverAOriginal; }
// Fase SnMoverAOriginal
if (silla.animacion.active) {
    silla.animacion.update(deltaTime);
    silla.snPos = silla.animacion.interpolatedPosition; }
else {
    silla.fase = SnBajarFinal; }

```

```

// Fase SnBajarFinal
silla.snPos.y = glm::mix(silla.snPos.y, silla.siPosOriginal.y, deltaTime
* 1.5f);
if (fabs(silla.snPos.y - silla.siPosOriginal.y) < 0.01f) {
    silla.snPos.y = silla.siPosOriginal.y;
    silla.fase    = SnEscalarOriginal; }

// Fase SnEscalarOriginal
silla.snScale = glm::mix(silla.snScale, 4.0f, deltaTime * 1.0f);
if (fabs(silla.snScale - 4.0f) < 0.05f) {
    silla.snScale = 4.0f;
    silla.fase    = Completa; }

```

Keyframe para animación profesor.

Se utiliza el modelo del profesor para poder hacer el recorrido por el laboratorio virtual de computación gráfica.



Figura 57. Modelo de profesor.

Se utilizan las variables globales para el desplazamiento global a lo largo del eje Z y las oscilaciones articulares de extremidades y torso. Considerando lo siguiente:

1. **posicionZMuneco** mantiene la coordenada Z actual del muñeco, iniciando en -10.
2. Cada frame, si **caminar == true**, se incrementa o decrementa **posicionZMuneco** según **velocidadCaminar**, moviendo al profesor desde **posicionBaseInicio** hacia **posicionBaseFin** (o viceversa si se alcanza uno de los extremos).
3. Una vez que el personaje llega a uno de los límites (**posicionBaseInicio.z o posicionBaseFin.z**), la dirección de avance invierte, haciendo que recorra de ida y vuelta el pasillo virtual.

4. Variables de ángulo para controlar la oscilación de extremidades. El booleano **piernaAdelante** indica cuál pierna va hacia adelante en este instante. Cuando se alcanza el ángulo máximo definido por **factorPanto** o **factorBrazo**, **piernaAdelante** se invierte, haciendo que el miembro regrese a su posición inicial y la pierna opuesta comience su avance.
5. Factores de amplitud: **factorPanto** determina el ángulo máximo de oscilación de las piernas. **factorBrazo** regula la amplitud del balanceo de los **brazos.factorTorso** controla la ligera torsión del torso (**anguloTorso**) para equilibrar el movimiento. **anguloTorso** oscila suavemente a razón de **factorTorso**, generando una leve torsión que acompaña la marcha. **anguloCabeza** puede mantenerse estático o realizar pequeños giros inversos al torso, mejorando la sensación de equilibrio.
6. Para el dibujo cada parte (piernas, brazos, torso, cabeza) se invoca con su propia matriz de modelo, de modo que todas las oscilaciones y desplazamientos compongan la pose completa del profesor en movimiento.

```

bool caminar = false; // Para activar/desactivar la animación
float anguloPiernaDerecha = 0.0f;
float anguloPiernaIzquierda = 0.0f;
float anguloBrazoDerecho = 0.0f;
float anguloBrazoIzquierdo = 0.0f;
bool piernaAdelante = true; // Indica dirección del movimiento
(adelante o atrás)
float anguloRodillaDer = 0.0f;
float anguloRodillaIzq = 0.0f;
float anguloTorso = 0.0f;
float anguloManoDer = 0.0f;
float anguloManoIzq = 0.0f;
float anguloRodillaDerecha = 0.0f;
float anguloRodillaIzquierda = 0.0f;
float factorBrazo = 0.5f;
float factorPanto = 0.3f;
float factorTorso = 1.5f;
float anguloPiernaInferiorDerecha = 0.0f;
float anguloPiernaInferiorIzquierda = 0.0f;
float anguloCabeza = 0.0f;
float humv = false;
glm::vec3 posicionBaseInicio = glm::vec3(10.4f, -3.9f, -20.0f);
glm::vec3 posicionBaseFin = glm::vec3(10.4f, -3.9f, -40.0f);
float posicionZMuneco = -10.0f; // Comenzar en -10
float velocidadCaminar = 3.0f; // Ajusta la velocidad de caminata

```

Se define una máquina de estados para controlar el recorrido y las acciones del modelo del profesor, alternando fases de desplazamiento, giros y gestos con el brazo.

- **CAMINAR_Z**: el profesor avanza en línea recta sobre el eje Z (por ejemplo, de z = -10 hacia z = -40).
- **ROTAR_IZQUIERDA**: al llegar al extremo en Z, entra en este estado para girar 90° hacia la izquierda, de forma que pase de mirar hacia -Z a mirar hacia -X.
- **CAMINAR_X**: avanza ahora a lo largo del eje X (e.g. de x = 10.4 hacia otro valor).
- **ROTAR_180**: alcanza el segundo extremo y activa un giro de 180° completo, para regresar caminando en sentido contrario.
- **POST_ROTACION**: fase de estabilización tras el giro de 180°; puede incluir pequeñas correcciones de posición u orientación.
- **FIN**: marca la conclusión del trayecto y detiene cualquier movimiento o giro adicional.
- **EstadoCaminar estado = CAMINAR_Z**: mantiene en todo momento en qué fase de este recorrido se encuentra el personaje.

```
enum EstadoCaminar {
    CAMINAR_Z,           // 0: Avanza a lo Largo del eje Z
    ROTAR_IZQUIERDA,     // 1: Realiza un giro de 90° hacia la izquierda
    CAMINAR_X,           // 2: Avanza a lo Largo del eje X
    ROTAR_180,           // 3: Realiza un giro de 180° para invertir la
    dirección
    POST_ROTACION,       // 4: Fase opcional tras el giro completo
    FIN                 // 5: Estado final, detiene la animación
};

EstadoCaminar estado = CAMINAR_Z;
```

Para controlar los giros del muñeco se utiliza **anguloRotacionMuneco** que es el ángulo actual de orientación del modelo, en grados. Se inicializa a 180° porque el modelo ya está orientado mirando hacia el eje -Z. Y **velocidadRotacion** define cuántos grados gira el profesor en cada frame durante las fases de giro cortas (como ROTAR_IZQUIERDA).

```
float anguloRotacionMuneco = 180.0f; // para mirar hacia -Z
float velocidadRotacion = 2.0f;      // Grados por frame
float anguloRotacion180 = 0.0f;      // Ángulo acumulado para el giro 180°
float velocidadRotacion180 = 1.0f;    // Velocidad de giro por frame
```

La animación por keyframes se realiza desde la función **Animation()**. Impulsa la animación de “caminar” del modelo del profesor combinando una pequeña **máquina de estados** (para manejar pausas, giros y fases pos-giro) con un **ciclo de marcha** continuo (movimiento de piernas, brazos, rodillas, torso y cabeza) cuando está en el estado de caminar normal.

