

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA
INGENIERÍA EN COMPUTACIÓN

*COMPUTACIÓN GRÁFICA E INTERACCIÓN HUMANO
COMPUTADORA*

NOMBRE: Becerra Lara Alison

No de Cuenta: 319233100

GRUPO DE LABORATORIO: 04

GRUPO DE TEORÍA: 04

SEMESTRE 2026-1

MANUAL TÉCNICO

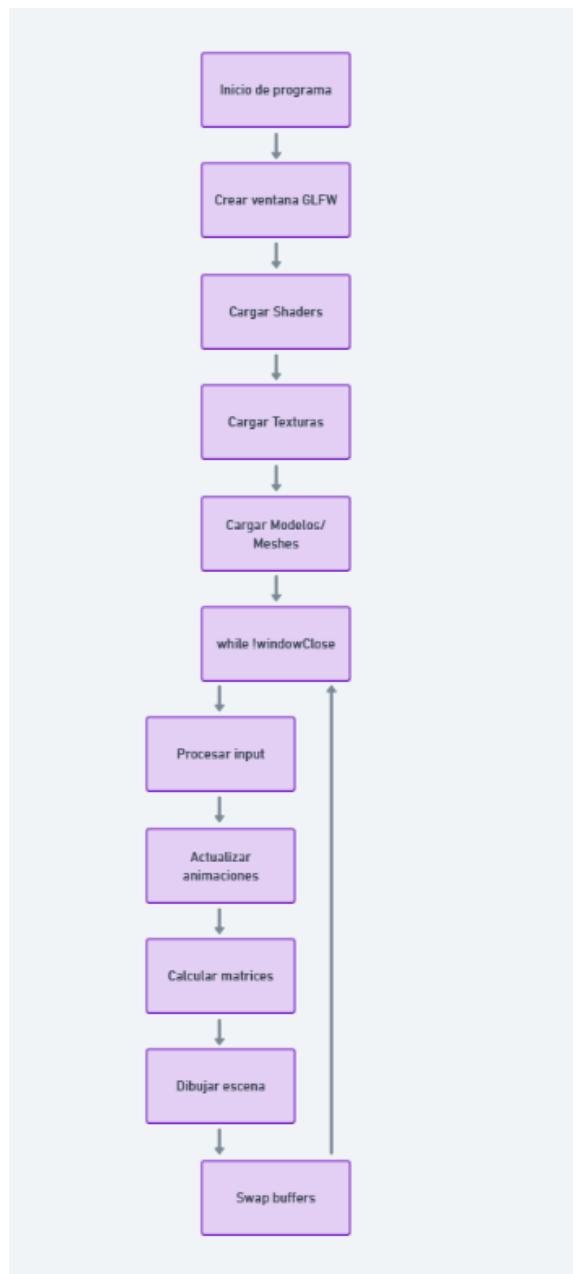
FECHA DE ENTREGA: 15 de noviembre de 2025

Manual Técnico del Zoológico Virtual

1. Objetivos

Desarrollar una escena 3D interactiva utilizando OpenGL que integre modelado básico, animación por transformaciones, texturizado, iluminación dinámica y control por teclado, con el fin de simular un hábitat virtual compuesto por animales animados, y elementos ambientales, permitiendo visualizar y comprender el funcionamiento de un programa gráfico aplicando los conocimientos adquiridos en el curso de Computación Gráfica e Interacción Humano Computadora.

2. Diagrama de flujo de software



Descripción:

Bloque: Inicio del programa

Flecha hacia: Crear ventana GLFW

Descripción: El programa comienza su ejecución.

Bloque: Crear ventana GLFW

Flecha hacia: Cargar Shaders

Descripción: Se inicializa GLFW y se crea la ventana principal de OpenGL.

Bloque: Cargar Shaders

Flecha hacia: Cargar Texturas

Descripción: Se compilan y enlazan los shaders del proyecto (Lighting, Skybox, etc.).

Bloque: Cargar Texturas

Flecha hacia: Cargar Modelos/Meshes

Descripción: Se cargan las texturas desde archivos y se preparan para usarse en OpenGL.

Bloque: Cargar Modelos/Meshes

Flecha hacia: while (!windowClose)

Descripción: Se cargan los modelos .obj, mallas y buffers de vértices.

Ciclo principal

Bloque: while (!windowClose)

Flecha hacia: Procesar input

Descripción: Inicia el loop principal del programa mientras la ventana siga abierta.

Bloque: Procesar input

Flecha hacia: Actualizar animaciones

Descripción: Se leen teclas, mouse y eventos.

Bloque: Actualizar animaciones

Flecha hacia: Calcular matrices

Descripción: Se actualizan posiciones, movimientos, rotaciones y estados animados.

Bloque: Calcular matrices (projection, view, model)

Flecha hacia: Dibujar escena

Descripción: Se calculan las matrices necesarias para renderizar objetos en 3D.

Bloque: Dibujar escena (objetos, modelos, luz)

Flecha hacia: Swap buffers

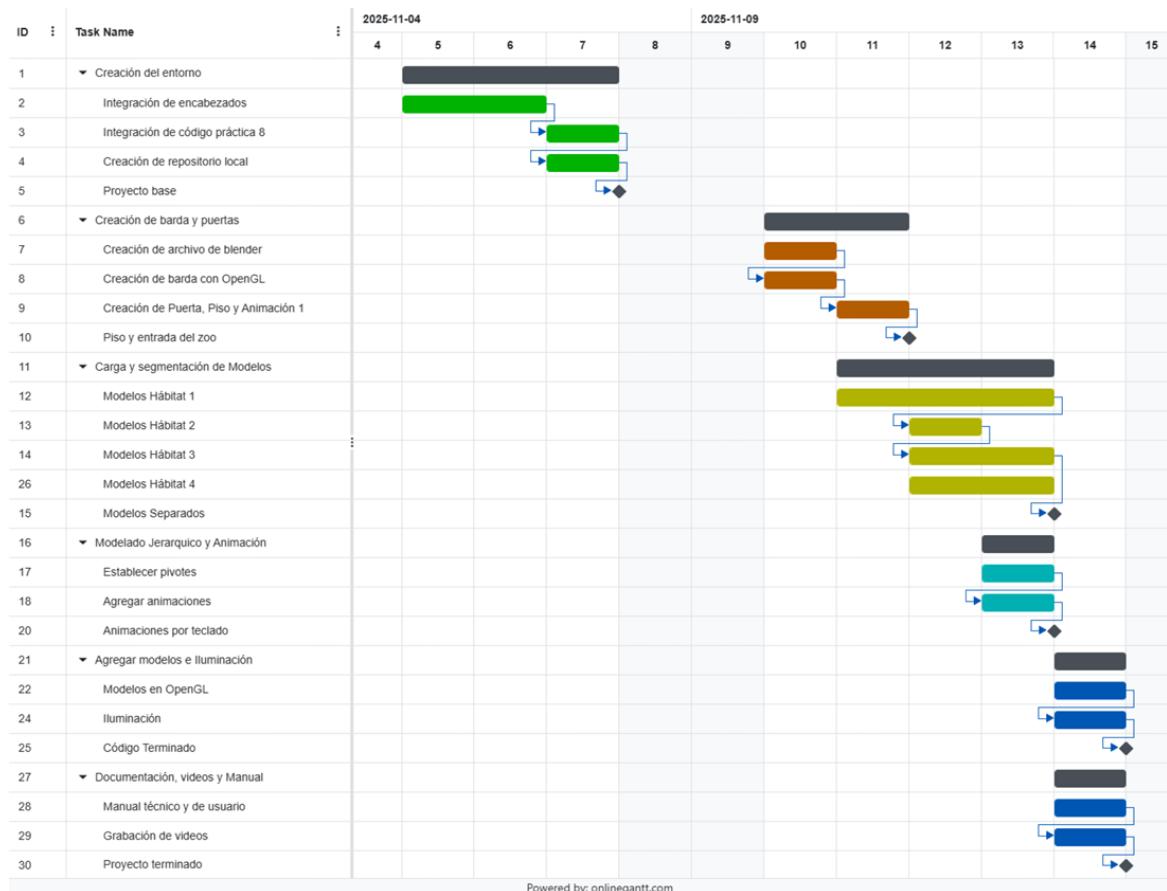
Descripción: Se dibuja todo en pantalla usando los shaders y las matrices.

