UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

DIVISIÓN DE INGENIERÍA ELÉCTRICA

INGENIERÍA EN COMPUTACIÓN

LABORATORIO DE COMPUTACIÓN GRÁFICA e

INTERACCIÓN HUMANO COMPUTADORA

DOCUMENTO:

MANUAL TÉCNICO

NOMBRES COMPLETOS / Nºs de Cuenta:

Gómez Enríquez Agustín…….317031405

Valenzuela Asencio Gustavo……117002029

Solé Pi Arnau Roger…….320338274

GRUPO DE LABORATORIO: 3

GRUPO DE TEORÍA: 5 / 6 / 6

SEMESTRE 2026-1

FECHA DE ENTREGA LÍMITE: 8 de noviembre del 2025

CALIFICACIÓN: _____

NOTA: No se muestra todo el código ya que es bastante código pero se muestran las partes importantes

Para mejorar la modularidad y evitar la centralización de toda la lógica en el archivo principal (main), el código se ha estructurado en los siguientes archivos:

1. Gestión de Activos y Recursos (Managers)

Este conjunto de clases sigue el patrón de diseño Manager para centralizar la carga y el acceso a los recursos.

- AssetConstants: Archivo que centraliza todas las constantes de la aplicación, principalmente nombres y rutas (paths) de modelos, texturas y skyboxes. Su objetivo es evitar el uso de cadenas de texto directamente en el código.

```cpp
# pragma once
#include <string>
#include <vector>

// Namespace para guardar nombres y rutas de assets
namespace AssetConstants {

    // Nombres de texturas
    namespace TextureNames{
        const std::string AGAVE = "agave";
        const std::string PASTO = "pasto";
        const std::string LADRILLO = "brick";
        const std::string TIERRA = "dirt";
        const std::string EMPEDRADO = "empedrado";
        const std::string AGUA = "agua";
        const std::string MAYAN_BRICKS = "mayan_bricks";
        // Texturas de Cuphead
        const std::string CUPHEAD_TEXTURE = "cuphead_texture";
        const std::string POPOTE_ROJO = "popote_rojo";
        const std::string CAUCHO = "caucho";
    }


    // Nombres de modelos
    namespace ModelNames{
        const std::string CABEZA_OLMECA = "cabeza_olmeca";
        const std::string PIRAMIDE = "piramide";
        const std::string MAIZ = "maiz";
        const std::string ARBOL_A = "arbol_a";
        const std::string ARBOL_B = "arbol_b";
        const std::string ARBOL_C = "arbol_c";
        const std::string CANOA = "canoa";
        const std::string MAYA_CANOA = "maya_canoa";
        const std::string PRIMO = "primo";
```