```

if (caminar) {
    float anguloMaxMuslo = 1.5f;
    float velocidadAnguloMuslo = 0.05f;
    float anguloMaxInferior = 2.5f;
    float velocidadAnguloInferior = 0.1f;
    float velocidadRotacion = 2.0f; // grados por frame
    float velocidadCaminarZ = 0.05f; // velocidad para en Z
    float velocidadCaminarX = 0.05f; // velocidad para en X
    if (estado == FIN) {
        // Sólo mueve La cabeza (balanceo lateral)
        anguloCabeza = sin(glfwGetTime() * 3.0f) * 90.0f;
        // Piernas, brazos, torso quietos
        anguloPiernaDerecha = 0.0f;
        anguloPiernaIzquierda = 0.0f;
        anguloPiernaInferiorDerecha = 0.0f;
        anguloPiernaInferiorIzquierda = 0.0f;
        anguloBrazoDerecho = 0.0f;
        anguloBrazoIzquierdo = 0.0f;
        anguloTorso = 0.0f;    }
    else if (estado == ROTAR_180) {
        // Solo avanzamos el giro 180°
        anguloRotacion180 += velocidadRotacion180;
        if (anguloRotacion180 >= 180.0f) {
            anguloRotacion180 = 180.0f;
            estado = POST_ROTACION; // Nuevo estado
        }
        // Mueve sólo la cabeza en ROTAR_180
        anguloCabeza = sin(glfwGetTime() * 3.0f) * 10.0f;
        // Piernas, brazos, torso quietos
        anguloPiernaDerecha = 0.0f;
        anguloPiernaIzquierda = 0.0f;
        anguloPiernaInferiorDerecha = 0.0f;
        anguloPiernaInferiorIzquierda = 0.0f;
        anguloBrazoDerecho = 0.0f;
        anguloBrazoIzquierdo = 0.0f;
        anguloTorso = 0.0f;    }
    else if (estado == POST_ROTACION) {
        // Sólo balanceo lateral de cabeza
        anguloCabeza = sin(glfwGetTime() * 3.0f) * 15.0f;
        // Piernas, brazos, torso quietos
    }
}

```

```

        anguloPiernaDerecha = 0.0f;
        anguloPiernaIzquierda = 0.0f;
        anguloPiernaInferiorDerecha = 0.0f;
        anguloPiernaInferiorIzquierda = 0.0f;
        anguloBrazoDerecho = 0.0f;
        anguloBrazoIzquierdo = 0.0f;
        anguloTorso = 0.0f;    }
    else {
        // Movimiento normal de caminar
        // Movimiento de cabeza balanceo Lateral
        anguloCabeza = sin(glfwGetTime() * 3.0f) * 10.0f;
        if (piernaAdelante) {
            anguloPiernaDerecha += velocidadAnguloMuslo;
            anguloPiernaIzquierda -= velocidadAnguloMuslo;
            anguloPiernaInferiorDerecha =
std::max(anguloPiernaInferiorDerecha - velocidadAnguloInferior,
-anguloMaxInferior);
            anguloPiernaInferiorIzquierda =
std::min(anguloPiernaInferiorIzquierda + velocidadAnguloInferior, 0.0f);
            anguloBrazoDerecho -= velocidadAnguloMuslo;
            anguloBrazoIzquierdo += velocidadAnguloMuslo;
            anguloRodillaDerecha =
std::min(anguloRodillaDerecha + 0.02f, anguloMaxInferior);
            anguloRodillaIzquierda =
std::max(anguloRodillaIzquierda - 0.02f, -anguloMaxInferior);
            anguloManoDer = anguloBrazoDerecho * 0.8f;
            anguloManoIzq = anguloBrazoIzquierdo * 0.8f;
            anguloTorso = anguloBrazoDerecho * factorTorso;
            if (anguloPiernaDerecha >= anguloMaxMuslo) {
                piernaAdelante = false;    }    }
        else {
            anguloPiernaDerecha -= velocidadAnguloMuslo;
            anguloPiernaIzquierda += velocidadAnguloMuslo;
            anguloPiernaInferiorDerecha =
std::min(anguloPiernaInferiorDerecha + velocidadAnguloInferior, 0.0f);
            anguloPiernaInferiorIzquierda =
std::max(anguloPiernaInferiorIzquierda - velocidadAnguloInferior,
-anguloMaxInferior);
            anguloBrazoDerecho += velocidadAnguloMuslo;
            anguloBrazoIzquierdo -= velocidadAnguloMuslo;
            anguloRodillaDerecha =
std::max(anguloRodillaDerecha - 0.02f, -anguloMaxInferior);
            anguloRodillaIzquierda =
std::min(anguloRodillaIzquierda + 0.02f, anguloMaxInferior);
            anguloManoDer = anguloBrazoDerecho * 0.8f;
            anguloManoIzq = anguloBrazoIzquierdo * 0.8f;

```