Bloque: Swap buffers

Flecha hacia: while (!windowClose)

Descripción: Se presenta la imagen en pantalla y se regresa al inicio del ciclo.

3. Diagrama de Gantt



4. Alcance del proyecto

El proyecto logró un alcance completo ya que se implementó un sistema de modelado jerárquico bien estructurado donde cada animal fue dividido en múltiples partes (cabeza, cuello, cuerpo, patas, cola, alas, etc.) y cada una fue exportada correctamente con su pivote realista, lo que permitió animaciones naturales mediante transformaciones locales. Esto fue con ayuda de la herramienta de modelado Blender.

Además, se integró los conceptos de iluminación incluyendo luz direccionales, puntuales y un spotlight vinculado a la cámara. El entorno general también se construyó de manera satisfactoria mediante el ensamblaje de cubos, modelos OBJ importados y texturas.

A nivel de animación se lograron múltiples (una para cada animal), como caminar, volar, nadar, mover la cola, abrir el pico, rotar extremidades y desplazarse en ciclos. Cada animación pudo activarse o detenerse por teclas, e incluso algunas incluían interpolaciones suaves para regresar a una posición inicial.

Finalmente, se desarrolló una escena completa con objetos del entorno (bancas, bardas, árboles, aviario), un skybox y una cámara que permite la exploración total del espacio.

En conjunto el alcance técnico del proyecto fue **satisfactorio**, cubriendo modelado jerárquico, animación por transformaciones, texturizado, iluminación, carga de modelos.

5. Limitantes

Aunque el resultado final fue funcional existieron limitaciones principalmente relacionadas con el modelado directo de objetos dentro de OpenGL. El mayor reto fue que crear modelos desde cero usando vértices y coordenadas de textura manuales, se volvió complejo y tardado, no fue práctico elaborar objetos complejos completamente desde código.

Como consecuencia, se optó por combinar y algunos objetos se crearon directamente en OpenGL mediante primitivas. Otros objetos más detallados se cargaron o modelaron en Blender y se importaron como archivos .obj, lo cual simplificó enormemente la construcción del entorno. Otra limitación fue la complejidad de mantener animaciones sincronizadas y naturales cuando intervenían muchos componentes al mismo tiempo, especialmente en animales articulados.

Otra de las limitantes fue que el trabajo estaba pensado para ser en equipo pero fue hecho individualmente por mí, por lo que no se tuvo la mejor planeación de tiempos.

6. M^{et}todología de software aplicada: **Extreme Programming XP**

Esta es una metodología ágil enfocada en ciclos cortos, retroalimentación constante y mejoras rápidas. Por las limitaciones de tiempo, se decidió trabajar en capas que producen una versión funcional parcialmente completa.

Desarrollo incremental

- Creación del entorno
- Creación de bardas y puertas
- Modelado y segmentación
- Animaciones
- Iluminación
- Integración final

Integración continua: El proyecto se basó en el siguiente ciclo

- Exportaba un modelo
- Lo probaba en OpenGL
- Ajustaba pivotes
- Aplicaba una animación
- Lo volvía a integrar

Refactorización constante

Durante el proyecto refiné varias veces:

- Animaciones
- Pivotes
- Organización del Main
- Carga de modelos
- Shader de lightning.frag

En cada iteración agregaba algo funcional y al final de acabar todo, notaba que podía refine, por ejemplo, al inicio el venado solo caminaba y se salía de la barda del zoológico, después de unas cuántas iteraciones arreglé esta parte.

Retroalimentación rápida

El avance del proyecto siempre dependió de correcciones inmediatas, anotaba en el código las partes que se veían extrañas, pivotes incorrectos, cargas inconclusas de texturas, luces incorrectas.

Pruebas funcionales

Se decidió solo una tecla por hábitat y luces para probar en cada uno:

- Animación de patas en hábitat de bosque
- Caminado adelante-atrás en hábitat polar
- Vuelo de pájaro y pico del tucán
- Movimiento de piraña y tiburón
- Transparencias con el shader
- Iluminación

Actividad del proyecto	Componente de XP	Descripción
Creación del entorno	Iteraciones cortas	Se desarrolló lo mínimo necesario para avanzar rápido.
Integración de encabezados y código base	Integración continua	Cada módulo se probaba y unía de inmediato.
Modelado y pivotes	Refactorización	Se rehicieron varias partes hasta lograr la alineación correcta.
Animaciones	Retroalimentación rápida	Se probaban visualmente y se ajustaban al instante.
Iluminación	Iteración incremental	La iluminación se mejoró gradualmente conforme avanzaba el proyecto.
Modelado jerárquico	Diseño simple	Al principio se usaron coordenadas basadas en blender y se fue refinando.
Documentación y videos	Elementos no funcionales al final	Se elaboraron después del funcionamiento total.