- ModelManager: Gestiona el ciclo de vida de los modelos 3D. Carga los modelos desde sus archivos y los almacena en una estructura de datos asociativa (un mapa), permitiendo recuperarlos eficientemente usando su nombre como clave.

```cpp
#include "ModelManager.h"

// Carga todos los modelos al inicializar el ModelManager
ModelManager::ModelManager()
{

    loadModel(AssetConstants::ModelNames::CABEZA_OLMECA, AssetConstants::ModelPaths::CABEZA_OLMECA);
    loadModel(AssetConstants::ModelNames::PIRAMIDE, AssetConstants::ModelPaths::PIRAMIDE);
    loadModel(AssetConstants::ModelNames::MAIZ, AssetConstants::ModelPaths::MAIZ);
    loadModel(AssetConstants::ModelNames::ARBOL_A, AssetConstants::ModelPaths::ARBOL_A);
    loadModel(AssetConstants::ModelNames::ARBOL_B, AssetConstants::ModelPaths::ARBOL_B);
    loadModel(AssetConstants::ModelNames::ARBOL_C, AssetConstants::ModelPaths::ARBOL_C);
    loadModel(AssetConstants::ModelNames::CANOA, AssetConstants::ModelPaths::CANOA);
    loadModel(AssetConstants::ModelNames::MAYA_CANOA, AssetConstants::ModelPaths::MAYA_CANOA);
    loadModel(AssetConstants::ModelNames::PRIMO, AssetConstants::ModelPaths::PRIMO);
    loadModel(AssetConstants::ModelNames::STREET_LAMP, AssetConstants::ModelPaths::STREET_LAMP);
    loadModel(AssetConstants::ModelNames::LAMP_RING, AssetConstants::ModelPaths::LAMP_RING);

    loadModel(AssetConstants::ModelNames::CABEZA_HOLLOW, AssetConstants::ModelPaths::CABEZA_HOLLOW);
    loadModel(AssetConstants::ModelNames::CUERPO1_HOLLOW, AssetConstants::ModelPaths::CUERPO1_HOLLOW);
    loadModel(AssetConstants::ModelNames::CUERPO2_HOLLOW, AssetConstants::ModelPaths::CUERPO2_HOLLOW);
    loadModel(AssetConstants::ModelNames::CUERPO3_HOLLOW, AssetConstants::ModelPaths::CUERPO3_HOLLOW);


    loadModel("cuphead", AssetConstants::ModelPaths::CUPHEAD);


    // Cargar modelos de Cuphead (jerárquico)
    loadModel(AssetConstants::ModelNames::CUPHEAD_TORSO, AssetConstants::ModelPaths::CUPHEAD_TORSO);
    loadModel(AssetConstants::ModelNames::CUPHEAD_CABEZA, AssetConstants::ModelPaths::CUPHEAD_CABEZA);
    loadModel(AssetConstants::ModelNames::CUPHEAD_LECHE, AssetConstants::ModelPaths::CUPHEAD_LECHE);
    loadModel(AssetConstants::ModelNames::CUPHEAD_POPOTE, AssetConstants::ModelPaths::CUPHEAD_POPOTE);
    loadModel(AssetConstants::ModelNames::CUPHEAD_BRAZO_IZQUIERDO, AssetConstants::ModelPaths::CUPHEAD_BRAZO_
    loadModel(AssetConstants::ModelNames::CUPHEAD_BRAZO_DERECHO, AssetConstants::ModelPaths::CUPHEAD_BRAZO_DE
```

```cpp
    // Cargar modelos de Cuphead (jerárquico)
    loadModel(AssetConstants::ModelNames::CUPHEAD_TORSO, AssetConstants::ModelPaths::CUPHEAD_TORSO);
    loadModel(AssetConstants::ModelNames::CUPHEAD_CABEZA, AssetConstants::ModelPaths::CUPHEAD_CABEZA);
    loadModel(AssetConstants::ModelNames::CUPHEAD_LECHE, AssetConstants::ModelPaths::CUPHEAD_LECHE);
    loadModel(AssetConstants::ModelNames::CUPHEAD_POPOTE, AssetConstants::ModelPaths::CUPHEAD_POPOTE);
    loadModel(AssetConstants::ModelNames::CUPHEAD_BRAZO_IZQUIERDO, AssetConstants::ModelPaths::CUPHEAD_BRAZO_
    loadModel(AssetConstants::ModelNames::CUPHEAD_BRAZO_DERECHO, AssetConstants::ModelPaths::CUPHEAD_BRAZO_DE
    loadModel(AssetConstants::ModelNames::CUPHEAD_ANTEBRAZO_IZQUIERDO, AssetConstants::ModelPaths::CUPHEAD_AN
    loadModel(AssetConstants::ModelNames::CUPHEAD_ANTEBRAZO_DERECHO, AssetConstants::ModelPaths::CUPHEAD_ANTE
    loadModel(AssetConstants::ModelNames::CUPHEAD_MUSLO_IZQUIERDO, AssetConstants::ModelPaths::CUPHEAD_MUSLO_
    loadModel(AssetConstants::ModelNames::CUPHEAD_MUSLO_DERECHO, AssetConstants::ModelPaths::CUPHEAD_MUSLO_DE
    loadModel(AssetConstants::ModelNames::CUPHEAD_PIE_IZQUIERDO, AssetConstants::ModelPaths::CUPHEAD_PIE_IZQU
    loadModel(AssetConstants::ModelNames::CUPHEAD_PIE_DERECHO, AssetConstants::ModelPaths::CUPHEAD_PIE_DERECH

    // Cargar modelos de Isaac (jerárquico)
    loadModel(AssetConstants::ModelNames::ISAAC_CUERPO, AssetConstants::ModelPaths::ISAAC_CUERPO);
    loadModel(AssetConstants::ModelNames::ISAAC_CABEZA, AssetConstants::ModelPaths::ISAAC_CABEZA);
    loadModel(AssetConstants::ModelNames::ISAAC_BRAZO_IZQUIERDO, AssetConstants::ModelPaths::ISAAC_BRAZO_IZQU
    loadModel(AssetConstants::ModelNames::ISAAC_BRAZO_DERECHO, AssetConstants::ModelPaths::ISAAC_BRAZO_DERECH
    loadModel(AssetConstants::ModelNames::ISAAC_PIERNA_IZQUIERDA, AssetConstants::ModelPaths::ISAAC_PIERNA_IZ
    loadModel(AssetConstants::ModelNames::ISAAC_PIERNA_DERECHA, AssetConstants::ModelPaths::ISAAC_PIERNA_DERE

    // Cargar modelos de Gojo (jerarqu♦a)
    loadModel(AssetConstants::ModelNames::GOJO, AssetConstants::ModelPaths::GOJO);
    loadModel(AssetConstants::ModelNames::GOJO_BRAZO_IZQ, AssetConstants::ModelPaths::GOJOBRAZOIZQ);
    loadModel(AssetConstants::ModelNames::GOJO_BRAZO_DER, AssetConstants::ModelPaths::GOJOBRAZODER);
    loadModel(AssetConstants::ModelNames::GOJO_PIERNA_IZQ, AssetConstants::ModelPaths::GOJOPIERNAIZQ);
    loadModel(AssetConstants::ModelNames::GOJO_PIERNA_DER, AssetConstants::ModelPaths::GOJOPIERNADER);
    loadModel(AssetConstants::ModelNames::GOJO_RODILLA_IZQ, AssetConstants::ModelPaths::GOJORODILLAIZQ);
    loadModel(AssetConstants::ModelNames::GOJO_RODILLA_DER, AssetConstants::ModelPaths::GOJORODILLADER);

    //Modelo prueba escenario
    loadModel(AssetConstants::ModelNames::BLACKHOLE, AssetConstants::ModelPaths::BLACKHOLE);
    loadModel(AssetConstants::ModelNames::PYRAMIDEMUSEO, AssetConstants::ModelPaths::PYRAMIDEMUSEO);
```