```
anguloTorso = anguloBrazoDerecho * factorTorso;
if (anguloPiernaDerecha <= -anguloMaxMuslo) {
    piernaAdelante = true; } } }
```

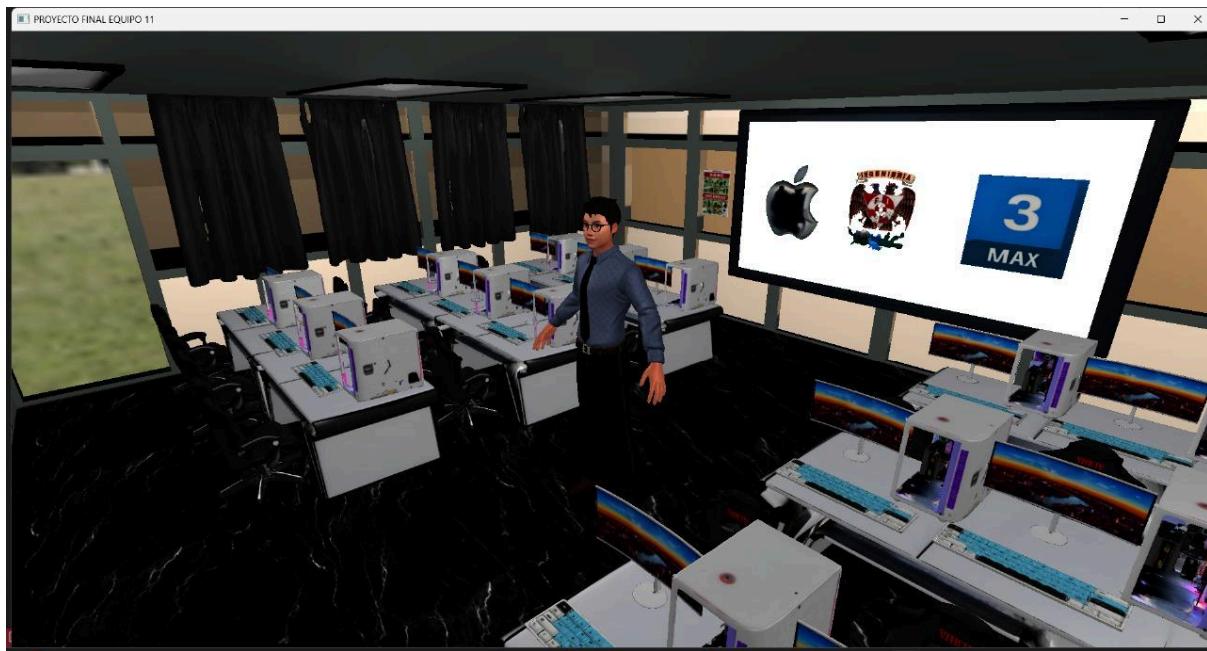


Figura 58. Modelo de profesor caminando por el laboratorio.

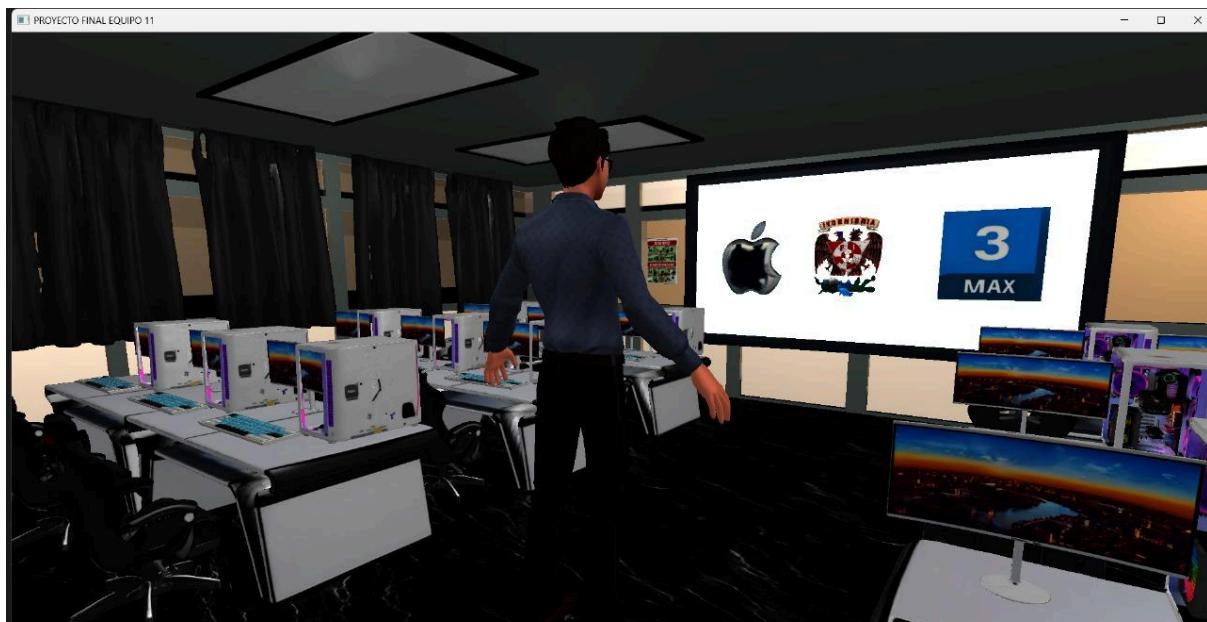


Figura 59. Modelo de profesor realizando animación.

Animación de logos en pantalla.

La animación de los logotipos que se dibujan en la “pantalla nueva” del laboratorio se articula en varias fases y utiliza tanto interpolaciones temporales

como dinámicas físicas simples (rebotes y colisiones) para dar vida a cada logo. Considerando este cambio y el rebote de los logos, se trabaja sobre la textura oscura de la pantalla nueva, haciendo los cambios de fases sobre la misma a cambiar el fondo blanco, para comenzar con las interacciones con los logos.

Las fases que se manejan son:

1. Preparación y aparición inicial donde se utilizan las siguientes variables:
 - **baseScale**: tamaño máximo del logo cuando esté completamente desplegado.
 - **animationProgress** $\in [0,1]$: indica el grado de aparición del logo (0 = invisible, 1 = a escala completa).
 - **light6AnimationProgress** $\in [0,1]$: controla la progresión de la animación de encendido de la luz puntual que “ilumina” el logo.
 - Banderas **modelVisibleAndReady** y **lightReady** que marcan cuándo el logo y la luz han completado su fase de aparición.
2. Teclas de reinicio, Pulsar F reinicia solo la animación de aparición/desaparición de este modelo (reset de **animationProgress**, **decayTimer**).
3. Interpolación de aparición: Cada frame, **animationProgress** crece linealmente hasta 1. El logo también “rebota” ligeramente en Y con **bounceHeight**, para añadir dinamismo en su entrada.

```

if (animationProgress < 1.0f)
    animationProgress += animationSpeed;
if (animationProgress >= 1.0f)
    modelVisibleAndReady = true;

scale = glm::mix(vec3(0), baseScale, animationProgress);

bounceHeight = 0.2f * sin(3π * animationProgress);
  
```

4. Animación de la luz puntual: Paralelamente, **light6AnimationProgress** crece con la misma lógica que la aparición del modelo. Solo cuando **lightReady = true** (luz al 100%), el logo pasa a la siguiente fase de rebote lateral.
5. Rebote lateral y colisiones dentro de la pantalla: Se tiene en cuenta la posición y la velocidad mediante **position** que inicia en el centro de la pantalla (por ejemplo, (-12.2, 3.5, -40)). **velocity** define la dirección y rapidez del desplazamiento lateral.

6. Movimiento y detección de bordes: Cada frame, se añade **velocity** a **position**. Luego se calcula un caja de colisión (**collisionHalfSize = baseScale * 0.5f**) y se verifica si el logo se acerca a cualquiera de los límites (**minPoint, maxPoint**), invierte el componente de **velocity** correspondiente (**velocity.x = -velocity.x**), provocando un efecto rebote contra los bordes de la pantalla.

```
// Solo activamos la animación de rebote si tanto el modelo como
// la luz están completamente visibles
if (modelVisibleAndReady && lightReady) {
    // El modelo y la luz están listos para la animación de rebote
    position += velocity;
    // Tamaño base fijo para colisiones, no cambia con la escala
    glm::vec3 collisionHalfSize = baseScale * 0.5f;
    // Comprobación de colisión
    if (position.x - collisionHalfSize.x <=
        minPoint.x || position.x + collisionHalfSize.x >= maxPoint.x) {
        velocity.x = -velocity.x;
    }
    if (position.y - collisionHalfSize.y <=
        minPoint.y || position.y + collisionHalfSize.y >= maxPoint.y) {
        velocity.y = -velocity.y;
    }
    if (position.z - collisionHalfSize.z <=
        minPoint.z || position.z + collisionHalfSize.z >= maxPoint.z) {
        velocity.z = -velocity.z;
    }
    // Temporizador para la desaparición del modelo después de 15 s
    decayTimer += deltaTime; // Incrementamos
    if (decayTimer >= 16.5f && !decayTriggered)
    { // 15 segundos de animación
        decayTriggered = true; // Activar la
        desaparición después de 15 segundos
    }
}
```

7. Animación de “pulsación” y rotación continua: Para la pulsación de escala **scaleTimer** acumula tiempo, **scaleFactor** oscila entre 0.4 y 1.0, generando una ligera expansión y contracción continua del logo mientras rebota. Mientras que para la rotación, el logo gira lentamente alrededor de su eje Y para añadir un toque dinámico y tridimensional.