7. Documentación del código

Inclusión de librerías y declaración de estado global

Al inicio del archivo se incluyen las librerías de C++ estándar, las de OpenGL/GLFW y las de GLM. Eso permite manejar ventanas, contexto de render, vectores, matrices y operaciones matemáticas 3D. También se incluyen clases propias del proyecto, como Shader, Camera, Model, Mesh, Texture y Window, que encapsulan tareas comunes (carga de modelos, manejo de cámara, texturas, etc.) para no tener todo el código “crudo” de OpenGL en el main.



```
#include <iostream>
#include <vector>
#include <cmath>
#include <algorithm>

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

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

#include "Shader.h"
#include "Camera.h"
#include "Model.h"
#include "Mesh.h"
#include "Texture.h"
#include "Window.h"
```

Después de los includes se declaran varias funciones auxiliares y un conjunto grande de variables globales. Esas variables representan el estado de la escena: posición de la cámara, luces puntuales, si las luces están encendidas o apagadas, el estado de la puerta de entrada, y toda la información necesaria para las animaciones de cada hábitat (ciervo adulto y bebé, pájaros, oso polar, pingüino, foca, tiburón, piraña). En resumen, esta parte define todo el “estado vivo” del zoológico: dónde está cada animal, cómo se mueve, y qué controles hay para encender o apagar sus animaciones.

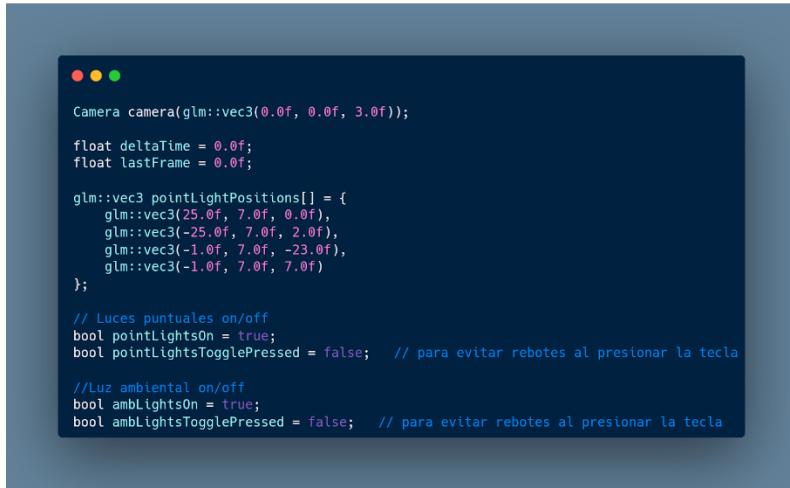


```
void CrearObjeto(GLuint& VAO, GLuint& VBO, GLuint& EBO,
                  GLfloat* vertices, GLuint* indices, int tamV, int tamI);
void CrearObjetoSkyBox(GLuint& VAO, GLuint& VBO,
                      GLfloat* vertices, int tamV);
void DibujarBarda(const glm::vec3& posicion, float rotacionY,
                  float largo, GLuint VAO, GLuint textura, GLint modelLoc);
void ProcessInput(Window& window);
void Animation();
```

Estado inicial de la escena: cámara, luces, puerta y parámetros globales

Esta primera sección del código define todas las variables fundamentales que controlan el comportamiento del mundo 3D. Aquí se establece la posición inicial de la cámara, así como las variables de tiempo (deltaTime y lastFrame) que permiten que las animaciones y los movimientos sean suaves y consistentes, sin depender del número de frames por segundo.

Además, se configuran las luces puntuales y la luz ambiental, junto con sus banderas de encendido y apagado. Estas banderas permiten controlar la iluminación de la escena mediante teclas, evitando que un solo toque genera múltiples cambios. También se definen las variables que gobiernan la apertura de la puerta principal del zoológico, incluyendo su ángulo de giro y la velocidad con la que se abre o cierra.



A continuación se declaran las variables para los tres pájaros: uno volador, otro rotatorio y un tucán.

Esta sección prepara sus posiciones iniciales, rotaciones y banderas de animación. El primer pájaro incluye parámetros especializados que controlan el vuelo, tales como amplitudes en X y Y, inclinación, altura de planeo y frecuencias de aleteo.

Para el tucán se integran valores adicionales que controlan la apertura del pico y las rotaciones de cabeza y cola, con límites que aseguran un comportamiento natural. En conjunto, esta parte define el estado inicial y los parámetros de movimiento de las aves del aviario.



```
// Animación para el hábitat 2 (Cervos)
bool animacionH2 = false;
bool animacionH2TogglePressed = false;

// Animaciones ciervos
glm::vec3 ciervoPos(0.0f, 0.0f, 0.0f);
float ciervoRot = 0.0f;
float head = 0.0f;
float FLegs = 0.0f;
float RLegs = 0.0f;
bool Ciervostep = false;

float ciervoSegLen = 5.0f; // distancia por tramo
float ciervoStep = 0.01f; // avance por frame
float ciervoMoved = 0.0f; // acumulado de la distancia recorrida en el tramo
int ciervoDir = +1; // +1 hacia +Z, -1 hacia -Z
bool ciervoTurning = false; // está girando
float ciervoTurnLeft = 180.0f; // grados por girar cuando toca
float ciervoTurnSpeed = 2.0f; // grados por frame (giro)

// Punto/rotación originales del ciervo
glm::vec3 ciervoHomePos(0.0f, 0.0f, 0.0f);
float ciervoHomeRot = 0.0f;

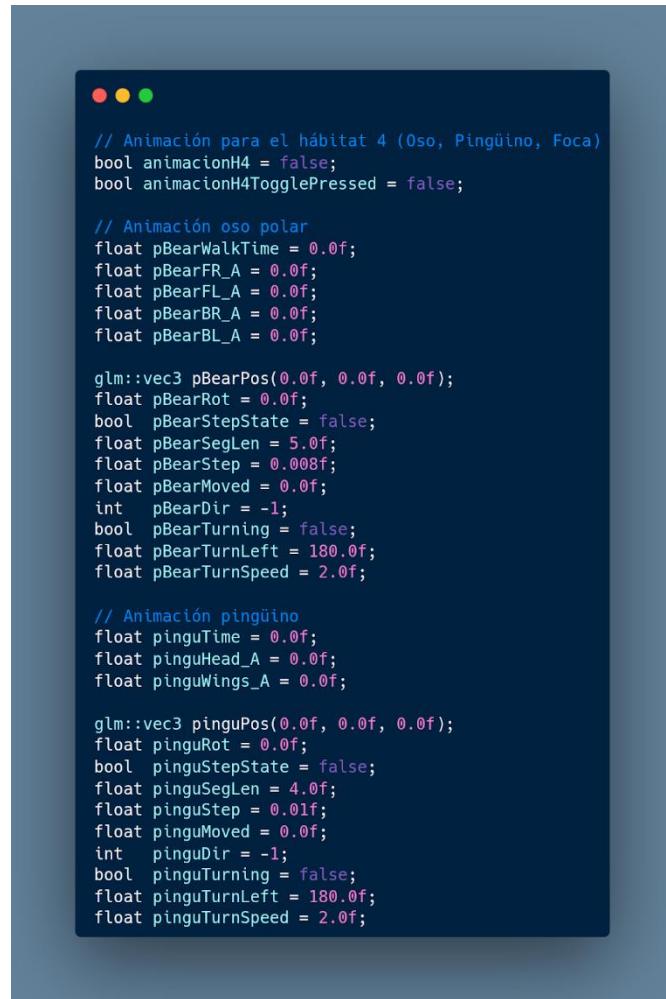
// Velocidad de regreso
float ciervoReturnPosStep = 0.02f; // unidades por frame (XYZ)
float ciervoReturnRotStep = 2.0f; // grados por frame

bool puertaTogglePressed = false;
bool ciervoTogglePressed = false; // Flag para el ciervo adulto

// Animación Ciervo Bebé
bool Ciervo2Anim = false;
bool ciervobebetogglePressed = false;
float head2 = 0.0f;
float neck = 0.0f;
```