- TextureManager: Opera de forma análoga al ModelManager, pero se especializa en cargar, almacenar y gestionar las texturas, facilitando su acceso a través de un nombre.

```cpp
#include "TextureManager.h"

// Carga todas las texturas al inicializar el TextureManager
TextureManager::TextureManager()
{
    loadTexture(AssetConstants::TextureNames::AGAVE, AssetConstants::TexturePaths::AGAVE_PATH);
    loadTexture(AssetConstants::TextureNames::PASTO, AssetConstants::TexturePaths::PASTO_PATH);
    loadTexture(AssetConstants::TextureNames::LADRILLO, AssetConstants::TexturePaths::LADRILLO_PATH);
    loadTexture(AssetConstants::TextureNames::TIERRA, AssetConstants::TexturePaths::TIERRA_PATH);
    loadTexture(AssetConstants::TextureNames::EMPEDRADO, AssetConstants::TexturePaths::EMPEDRADO_PATH);
    loadTexture(AssetConstants::TextureNames::AGUA, AssetConstants::TexturePaths::AGUA_PATH);
    loadTexture(AssetConstants::TextureNames::MAYAN_BRICKS, AssetConstants::TexturePaths::MAYAN_BRICKS_PATH);
    // Cargar texturas de Cuphead
    loadTexture(AssetConstants::TextureNames::CUPHEAD_TEXTURE, AssetConstants::TexturePaths::CUPHEAD_TEXTURE_
    loadTexture(AssetConstants::TextureNames::POPOTE_ROJO, AssetConstants::TexturePaths::POPOTE_ROJO_PATH);

    loadTexture(AssetConstants::TextureNames::CAUCHO, AssetConstants::TexturePaths::CAUCHO_PATH);
}

// Obtiene una textura por su nombre
Texture* TextureManager::getTexture(const std::string& textureName)
{
    auto it = textures.find(textureName);
    if (it != textures.end()) {
        return it->second;
    }
    return nullptr;
}

// Carga una textura en el manager
void TextureManager::loadTexture(const std::string& textureName, const std::string& texturePath)
{
    Texture* texture = new Texture(texturePath.c_str());
    if (texture->LoadTextureA()) {
```