```
// Asegurarse de que la escala no sea 0 antes de dibujar el modelo
if (scale.x > 0.0f || scale.y > 0.0f || scale.z > 0.0f) {
    // Comenzar a dibujar si solo cuando la luz esté lista
    if (lightReady) {
```

```

        // Aplicar la transformación al modelo
        glm::mat4 modelq = glm::mat4(1.0f);
// dibujamos en la posición fija
        if (animationProgress < 1.0f) {
// Animación de aparición con escala de 0 a la escala definida
            float scaleFactor =
glm::mix(0.0f, 1.0f, animationProgress); // De 0 a 1 con animacion
            scale = baseScale *
scaleFactor; // Ajustar la escala del modelo
            // Rebote en Y durante la aparición
            float bounceHeight = 0.2f *
sin(3.0f * glm::pi<float>() * animationProgress); // Rebote en Y
            modelq = glm::translate(modelq,
glm::vec3(-12.2f, 3.5f + bounceHeight, -40.0f)); // Posición fija
        }
        else {
// Después de la aparición, permitir el rebote en X y Z
            modelq = glm::translate(modelq,
position); // Usamos la posición calculada para el rebote lateral
        }
        // Rotación mientras el modelo está en movimiento
        modelq = glm::rotate(modelq,
glm::radians(rotationAngle), glm::vec3(0.0f, 1.0f, 0.0f));
        modelq = glm::scale(modelq, scale);
// Aplicar la escala modificada de manera continua
        // Enviar los datos al shader
glUniform1f(glGetUniformLocation(lightingShader.Program,
"explosionFactor"), 0.0f);
glUniform1i(glGetUniformLocation(lightingShader.Program,
"transparency"), 0);
glUniformMatrix4fv(glGetUniformLocation(lightingShader.Program,
"model"), 1, GL_FALSE, glm::value_ptr(modelq));
        fi.Draw(lightingShader);
    }
}

```

- Desaparición tras tiempo límite: Después de ~15 s de rebote, **decayTriggered** se activa. Para la interpolación de desaparición en los siguientes 5 s, la escala se reduce suavemente a 0, haciendo que el logo se desvanezca.

```

decayTimer += deltaTime;
if (decayTimer >= 16.5f) decayTriggered = true;
float fadeFactor = clamp(1.0f - (decayTimer - 16.5f) / 5.0f, 0.0f,
1.0f);
scale = lastScale * fadeFactor;

```

- Paso a la siguiente animación de logos: Cuando la escala llega a cero y ***decayTriggered == true***, se activa ***othersCanAppear = true***, permitiendo que los otros dos logotipos (***ap*** y ***td***) inicien su secuencia de aparición y rebote, usando una función genérica ***applyReboundAndScale(...)***.
- Renderización final: Cada frame, una vez calculadas posición, rotación y escala, se construye la matriz de modelo, luego se envían los ***uniforms*** al shader de iluminación, finalmente para que el logo se dibuje con sus transformaciones aplicadas.

```

modelMat = translate(position) * rotate(rotationAngle) *
scale(scale); // Rotación mientras el modelo se mueve
modelq = glm::rotate(modelq,
glm::radians(rotationAngle), glm::vec3(0.0f, 1.0f, 0.0f));
modelq = glm::scale(modelq,
scale); // Aplicar la escala modificada de manera continua
glUniform1f(glGetUniformLocation(lightingShader.Program,
"explosionFactor"), 0.0f);
glUniform1i(glGetUniformLocation(lightingShader.Program,
"transparency"), 0);
glUniformMatrix4fv(glGetUniformLocation(lightingShader.Program,
"model"), 1, GL_FALSE, glm::value_ptr(modelq));
fi.Draw(lightingShader);

```

Este sistema modular de **fases**, **timers** y **mecanismos de interpolación** garantiza que los logotipos entren con estilo, reboten de forma realista dentro de la pantalla, pulsen y finalmente se desvanezcan, ofreciendo una experiencia visual atractiva y dinámica para los patrocinadores y usuarios del laboratorio virtual.

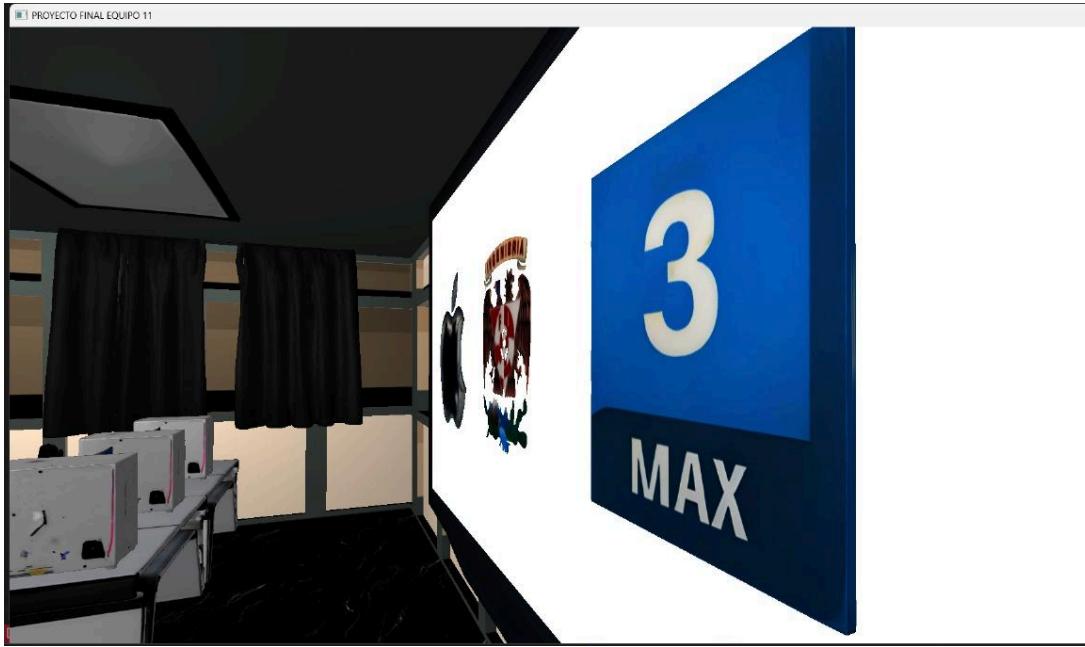


Figura 60. Logos cargados en la textura de la pantalla.

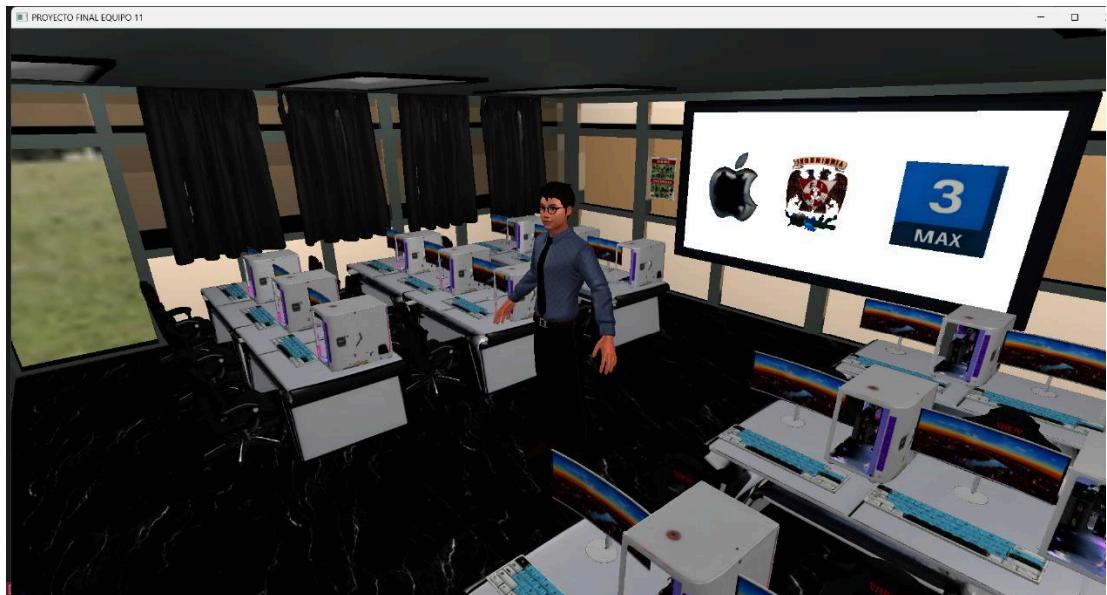


Figura 61. Modelo nuevo de salón con profesor y logos.

Animaciones adicionales.

Las animaciones adicionales se concentran en el cambio del resto del mobiliario, por lo que estas animaciones globales son de aparición y desaparición.

Cada sección sigue el mismo patrón: un flag que arranca la transición, un factor de escala que se interpola con **deltaTime**, y al llegar a 0 o 1 se reconfiguran visibilidad y se disparan nuevas fases.

1. Pantallas “pi” → “n”: Minimizar “pi” con **piMinimizando** activa la interpolación de **piScaleFactor** de 1→0 a velocidad 2×. Al llegar a 0, oculta

pi y prepara n (**visible=false→true**) para aparecer con "n" **nApareciendo** incrementa **nScaleFactor** de 0→1 al mismo ritmo; al completarse, se detiene la fase de aparición.