Esta sección agrupa todas las variables que controlan el hábitat polar. Aquí se activa o desactiva la animación completa y se definen los estados individuales del oso, del pingüino y de la foca.

Cada animal tiene parámetros propios: posición, rotación, avance dentro de una ruta, y ángulos específicos de movimiento de extremidades. A través de estas variables se consigue que los animales avancen, giren, oscilen y se comporten de manera coherente dentro de su área polar.



```
// Animación para el hábitat 4 (Oso, Pingüino, Foca)
bool animacionH4 = false;
bool animacionH4TogglePressed = false;

// Animación oso polar
float pBearWalkTime = 0.0f;
float pBearFR_A = 0.0f;
float pBearFL_A = 0.0f;
float pBearBR_A = 0.0f;
float pBearBL_A = 0.0f;

glm::vec3 pBearPos(0.0f, 0.0f, 0.0f);
float pBearRot = 0.0f;
bool pBearStepState = false;
float pBearSegLen = 5.0f;
float pBearStep = 0.008f;
float pBearMoved = 0.0f;
int pBearDir = -1;
bool pBearTurning = false;
float pBearTurnLeft = 180.0f;
float pBearTurnSpeed = 2.0f;

// Animación pingüino
float pinguTime = 0.0f;
float pinguHead_A = 0.0f;
float pinguWings_A = 0.0f;

glm::vec3 pinguPos(0.0f, 0.0f, 0.0f);
float pinguRot = 0.0f;
bool pinguStepState = false;
float pinguSegLen = 4.0f;
float pinguStep = 0.01f;
float pinguMoved = 0.0f;
int pinguDir = -1;
bool pinguTurning = false;
float pinguTurnLeft = 180.0f;
float pinguTurnSpeed = 2.0f;
```

Aquí se definen todas las variables relacionadas con los peces del acuario.

El tiburón tiene parámetros que permiten animar su cuerpo con ondas mediante funciones seno, así como su movimiento dentro de un rango limitado con un patrón en zigzag. Las señales de dirección controlan cuándo debe cambiar de rumbo.

La piraña incluye una animación más compacta, con un movimiento de avance–retroceso y un giro suave basado en un acumulador. Estas variables permiten que la animación de cada especie sea independiente y consistente.

```
// Animación para el hábitat 1 (Tiburón, Piraña)
bool animacionH1 = false;
bool animacionH1TogglePressed = false;

// Animación del tiburón
float sharkTime = 0.0f;
float sharkHeadAngle = 0.0f;
float sharkTailA = 0.0f;
float sharkBodyAngle = 0.0f; // Ángulo para oscilar el cuerpo

glm::vec3 sharkPos(0.0f);
float sharkRot = 0.0f;
float sharkZDir = 1.0f; // dirección en Z: +1 adelante, -1 atrás
float sharkXDir = 1.0f; // dirección en X: zigzag izquierda/derecha

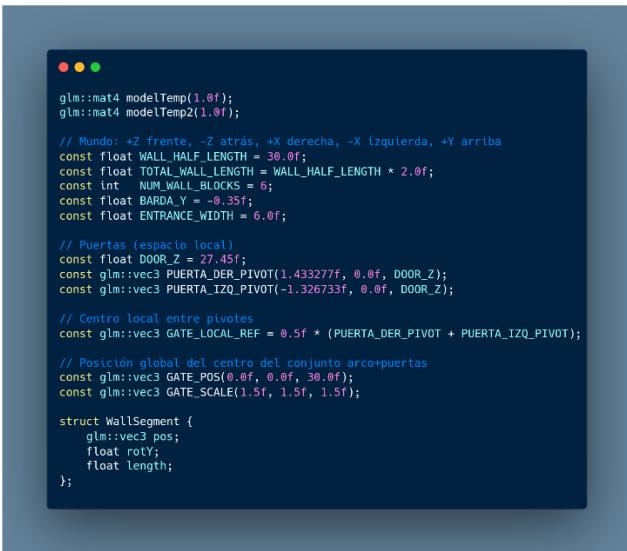
// Animación piraña
float piraHeadAngle = 0.0f;
float piraTailAngle = 0.0f;
float piraTime = 0.0f; // tiempo para animación
glm::vec3 piranhaPos(0.0f);
float piranhaRot = 0.0f;
// Direcciones ping-pong (+1)
// --- Para rotación suave sin trigonometría ---
float piraRotAngle = 0.0f; // acumulador de rotación
float piraDirRot = 1.0f; // +1 gira derecha, -1 gira izquierda
```

Herramientas temporales y parámetros geométricos del recinto

Esta parte reúne matrices auxiliares usadas como buffers temporales para transformaciones jerárquicas dentro del render.

También define parámetros geométricos del zoológico, como la extensión de las bardas, el número de segmentos, la posición de la entrada, los pivotes de las puertas y un punto de referencia global para todo el arco del portón.