- SkyboxManager: Administra la creación y gestión de las diferentes skyboxes (cajas de cielo) utilizables en la escena. Facilita el cambio de la skybox activa mediante la obtención de instancias por nombre.
- MaterialManager: Centraliza la creación y gestión de los distintos materiales (que definen propiedades como el brillo, especularidad, etc.) que se aplican a los objetos de la escena.
- LightManager: Crea y gestiona las "luces base" o plantillas (puntuales, direccionales, spotlights) disponibles. Es importante notar que este manejador solo define las luces; no decide cuáles se renderizan en un fotograma específico. Las posiciones de estas luces se definen jerárquicamente más adelante.

Ninguno de estos archivos manager se encarga de la lógica de renderizado; su única función es cargar, crear y facilitar el acceso a los recursos.

2. Estructura de la Escena

Entidad: Representa a cualquier objeto individual en la escena. Almacena:

Tipo de Entidad: Indica si es un mesh o un model.

Identificadores: Nombres para la entidad, modelo, texturas y materiales.

Jerarquía: Un vector de entidades hijas, permitiendo construir una escena jerárquica.

Transformaciones: Variables para posición, rotación y escalado (local, inicial y la matriz de transformación final).

Manejo de Rotación con Quaternions: Para evitar problemas de orden de aplicación y ambigüedad la rotación se maneja con quaternions. La variable rotacionLocalQuat almacena la orientación actual acumulada. Las variables de rotación (p.ej. rotacionActual) almacenan los incrementos deseados para el siguiente frame. En la actualización, este incremento se aplica al quaternion principal y el incremento se reinicia a cero.

Componentes: La clase Entidad también funciona como un contenedor de componentes, pudiendo tener instancias de:

ComponenteFisico

ComponenteAnimacion

- ComponenteFisico: modela las propiedades físicas y el comportamiento de la entidad (colisiones, gravedad, etc.).
- ComponenteAnimacion: Encapsula toda la lógica de animación para una entidad (puede ser nullptr si la entidad no se anima).

- Actualización: Contiene la función actualizarAnimacion(entidad, deltaTime), que gestiona las animaciones (cíclicas o activables) basándose en el nombre de la entidad.
- Gestión de Estado: Utiliza una variable short como un conjunto de banderas para identificareficientemente el estado (activo/inactivo) de hasta 16 animaciones simultáneas.
- Animación por Keyframes: Incluye soporte completo para animaciones basadas en keyframes, gestionando la carga de estos desde archivos de texto, la interpolación y la reproducción.