```

if (piMinimizando) {
    piScaleFactor -= deltaTime * 2.0f;
    if (piScaleFactor <= 0.0f) {
        piScaleFactor = 0.0f;
        piMinimizando = false;
        piVisible = false;
        nVisible = true;
        nApareciendo = true;
        nScaleFactor = 0.0f; } }
if (nApareciendo) {
    nScaleFactor += deltaTime * 2.0f;
    if (nScaleFactor >= 1.0f) {
        nScaleFactor = 1.0f;
        nApareciendo = false; } }

```

2. **Mesas viejas → mesas nuevas:** Se hace la reducción la escala de las mesas antiguas baja a 0; al completarse, se ocultan y se habilita la aparición de las nuevas. En la aparición se escala de las mesas nuevas sube de 0→1, luego se desactiva la fase.

```

if (mesasViejasReduciendo) {
    mesasScaleFactorViejas -= deltaTime * 2.0f;
    if (mesasScaleFactorViejas <= 0.0f) {
        mesasScaleFactorViejas = 0.0f;
        mesasViejasReduciendo = false;
        mesasViejasVisibles = false;
        mesasNuevasVisibles = true;
        mesasAnimandoAparecer = true;
        mesasScaleFactorNuevas = 0.0f; } }
if (mesasAnimandoAparecer) {
    mesasScaleFactorNuevas += deltaTime * 2.0f;
    if (mesasScaleFactorNuevas >= 1.0f) {
        mesasScaleFactorNuevas = 1.0f;
        mesasAnimandoAparecer = false; } }

```

3. Teclados viejo (“te”) → nuevo (“tn”): se minimiza el viejo (**te**), se oculta tras llegar a escala 0, y luego aparece el nuevo (**tn**).

```

if (teMinimizando) {
    teScaleFactor -= deltaTime * 2.0f;
    if (teScaleFactor <= 0.0f) {
        teScaleFactor = 0.0f;
        teMinimizando = false;
        teVisible = false;
        tnVisible = true;
        tnMinimizando = true;
        tnScaleFactor = 0.0f; } }
if (tnMinimizando) {
    tnScaleFactor += deltaTime * 2.0f;
    if (tnScaleFactor >= 1.0f) {
        tnScaleFactor = 1.0f;
        tnMinimizando = false; } }

```

```

        teVisible      = false;
        tnVisible      = true;
        tnApareciendo = true;
        tnScaleFactor = 0.0f; } }

if (tnApareciendo) {
    tnScaleFactor += deltaTime * 2.0f;
    if (tnScaleFactor >= 1.0f) {
        tnScaleFactor = 1.0f;
        tnApareciendo = false; } }

```

4. Modelos PV y MESS (flag **minimizandoConT**): Baja la escala de ambos modelos a ritmo fijo (0.02 por frame), y al llegar a 0 oculta cada uno.

```

if (minimizandoConT) {
    escalaPv -= 0.02f;
    escalaMess-= 0.02f;
    if (escalaPv <= 0.0f) {
        escalaPv = 0.0f;
        modeloPvVisible = false;
    }
    if (escalaMess <= 0.0f) {
        escalaMess = 0.0f;
        modeloMessVisible = false;
    } }

```

5. Sillas especiales 1 y 2: Ambas sillas reducen su escala de forma idéntica y se ocultan al llegar a 0.

```

if (sillaEspecial1Minimizando) {
    escalaSillaEspecial1 -= deltaTime * 2.0f;
    if (escalaSillaEspecial1 <= 0.0f) {
        escalaSillaEspecial1 = 0.0f;
        sillaEspecial1Minimizando = false;
        sillaEspecial1Visible = false;
    } }
if (sillaEspecial2Minimizando) {
    escalaSillaEspecial2 -= deltaTime * 2.0f;
    if (escalaSillaEspecial2 <= 0.0f) {
        escalaSillaEspecial2 = 0.0f;
        sillaEspecial2Minimizando = false;
        sillaEspecial2Visible = false;
    } }

```

6. Modelo "MM": Desciende la escala de "MM" y oculta el modelo al completarse.