La estructura Wall Segment permite manejar las piezas de la barda como módulos individuales, facilitando generar el recinto de forma flexible y ordenada.



```

// Mundo: +Z frente, -Z atrás, +X derecha, -X izquierda, +Y arriba
const float WALL_HALF_LENGTH = 30.0f;
const float TOTAL_WALL_LENGTH = WALL_HALF_LENGTH * 2.0f;
const int NUM_WALL_BLOCKS = 6;
const float BARDAS_Y = -0.35f;
const float ENTRANCE_WIDTH = 6.0f;

// Puertas (espacio local)
const float DOOR_Z = 27.45f;
const glm::vec3 PUERTA_DER_PIVOT(1.433277f, 0.0f, DOOR_Z);
const glm::vec3 PUERTA_IZQ_PIVOT(-1.326733f, 0.0f, DOOR_Z);

// Centro local entre pivotes
const glm::vec3 GATE_LOCAL_REF = 0.5f * (PUERTA_DER_PIVOT + PUERTA_IZQ_PIVOT);

// Posición global del centro del conjunto arco+puertas
const glm::vec3 GATE_POS(0.0f, 0.0f, 30.0f);
const glm::vec3 GATE_SCALE(1.5f, 1.5f, 1.5f);

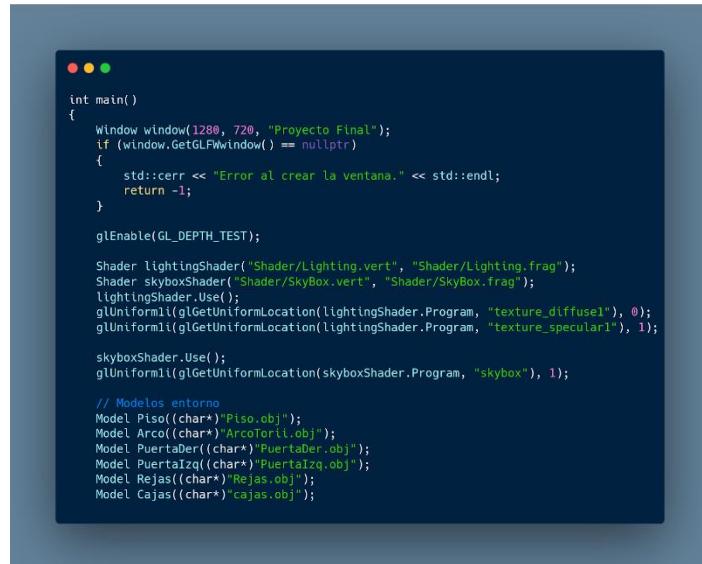
struct WallSegment {
    glm::vec3 pos;
    float rotY;
    float length;
};

```

Función main: creación de ventana, contexto OpenGL y shaders

En el main() se crea primero la ventana mediante la clase Window. Si la creación falla, se imprime un mensaje de error y el programa termina. Una vez que la ventana existe, se habilita el DEPTH_TEST para que OpenGL pueda manejar correctamente qué objetos quedan delante o detrás en función de la profundidad.

Después se instancian los shaders principales: uno para iluminación (lightingShader) y otro para el skybox (skyboxShader). En ese momento se configurarán también los samplers de textura (por ejemplo, difusa y especular) para que los modelos sepan en qué unidad de textura buscar sus mapas. En esencia, aquí se arma el “pipeline” básico de render: ventana, contexto de OpenGL y programas de shader que van a encargarse de dibujar la escena.



```

int main()
{
    window window(1280, 720, "Proyecto_Final");
    if (window.GetGLFWwindow() == nullptr)
    {
        std::cerr << "Error al crear la ventana." << std::endl;
        return -1;
    }

    glEnable(GL_DEPTH_TEST);

    Shader lightingShader("Shader/Lighting.vert", "Shader/Lighting.frag");
    Shader skyboxShader("Shader/SkyBox.vert", "Shader/SkyBox.frag");
    lightingShader.Use();
    glUniform1i(glGetUniformLocation(lightingShader.Program, "texture_diffuse1"), 0);
    glUniform1i(glGetUniformLocation(lightingShader.Program, "texture_specular1"), 1);

    skyboxShader.Use();
    glUniform1i(glGetUniformLocation(skyboxShader.Program, "skybox"), 1);

    // Modelos entorno
    Model Piso((char*)"Piso.obj");
    Model Arco((char*)"ArcoToril.obj");
    Model PuertaDer((char*)"PuertaDer.obj");
    Model PuertaIzq((char*)"PuertaIzq.obj");
    Model Rejas((char*)"Rejas.obj");
    Model Cajas((char*)"cajas.obj");
}

```

Carga de modelos: entorno, animales y elementos decorativos

Una parte importante del main es la creación de muchos objetos Model. Cada uno carga un archivo .obj distinto: el piso, el arco de entrada, las puertas, las rejas, las cajas, los modelos del ciervo adulto (cuerpo, cabeza y patas), el ciervo bebé, el pingüino, la foca, el oso polar, el tiburón, la piraña, el iglú, bardas metálicas y de vidrio, la pecera, el agua, la arena, los corales, las bancas, el aviario, el árbol interior, las hojas, los arbustos y los distintos pájaros.

Estos modelos provienen de Blender u otro programa de modelado y se integran en la escena a través de la clase Model. Esta sección, en pocas palabras, es la que conecta los “assets” del proyecto con el motor: por cada elemento visible del zoológico hay un Model cargado desde disco listo para dibujarse con un shader.



```
// Ciervo adulto
Model CiervoBody((char*)"Ciervo_cuerpo.obj");
Model CiervoHead((char*)"Ciervo_head.obj");
Model CiervoF_LeftLeg((char*)"Ciervo_fleftleg.obj");
Model CiervoF_RightLeg((char*)"Ciervo_frightleg.obj");
Model CiervoB_LeftLeg((char*)"Ciervo_bleftleg.obj");
Model CiervoB_RightLeg((char*)"Ciervo_brightleg.obj");

// Flora / entorno extra
Model CiervoArbusto((char*)"Ciervo_arbusto.obj");
Model CiervoRamas((char*)"Ciervo_arbustoramas.obj");
Model ArbustosReja((char*)"Arbustos_reja.obj");
Model Pinos((char*)"PinosHojas.obj");
Model PinosRama((char*)"Models/PinosRama.obj");
Model Aviario((char*)"Aviario.obj");
Model AviarioV((char*)"VidrioAviario.obj");
Model Flores((char*)"Flores.obj");
Model Banca((char*)"BancasMadera.obj");
Model ArbolAv((char*)"aviarioarbol.obj");
Model HojasAv((char*)"hojasaviario.obj");
```

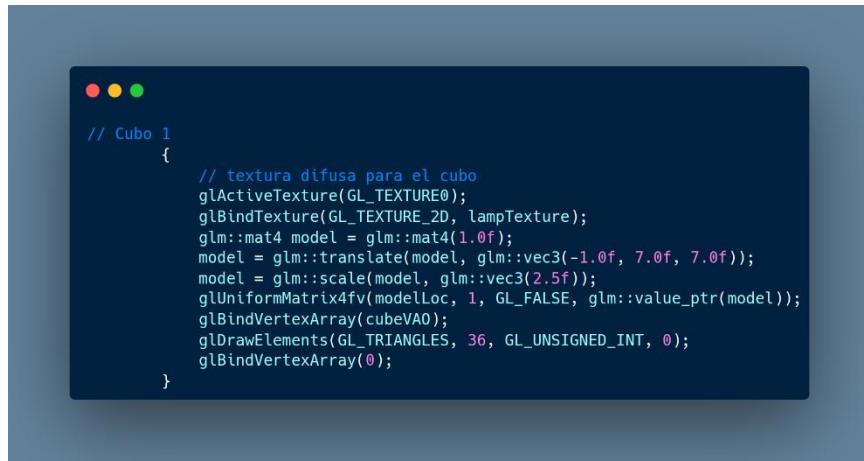
Geometrías básicas: skybox, barda y cubo parametrizable

Además de los modelos importados, el código define a mano algunos arreglos de vértices e índices. Por ejemplo, se especifican los vértices del cubo del skybox, que rodea toda la escena con una textura cúbica y simula el cielo o fondo. También se definen los vértices e índices de un cubo genérico que se usa para varias cosas (lámparas, postes, carteles) y de una “barda base”, que es otro cubo texturizado que luego se escala para formar tramos de muro más largos.