3. Orquestación y Renderizado

- SceneInformation: Este es el archivo principal que cohesiona la escena (equivalente al main lógico). Contiene toda la información de la escena:
- Gestión de Entidades: Almacena el arreglo principal de entidades raíz.
- Vinculación (Binding): Incluye la función vincularEntidad, que conecta las entidades con sus recursos correspondientes (modelos, texturas, materiales) usando los nombres almacenados.
- Cámara: Almacena la instancia de la cámara activa.
- Lógica de Iluminación: Mantiene los arreglos de las luces actualmente activas que deben renderizarse.
- Ciclo Día/Noche: Gestiona la transición de la luz direccional y la skybox mediante un temporizador.
- actualizarFrame(deltaTime): Para actualizaciones cíclicas (animaciones, ciclo día/noche).
- actualizarFrameInput(keys, deltaTime): Para actualizaciones basadas en la entrada del teclado (cambio de personaje, inicio de animaciones).
- Selección de Luces: Si hay más de 5 luces pointlight prendidas, solo se seleccionan y envían al shader las 5 más cercanas a la cámara.

- SceneRenderer: Esta clase es la única responsable del renderizado. Toma la información de SceneInformation (entidades, luces activas, skybox, cámara, proyección) y ejecuta los siguientes pasos por fotograma:
    - Renderiza la skybox.
    - Activa el shader principal.
    - Envía las luces activas (direccional, puntuales, spotlights) a los uniforms del shader.
    - Renderiza todas las entidades.
    - Renderizado Jerárquico: El renderizado de entidades es recursivo, utilizando un enfoque de búsqueda en profundidad (DFS). La función de renderizado recibe la matriz de transformación del padre, la combina con la transformación local de la entidad actual, y pasa el resultado a sus hijas. Antes de dibujar, se actualiza la matriz de modelo de la entidad, se determina si es mesh o model, y se invocan las funciones de dibujado correspondientes (renderMesh o renderModelo), aplicando las texturas y materiales vinculados.

EVIDENCIA LINEAMIENTOS:

Algunos lineamientos se saltan ya que por ejemplo lo del texturizado de elementos se puede observar en las demás.