```

if (minimizandoMM) {

```

```
escalaMM -= deltaTime * 2.0f;  
if (escalaMM <= 0.0f) {  
    escalaMM = 0.0f;  
    minimizandoMM = false;  
    modeloMMVisible = false;  
} }
```



Figura 62. Laboratorio con modelos principales remodelados.

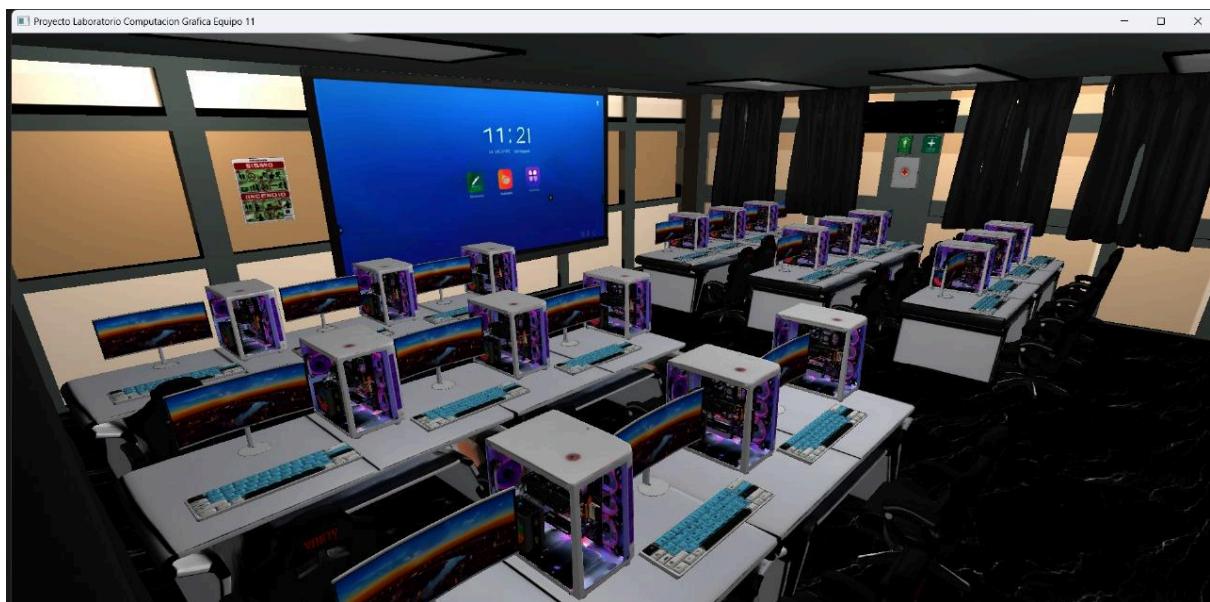


Figura 63. Salón totalmente remodelado.

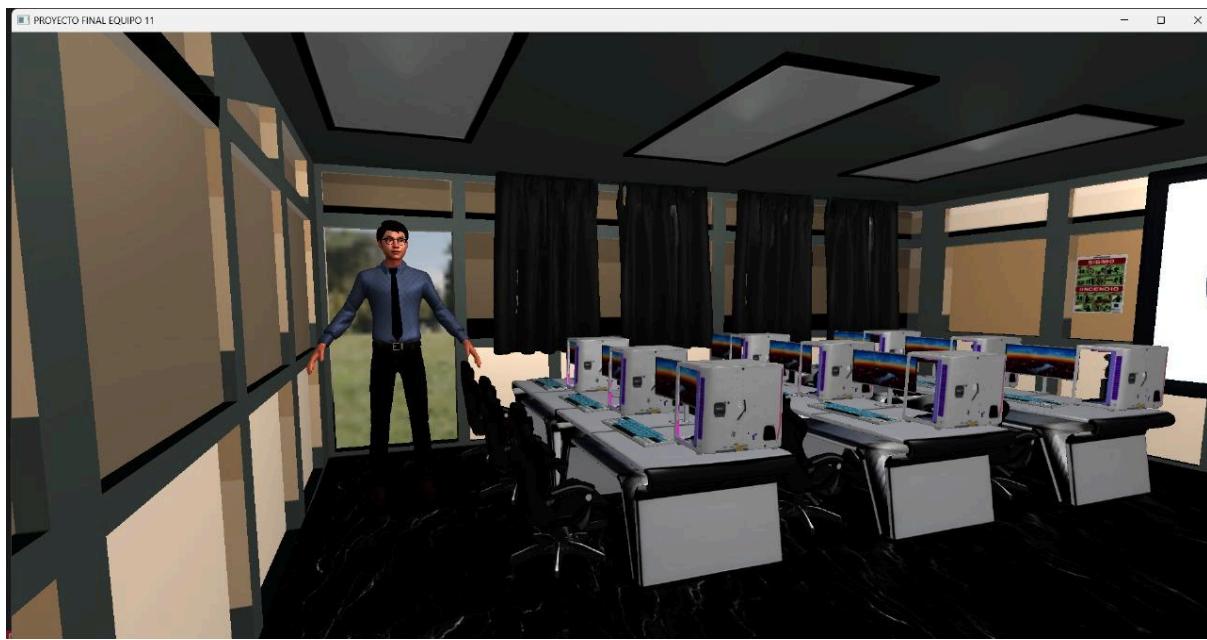


Figura 64. Nuevo Laboratorio de computación gráfica con profesor.

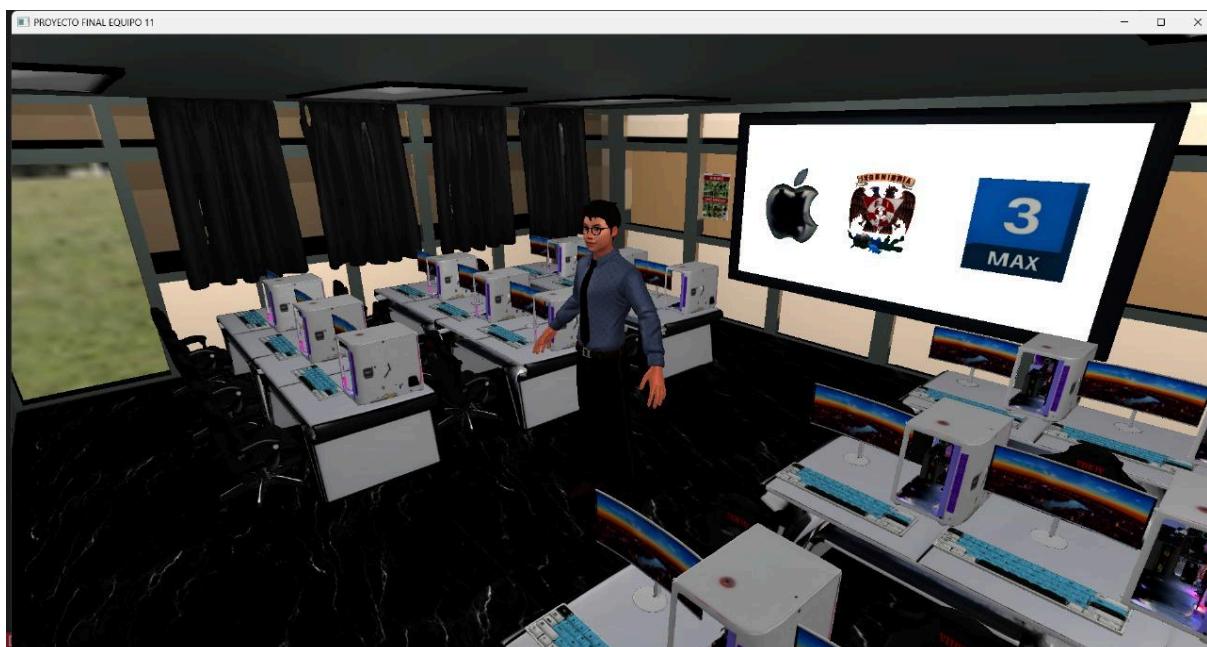


Figura 65. Laboratorio de computación gráfica innovado.

Manejo de entrada de teclado.

El **KeyCallback** es el punto de entrada para manejar todas las interacciones de teclado. Se invoca cada vez que el usuario pulsa o suelta una tecla, y su firma es: **key**: el código de la tecla (p. ej. **GLFW_KEY_N**).

- **Action**: si la tecla se ha pulsado (**GLFW_PRESS**), soltado (**GLFW_RELEASE**) o repetido.
- **window**: la ventana asociada, útil para cerrar la aplicación.

```
void KeyCallback(GLFWwindow* window,
                  int key,
                  int scancode,
                  int action,
                  int mode)
```

1. Salir con ESC: cuando presionas ESC se marca la ventana para cerrarse, terminando el bucle principal y saliendo limpiamente de la aplicación.

```
if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS) {
    glfwSetWindowShouldClose(window, GL_TRUE); }
```

2. Actualización del array de teclas (keys[1024]): Sirve para mantener un array booleando **keys[]** que tu función **DoMovement()** lee cada frame, permitiendo movimiento continuo mientras una tecla está presionada (**W,A,S,D**, flechas, etc.).

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

3. Disparar explosión de pantallas (tecla N): Activa la bandera **explosionActive**, que arranca la fase de encogimiento (“explosión”) de tu primer modelo de pantalla.

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

4. Reinicio de explosión/implosión (tecla M): Vuelve a estado inicial restableciendo factores y visibilidades para poder volver a “reproducir” la explosión y aparición de la segunda pantalla.