Construcción de VAOs y VBOs para cubos y skybox

Con las funciones auxiliares `CrearObjeto` y `CrearObjetoSkyBox` se generan los VAOs, VBOs y EBOs necesarios para dibujar tanto el cubo general como el de la barda y el skybox. Estas funciones encapsulan la configuración de los atributos de vértice (posición, normales, coordenadas de textura) y la subida de los datos a la GPU.

La idea es no repetir el código de configuración para cada objeto geométrico básico: una vez configurado el VAO, basta con hacer un `glBindVertexArray` y una llamada de dibujo para que OpenGL pinte el cubo con la textura que se haya activado en ese momento.

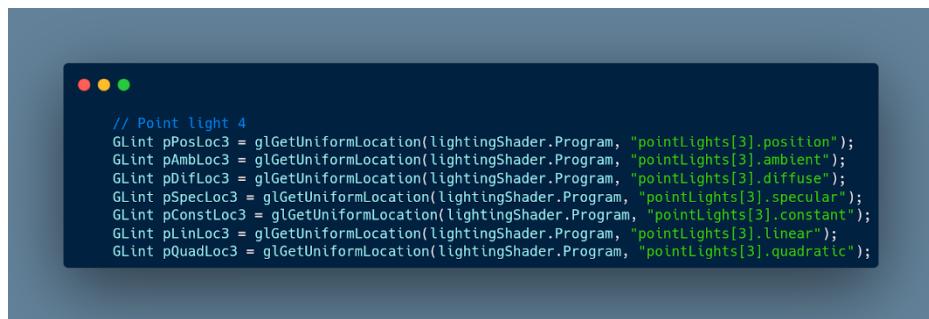


```
// Cubo 1
{
    // textura difusa para el cubo
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, lampTexture);
    glm::mat4 model = glm::mat4(1.0f);
    model = glm::translate(model, glm::vec3(-1.0f, 7.0f, 7.0f));
    model = glm::scale(model, glm::vec3(2.5f));
    glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
    glBindVertexArray(cubeVAO);
    glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_INT, 0);
    glBindVertexArray(0);
}
```

Cacheo de uniform locations para el shader de iluminación

Antes de entrar al bucle principal, el código consulta y guarda los location de múltiples variables uniform del shader de iluminación: matrices model, view, projection, posición de la cámara, parámetros de la luz direccional, de las cuatro luces puntuales, del spotlight asociado a la cámara, así como propiedades del material y banderas de transparencia.

Guardar estos GLint evita estar llamando a glGetUniformLocation muchas veces dentro del loop, lo que haría el código más lento y repetitivo. Esta sección es como preparar un “mapa” de dónde está cada uniform dentro del programa de shaders para actualizarlo eficientemente en cada frame.



```
// Point light 4
GLint pPosLoc3 = glGetUniformLocation(lightingShader.Program, "pointLights[3].position");
GLint pAmbLoc3 = glGetUniformLocation(lightingShader.Program, "pointLights[3].ambient");
GLint pIfLoc3 = glGetUniformLocation(lightingShader.Program, "pointLights[3].diffuse");
GLint pSpecLoc3 = glGetUniformLocation(lightingShader.Program, "pointLights[3].specular");
GLint pConstLoc3 = glGetUniformLocation(lightingShader.Program, "pointLights[3].constant");
GLint plinLoc3 = glGetUniformLocation(lightingShader.Program, "pointLights[3].linear");
GLint pQuadLoc3 = glGetUniformLocation(lightingShader.Program, "pointLights[3].quadratic");
```

LOOP principal de render y animación

El corazón del programa es el while (!window.ShouldClose()). En cada iteración se calcula el deltaTime a partir del tiempo actual, se procesan los eventos de la ventana, se leen las entradas de teclado y ratón con ProcessInput, y se actualizan los parámetros de animación llamando a Animation().

Después se limpia el buffer de color y de profundidad y se construyen las matrices de proyección y vista a partir de la cámara. Se activa el shader de iluminación y se envían estas matrices, así como la posición de la cámara y la configuración actual de las luces (direccional, puntuales y spotlight). A partir de ahí comienza la fase de dibujo: primero se

recorren los segmentos de barda y se dibujan usando DibujarBarda. Luego se pinta el piso, el arco, y las dos hojas de la puerta aplicando la rotación acumulada de la animación.