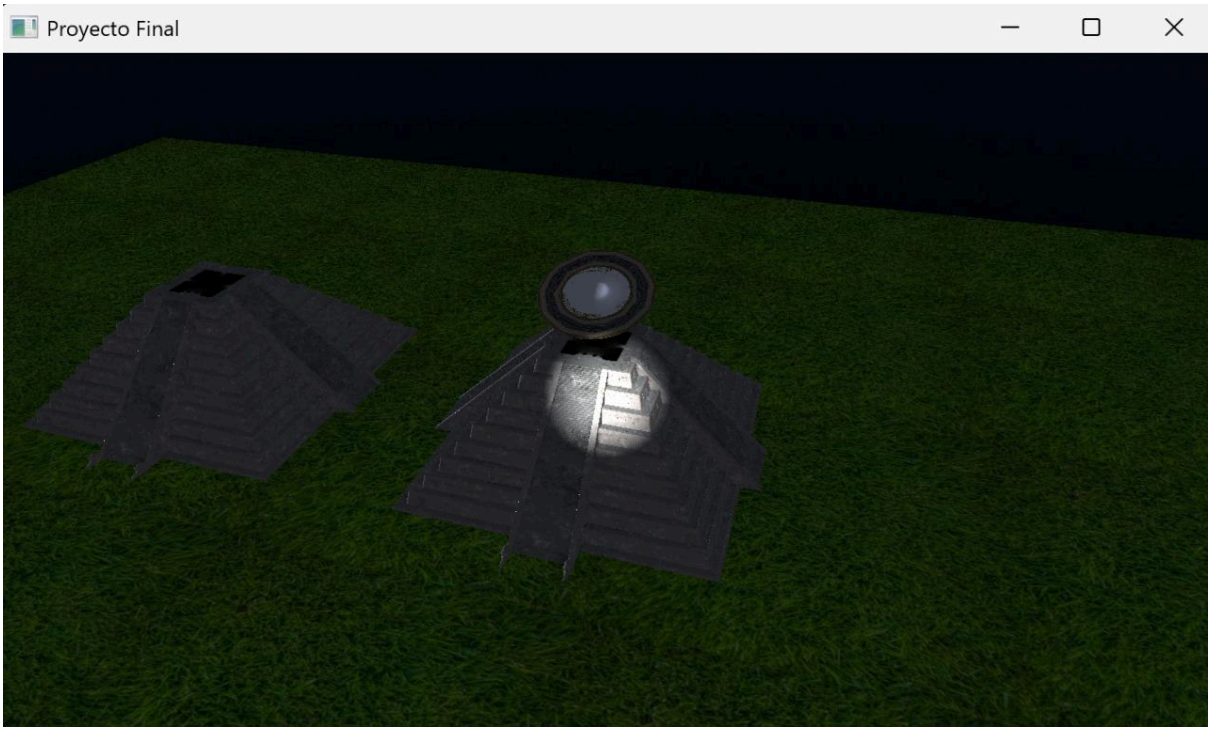
NPCS

PERSONAJES

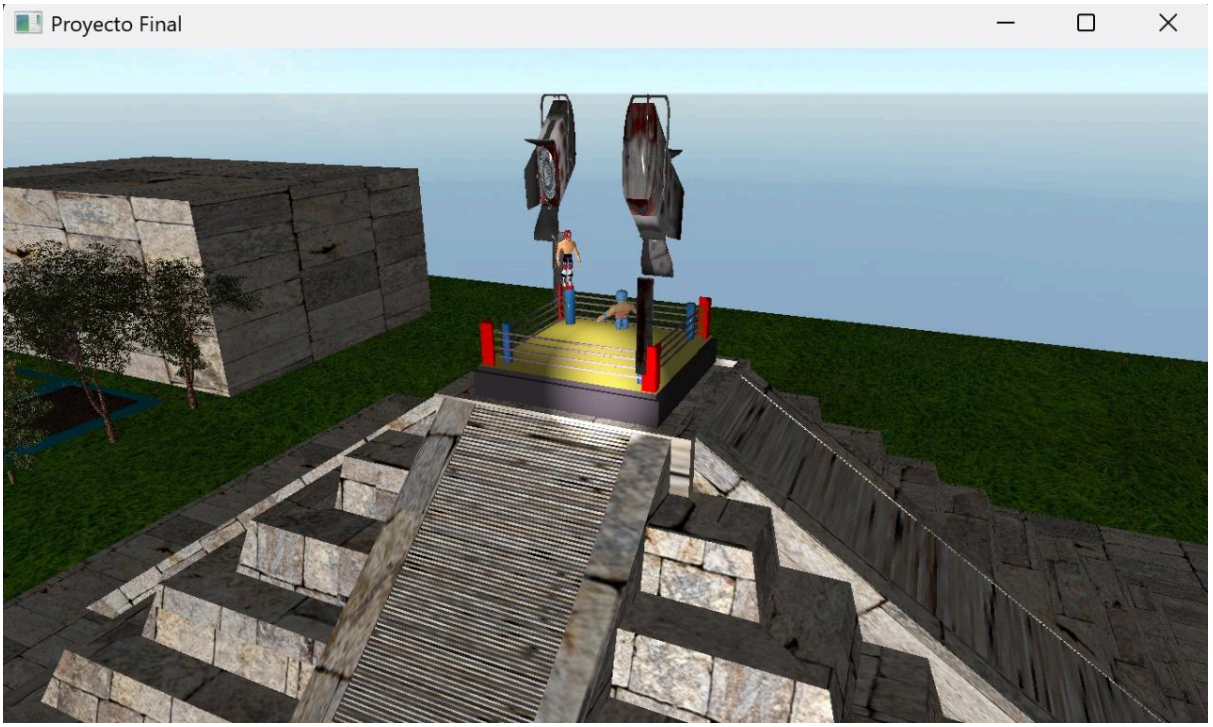

Proyecto Final


Proyecto Final

CÁMARAS



LUCES


Proyecto Final


Proyecto Final

Referencias

OpenGL tutorial. (s.f).Tutorial 17 : Rotations.
https://www.opengl-tutorial.org/intermediate-tutorials/tutorial-17-quaternions/