```
if (key == GLFW_KEY_M && action == GLFW_PRESS) {
    explosionFactor      = 0.0f;
    implosionFactor     = 0.0f;
    explosionActive     = false;
    implosionActive     = false;
    model1Visible       = true; }
```

```
model2Visible      = false; }
```

5. Iniciar animación de “ga” (tecla B): Resetea y activa la primera explosión de tu primer componente de PC (modelo “ga”), preparándolo para contraerse de nuevo.

```
if (key == GLFW_KEY_B && action == GLFW_PRESS) {  
    gaExplosionActive = true;  
    gaInflating      = true;  
    gaContracting    = false;  
    gaExplosionFactor = 1.0f;  
    gaVisible        = true; }
```

6. Comenzar “armado” de todas las PCs (tecla X): Recorre todas las instancias de **InstanciaAnimacion** y dispara la fase de contracción de “ga” (**explosión inicial**) en cada PC.

```
if (key == GLFW_KEY_X && action == GLFW_PRESS) {  
    for (auto& instancia : animaciones) {  
        instancia.gaContracting = true;  
        instancia.gaExplosionActive = true; } }
```

7. Animación de sillas (tecla P): Para cada silla que no esté ya en animación, configura un par de keyframes (subida y aparición), arranca su **SillaKeyframeAnimation** y marca **siMoving** para que inicie la fase de interpolación.

```
if (key == GLFW_KEY_P && action == GLFW_PRESS) {  
    for (auto& silla : sillas) {  
        if (!silla.animacion.active  
            && silla.siVisible  
            && silla.fase ==  
SillaAnimada::FaseAnimSilla::Completa)  
            silla.fase =  
SillaAnimada::FaseAnimSilla::EscalandoAntesDeMover;  
            silla.animacion.keyframes = {  
                {0.0f, silla.snPos, 1.0f},  
                {1.0f, {silla.targetPos.x, 4.0f, silla.targetPos.z},  
1.0f};  
                silla.animacion.start();  
                silla.siMoving = true; } }
```

8. Reiniciar animación de sillas (tecla L): Devuelve cada silla a su estado original, limpiando flags y keyframes para permitir nuevas ejecuciones.

```

if (key == GLFW_KEY_L && action == GLFW_PRESS) {
    for (auto& silla : sillas) {
        // Restablece posiciones, escalas y flags de ambas sillas si/sn
        silla.siPos          = silla.siPosOriginal;
        silla.siScale        = 4.0f;
        silla.siVisible      = true;
        silla.snPos          = silla.targetPos + glm::vec3(0,4,0);
        silla.snScale        = 0.0f;
        silla.snVisible      = false;
        silla.siMoving       = false;
        silla.snAppearing    = false;
        silla.snReturning    = false;
        silla.snRegresoTerminado = false;
        silla.animacion.keyframes.clear();
        silla.animacion.active = false;
        silla.fase =
SillaAnimada::FaseAnimSilla::EscalandoAntesDeMover;
    }
}

```

9. Reiniciar por completo todas las PCs (tecla C): Lleva todas las **InstanciaAnimacion** a su estado inicial, como justo antes de pulsar 'X'.

```

if (key == GLFW_KEY_C && action == GLFW_PRESS) {
    for (auto& instancia : animaciones) {
        // Restablece **todas** Las variables de animación de ga
    }
    gaAntesDeX = false;
    animacionXCompleta = false; }

```

10. Transición general de mobiliario (tecla T): Activa simultáneamente múltiples animaciones de minimizado/aparición de mobiliario (mesas, teclados, sillas especiales, cortinas, rack, etc.), preparando la transición de toda la sala.

```

if (key == GLFW_KEY_T && action == GLFW_PRESS) {
    piMinimizando      = true;
    mesasViejasReduciendo= true;
    teMinimizando      = true;
    minimizandoConT    = true;
    sillaEspecial1Minimizando = true;
    sillaEspecial2Minimizando = true;
    minimizandoMM      = true;
    cortinasViejasVisibles = false;
    cortinasNuevasVisibles = true;
}

```

```

rackVisible          = false;
modemVisible        = true;
aireViejoVisible    = false;
aireNuevoVisible    = true;
proyectorVisible    = false; }

```

11. Cambiar modelo de salón (tecla Y): Alterna la visibilidad entre el salón antiguo y el nuevo.

```

if (key == GLFW_KEY_Y && action == GLFW_PRESS) {
    modeloSalonVisible    = false;
    modeloSalonNuevoVisible= true; }

```

12. Encender/apagar luces puntuales (teclas U, I, O): Invierten los flags **lucesXneg10**, **lucesX0** y **lucesX10**, controlando qué columnas de luces puntuales se iluminan.

```

if (key == GLFW_KEY_U && action == GLFW_PRESS) lucesXneg10 =
!lucesXneg10;
if (key == GLFW_KEY_I && action == GLFW_PRESS) lucesX0      =
!lucesX0;
if (key == GLFW_KEY_O && action == GLFW_PRESS) lucesX10      =
!lucesX10;

```



Figura 66. Teclas de navegación.

Espacios estratégicos para patrocinadores.

Se plantea la propuesta sobre los espacios publicitarios que tendremos en el laboratorio de computación gráfica; el objetivo es maximizar la visibilidad de las marcas asociadas, integrarlas de forma natural en el entorno virtual y generar puntos de contacto de alto impacto con estudiantes, docentes y visitantes. A continuación se describen las ubicaciones y formatos recomendados:

Ubicación	Formato publicitario	Beneficios para el patrocinador
Pantalla de bienvenida	Logotipo estático o animado al iniciar la simulación	100 % de visibilidad al abrir la aplicación; refuerzo de marca en cada inicio; posibilidad de mostrar breve slogan o enlace interactivo a la web del patrocinador.
Skybox exterior (murales)	Banners panorámicos 360° en la fachada virtual del laboratorio	Exposición ambiental constante durante la navegación; asociación de la marca con el entorno universitario; alto nivel de inmersión y recuerdo de marca.
Monitores de cada estación	"Skins" o marcos con branding en los bordes de los monitores	Visibilidad en primer plano durante la interacción; se puede alternar entre distintas pantallas patrocinadas (rotativas) para múltiples anunciantes.
Vidrios exteriores	Vinilos semitransparentes con logotipos y mensajes clave	Integración sutil en la arquitectura virtual; buena legibilidad gracias a la iluminación exterior; posicionamiento de marca a la altura de la vista mientras se recorre el perímetro del salón.
Mesas centrales	Pegatinas vinílicas o grabados en la superficie ("tabletop branding")	Contacto directo cuando los usuarios se acercan a modelar o tomar notas; ideal para indicar zonas patrocinadas (por ejemplo, "Área de Innovación patrocinada por...").

Mobiliario secundario	Respaldos de sillas o costados de escritorios con vinilos corporativos	Exposición discreta pero constante durante conferencias y prácticas; ideal para reforzar la presencia de marca en un entorno académico.
-----------------------	--	---

Tabla 9. Propuestas de espacios para patrocinadores.

Gestión de recursos y limpieza.

Una correcta liberación de recursos es esencial para evitar fugas de memoria, garantizar que el programa cierre sin errores y preparar el sistema para futuras ejecuciones o cargas de módulos. En el proyecto para el mejoramiento del laboratorio de computación gráfica, esta gestión se lleva a cabo en dos niveles: recursos de GPU (buffers, texturas, shaders) y recursos del contexto gráfico y sistema (ventana, callbacks y memoria de GLFW).

Liberación de objetos de OpenGL.

Tras salir del bucle principal de renderizado, el programa ejecuta un bloque de código dedicado a destruir o eliminar todos los objetos creados en la GPU:

- Vertex Array Objects (VAO) y Vertex Buffer Objects (VBO).
- Esto libera la memoria en la GPU asociada a los datos de vértices y evita que queden buffer objects “huérfanos” tras el cierre de la aplicación.