El resto del bucle va dibujando, en un orden lógico, cada hábitat y cada animal: el ciervo adulto y el bebé, los tres pájaros (incluyendo el tucán con su pico articulado), las rejas, arbustos, pinos y aviaro, el pingüino, la piraña y el tiburón, la foca, el oso polar, el iglú, las bardas metálicas y de vidrio del área polar, el hábitat marino (arena, corales, agua y pecera), bancas, lámparas y carteles construidos con cubos texturizados. Al final de cada frame se dibuja el skybox con su shader específico, ajustando el depth para que siempre quede detrás de los objetos de la escena, y se intercambian los buffers de la ventana.



```
// ----- LOOP PRINCIPAL -----
while (!window.ShouldClose())
{
    float currentTime = (float)glfwGetTime();
    deltaTime = currentTime - lastFrame;
    lastFrame = currentTime;

    window.PollEvents();
    ProcessInput(window);
    Animation();

    glClearColor(0.1f, 0.1f, 0.1f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glm::mat4 projection = glm::perspective(
        glm::radians(camera.GetZoom()),
        (float)window.GetBufferWidth() / (float)window.GetBufferHeight(),
        0.1f, 100.0f
    );
    glm::mat4 view = camera.GetViewMatrix();

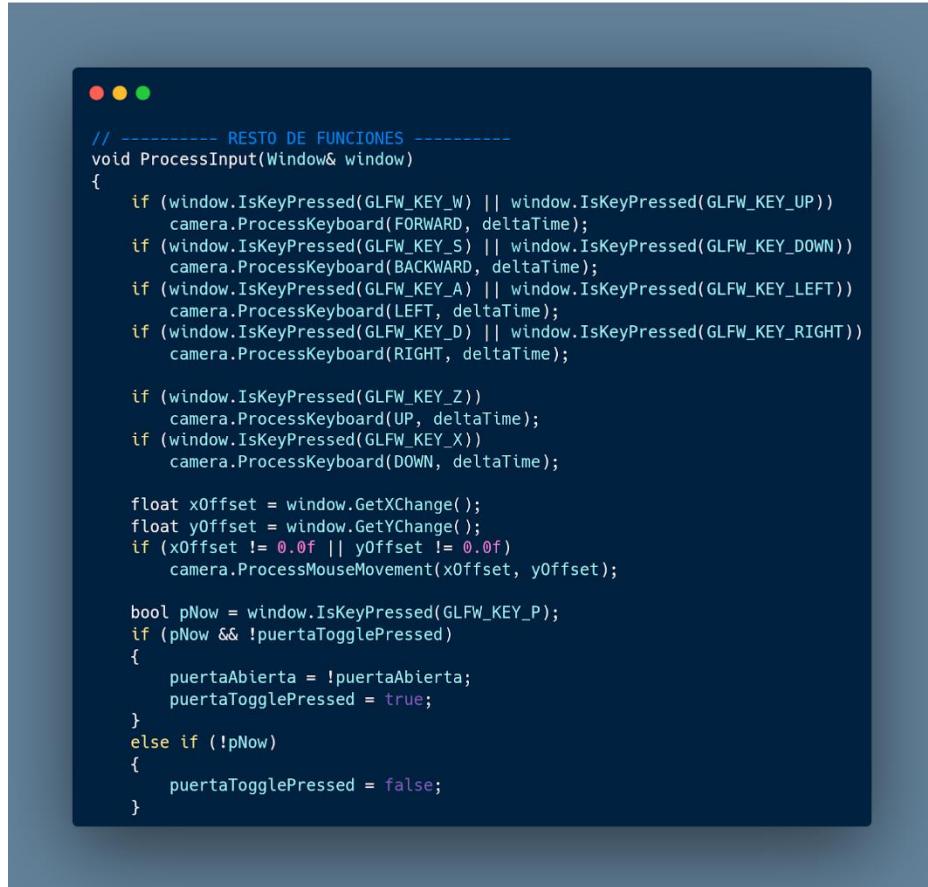
    // ----- SHADER ILUMINACIÓN -----
    lightingShader.Use();
    glUniformMatrix4fv(viewLoc, 1, GL_FALSE, glm::value_ptr(view));
    glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(projection));
    glUniform3fv(viewPosLoc, 1, glm::value_ptr(cameraGetPosition()));
}
```

Manejo de entrada: función ProcessInput

ProcessInput(Window& window) centraliza toda la lectura de teclado y ratón. Por un lado, traduce las teclas de movimiento (WASD, flechas, Z, X) a llamadas a la cámara para avanzar, retroceder, desplazarse lateralmente o moverse en vertical. También lee el movimiento del ratón para actualizar la dirección de la cámara.

Por otro lado, ProcessInput es donde se controlan todas las animaciones y luces a través de teclas específicas: la puerta se abre o cierra, se encienden y apagan las animaciones de cada hábitat (acuático, ciervos, pájaros, polar) y se pueden activar o desactivar las luces puntuales y la luz ambiental. Para evitar que una sola pulsación de tecla dispare el cambio varias veces, se usan banderas de “toggle pressed” que registran si la tecla ya fue

procesada y sólo permiten cambiar el estado cuando se detecta que la tecla se ha soltado y se ha vuelto a presionar.



```
// ----- RESTO DE FUNCIONES -----
void ProcessInput(Window& window)
{
    if (window.IsKeyPressed(GLFW_KEY_W) || window.IsKeyPressed(GLFW_KEY_UP))
        camera.ProcessKeyboard(FORWARD, deltaTime);
    if (window.IsKeyPressed(GLFW_KEY_S) || window.IsKeyPressed(GLFW_KEY_DOWN))
        camera.ProcessKeyboard(BACKWARD, deltaTime);
    if (window.IsKeyPressed(GLFW_KEY_A) || window.IsKeyPressed(GLFW_KEY_LEFT))
        camera.ProcessKeyboard(LEFT, deltaTime);
    if (window.IsKeyPressed(GLFW_KEY_D) || window.IsKeyPressed(GLFW_KEY_RIGHT))
        camera.ProcessKeyboard(RIGHT, deltaTime);

    if (window.IsKeyPressed(GLFW_KEY_Z))
        camera.ProcessKeyboard(UP, deltaTime);
    if (window.IsKeyPressed(GLFW_KEY_X))
        camera.ProcessKeyboard(DOWN, deltaTime);

    float xOffset = window.GetXChange();
    float yOffset = window.GetYChange();
    if (xOffset != 0.0f || yOffset != 0.0f)
        camera.ProcessMouseMovement(xOffset, yOffset);

    bool pNow = window.IsKeyPressed(GLFW_KEY_P);
    if (pNow && !puertaTogglePressed)
    {
        puertaAbierta = !puertaAbierta;
        puertaTogglePressed = true;
    }
    else if (!pNow)
    {
        puertaTogglePressed = false;
    }
}
```

Lógica de animación por frame: función Animation

La función Animation() actualiza, en función de deltaTime y de los flags activos, todos los parámetros que definen la pose y la trayectoria de cada elemento animado. Aquí no se dibuja nada; sólo se modifican posiciones, rotaciones y ángulos de articulaciones que luego se aplican al dibujar.

En el caso de la puerta, la función incrementa o decrementa el ángulo de apertura hasta llegar a 0° o 90°. Para el ciervo adulto, se actualizan las patas y la cabeza con un movimiento oscilatorio, y se hace que el animal camine en línea recta durante un tramo, gire 180° y regrese, repitiendo ese patrón. El ciervo bebé tiene una animación más “narrativa”: baja el cuello, hace varias inclinaciones de cabeza y luego vuelve a una pose neutra de forma suave.

Los pájaros tienen comportamientos diferentes: uno vuela describiendo una trayectoria suave en el aire con aleteo continuo, y los otros dos oscilan su cabeza, cola y cuerpo alrededor de un rango de rotación. El tucán, además, abre y cierra el pico limitado por un ángulo máximo para que no se vea exagerado.

Para el hábitat polar, la animación del oso polar, pingüino y foca combina caminar en pingpong con movimientos de patas, alas, cabeza y aletas, usando senos y fases desplazadas. Cuando se desactiva la animación de este hábitat, los tiempos y ángulos se reinician a cero para que todo vuelva a un estado neutro.

Finalmente, el tiburón y la piraña del hábitat marino nadan dentro de límites predefinidos, cambiando de dirección cuando llegan a sus fronteras. Sus colas, cuerpos y cabezas oscilan con diferentes frecuencias para dar sensación de nado. La piraña, además, rota suavemente de un lado a otro. Toda esta lógica convierte valores numéricos en “vidas” y movimientos que luego se reflejan en la escena de cada frame.