```
glDeleteVertexArrays(1, &VAO_id);
glDeleteBuffers(1, &VBO_id);
```

Para las texturas y cubemaps:

- Cada textura cargada mediante SOIL2 o stb_image y subida con **glTexImage2D** se asocia a un identificador (**textureID**). Al finalizar, se llama a **glDeleteTextures(1, &textureID)**;
- El skybox utiliza un **cubemap**, que internamente consiste en seis texturas; igualmente se liberan en bloque usando **glDeleteTextures(1, &cubemapTextureID)**;

Los shaders y programas de shader se liberan de la siguiente forma:

- Los objetos Shader encapsulan la creación de programas (**Program**) y shaders individuales (vértice y fragmento). Para cada Shader se ejecuta: **glDeleteProgram(shader.Program)**;

- Esto descarta tanto los shaders compilados como el programa enlazado, liberando espacio de instrucción en la GPU.

Para los modelos (Assimp) y buffers asociados:

- La clase **Model** gestiona múltiples mallas y texturas. En su destructor o método **Cleanup()**, realiza la eliminación de VAO/VBO/EBO propios de cada sub-mesh. Además, la liberación de texturas específicas de material.

Asimismo, **Assimp** puede liberar internamente estructuras de datos con: **importer.FreeScene()**; sólo si se emplea un **Importer** local.

Desconexión de callbacks y destrucción de la ventana.

Eliminar callbacks antes de cerrar, es buena práctica “desregistrar” los callbacks para evitar accesos a punteros nulos o lecturas fuera de contexto. Con esto elimina las referencias a nuestras funciones de entrada en la estructura interna de GLFW.

```
glfwSetKeyCallback(window, nullptr);
glfwSetCursorPosCallback(window, nullptr);
```

Se debe destruir la ventana y el contexto para liberar la memoria del contexto OpenGL, de la ventana y de cualquier recurso interno de GLFW asociado a ella. Se hace con **glfwDestroyWindow(window)**.

Finalizar **GLFW**, para cerrar por completo la biblioteca y desasociar hilos o recursos del sistema operativo: **glfwTerminate()**;

Consideraciones adicionales.

- Orden de destrucción: siempre se liberan primero los objetos dependientes (buffers, texturas, shaders) y al final el contexto (ventana). Esto previene accesos inválidos durante la limpieza.
- Robustez del destructor: encapsular la limpieza en los destructores de las clases (**Shader**, **Model**) garantiza que, aunque la aplicación termine por excepción, el stack unwinding ejecutará sus destructores y liberará los recursos.
- Verificación de errores: tras cada llamada de borrado puede consultarse **glGetError()** para confirmar que no hay valores residuales en el estado de error de OpenGL. Esto ayuda a detectar liberaciones dobles o IDs inválidos.

Con esta estrategia de gestión y limpieza, el proyecto asegura que todos los recursos de GPU y sistema se liberen adecuadamente, manteniendo la estabilidad del sistema operativo, favoreciendo la depuración y evitando la acumulación de memoria no liberada en ejecuciones reiteradas.

Herramientas de comunicación y trabajo.

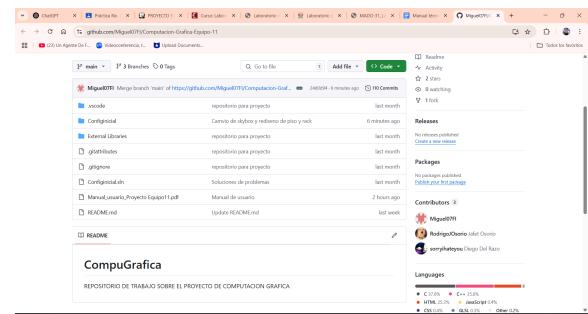
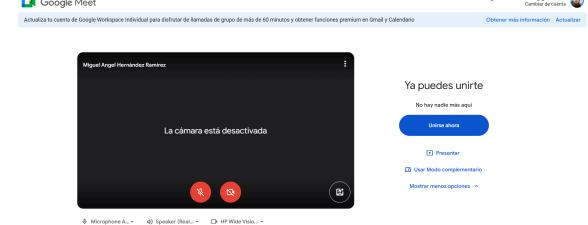
Herramienta	Evidencias
<p>Trello: Mediante uso de un pizarrón de actividades donde se registraron los avances logrados por fechas.</p> <p>Enlace: https://trello.com/b/66JE9wQI/proyecto-equipo11</p>	
<p>GitHub: Creación de repositorio con diferentes ramas se trabajaron los diferentes ámbitos del proyecto por cada colaborador.</p> <p>Enlace: https://github.com/Miguel07FI/Computacion-Grafica-Equipo-11</p>	
<p>Meet: Se realizaron reuniones entre los integrantes del equipo para platicar decisiones importantes y avances.</p>	



Tabla 9. Medios de comunicación utilizados.

Fin de manual técnico.

Este manual está dedicado a desarrolladores que deseen mantener, extender o refactorizar el recorrido virtual del laboratorio de computación gráfica. A través de este documento se establecen las bases arquitectónicas del proyecto, se detalla el funcionamiento de sus componentes clave y se proponen lineamientos para facilitar la incorporación de nuevas funcionalidades gráficas, físicas o interactivas. Asimismo, se fomenta el uso de prácticas robustas de programación y documentación para asegurar la escalabilidad y mantenibilidad del sistema a largo plazo.

Soporte y Contacto.

Para dudas o reportes de errores, contacta a:

Nombre.	Correo electrónico.	Teléfono.
Del Razo Sánchez Diego Adrián.	dadrs03@hotmail.com	5537299820
Hernández Ramírez Miguel Ángel.	miguelhernandez0532@gmail.com	5551436962
Osorio Ángeles Rodrigo Jafet.	rodri.osorio19@gmail.com	5618316866

Tabla 10. Soporte y contacto.

ENLACE DE REPOSITORIO PARA CONSULTA DETALLADA

<https://github.com/MiguelO7FI/Computacion-Grafica-Equipo-11>

REFERENCIAS SIGNIFICATIVAS

Haven, P. (s. f.). HDRIs: Skies • Poly Haven. Poly Haven.
<https://polyhaven.com/hdris/skies>

Tecnologías interactivas y computación gráfica. (s. f.). YouTube.
<https://www.youtube.com/@ArturoVMS>

Tripo AI - Create Your First 3D Model with Text and Image in Seconds. (s. f.).
<https://www.trip03d.ai/>