The screenshot shows a code editor window with a dark theme. The code is written in C++ and defines an `Animation()` function. The function contains logic for a door (Puerta) and two types of crows (Ciervo Adulto and Ciervo Bebe). For the door, it checks if it's open and rotates it towards 90 degrees if it's below or away from 90 degrees. For the crows, it handles leg and head movement. It uses variables like `rotPuerta`, `velocidadPuerta`, `animacionH2`, `RLegs`, `FLegs`, and `head`. It also manages a boolean variable `Ciervostep` to alternate leg movement directions.

```
void Animation()
{
    // Puerta
    if (puertaAbierta)
    {
        if (rotPuerta < 90.0f) rotPuerta += velocidadPuerta;
    }
    else
    {
        if (rotPuerta > 0.0f) rotPuerta -= velocidadPuerta;
    }

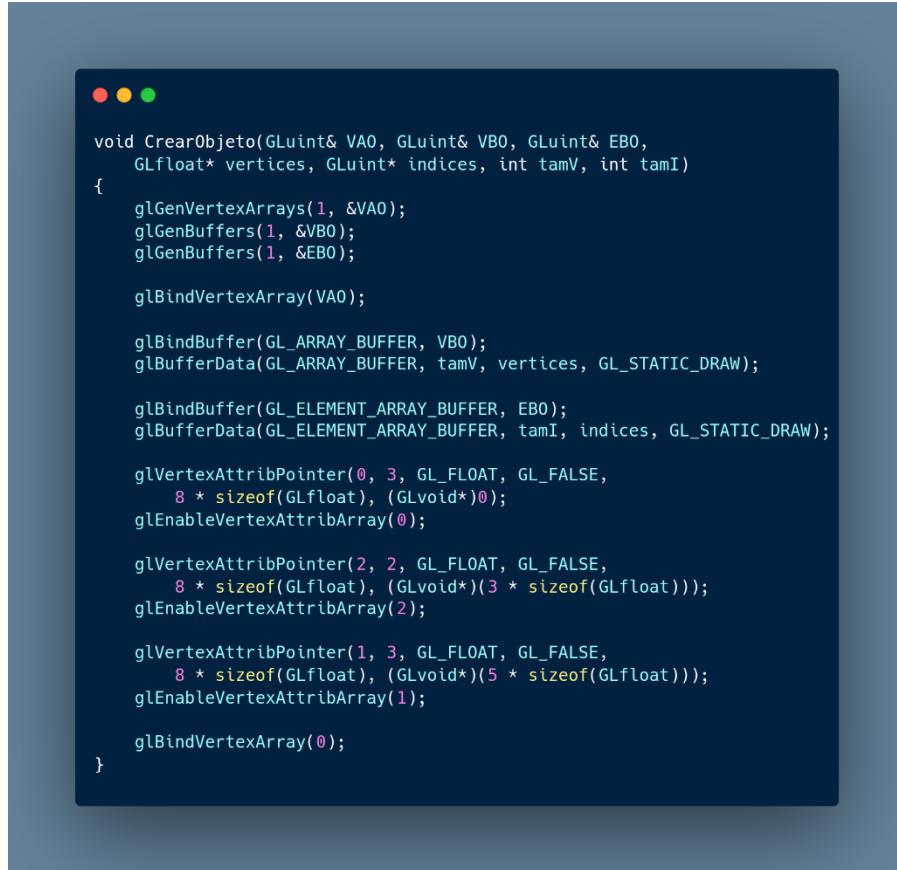
    // Ciervo Adulto (Movimiento y Patas y Ciervo Bebe
    if (animacionH2)
    {
        //Animación de patas Ciervo Adulto
        if (!Ciervostep)
        {
            RLegs += 0.3f;
            FLegs += 0.3f;
            head += 0.3f;
            if (RLegs > 15.0f) Ciervostep = true;
        }
        else {
            RLegs -= 0.3f;
            FLegs -= 0.3f;
            head -= 0.3f;
            if (RLegs < -15.0f) Ciervostep = false;
        }
    }
}
```

Funciones auxiliares de geometría: `CrearObjeto`, `CrearObjetoSkyBox` y `DibujarBarda`

Al final del archivo se encuentran las funciones auxiliares que encapsulan tareas repetitivas de OpenGL. `CrearObjeto` recibe un arreglo de vértices e índices y arma un VAO completo con su VBO y EBO, configurando los atributos de posición, coordenadas de textura y normales según el layout elegido. `CrearObjetoSkyBox` hace lo mismo pero para el caso más simple del skybox, donde sólo se necesita la posición del vértice.

`DibujarBarda` es una pequeña función de conveniencia que, a partir de la posición, rotación y largo deseado, construye la matriz de modelo para un tramo de barda, activa la textura

correspondiente y emite la llamada de dibujo. De esta forma, toda la lógica de cómo se transforma y pinta una sección de muro queda encapsulada y reusada dentro del bucle principal simplemente recorriendo el vector de WallSegment.



```
void CrearObjeto(GLuint& VAO, GLuint& VBO, GLuint& EBO,
    GLfloat* vertices, GLuint* indices, int tamV, int tamI)
{
    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);
    glGenBuffers(1, &EBO);

    glBindVertexArray(VAO);

    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, tamV, vertices, GL_STATIC_DRAW);

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, tamI, indices, GL_STATIC_DRAW);

    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
        8 * sizeof(GLfloat), (GLvoid*)0);
    glEnableVertexAttribArray(0);

    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE,
        8 * sizeof(GLfloat), (GLvoid*)(3 * sizeof(GLfloat)));
    glEnableVertexAttribArray(2);

    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE,
        8 * sizeof(GLfloat), (GLvoid*)(5 * sizeof(GLfloat)));
    glEnableVertexAttribArray(1);

    glBindVertexArray(0);
}
```

8. Conclusiones

Puedo concluir que durante el curso comprender de forma profunda cómo funciona el pipeline gráfico fue de gran ayuda, desde el manejo de buffers hasta la aplicación de matrices de transformación, iluminación y texturas, a diferencia de motores gráficos de alto nivel que veía en teoría, este enfoque brinda un dominio más preciso sobre cada etapa del renderizado.

El proyecto logró integrar de manera exitosa múltiples áreas de la computación gráfica, incluyendo modelado en Blender, carga de modelos, jerarquías, animación, iluminación y texturización. El principal reto del proyecto fue la creación de objetos completamente desde vértices en OpenGL, ya que requiere poner manualmente posiciones, normales y coordenadas de textura. Aun así se logró integrar objetos personalizados como bardas, puertas y cubos iluminados.

Las limitaciones principales fueron el tiempo y la complejidad técnica del renderizado manual, pero el resultado final integra la mayoría de prácticas vistas en el curso.