#include <iostream>

#include <vector>

#include <GL/glut.h>

#include <assimp/Importer.hpp>

#include <assimp/scene.h>

#include <assimp/postprocess.h>

// Variables globales de cámara y modelo

GLfloat cameraX = 0.0f;

GLfloat cameraY = 1.5f;

GLfloat cameraZ = 5.0f;

GLfloat cameraAngleX = 0.0f;

GLfloat cameraAngleY = 0.0f;

GLfloat weaponOffsetX = 0.0f; // Ajuste de posición del arma respecto a la cámara en X

GLfloat weaponOffsetY = -0.5f; // Ajuste de posición del arma respecto a la cámara en Y

GLfloat weaponOffsetZ = -2.0f; // Ajuste de posición del arma respecto a la cámara en Z

GLfloat rotationAngle = 180.0f; // Variable para controlar la rotación del objeto (inicializada a 180 grados)

// Estructura para almacenar datos del modelo

struct Mesh {

std::vector<GLfloat> vertices;

std::vector<GLuint> indices;

};

std::vector<Mesh> meshes;

// Función para cargar un modelo OBJ utilizando Assimp

void loadOBJ(const std::string& filePath) {

Assimp::Importer importer;

const aiScene\* scene = importer.ReadFile(filePath, aiProcess\_Triangulate | aiProcess\_FlipUVs);

if (!scene || scene->mFlags & AI\_SCENE\_FLAGS\_INCOMPLETE || !scene->mRootNode) {

std::cerr << "Error al cargar el modelo: " << importer.GetErrorString() << std::endl;

return;

}

// Recorrer todos los nodos de la escena y extraer los datos del modelo

for (unsigned int i = 0; i < scene->mNumMeshes; ++i) {

Mesh mesh;

const aiMesh\* aiMesh = scene->mMeshes[i];

for (unsigned int j = 0; j < aiMesh->mNumVertices; ++j) {

mesh.vertices.push\_back(aiMesh->mVertices[j].x);

mesh.vertices.push\_back(aiMesh->mVertices[j].y);

mesh.vertices.push\_back(aiMesh->mVertices[j].z);

}

for (unsigned int j = 0; j < aiMesh->mNumFaces; ++j) {

const aiFace& face = aiMesh->mFaces[j];

for (unsigned int k = 0; k < face.mNumIndices; ++k) {

mesh.indices.push\_back(face.mIndices[k]);

}

}

meshes.push\_back(mesh);

}

}

// Función de inicialización de OpenGL

void init() {

glEnable(GL\_DEPTH\_TEST);

glEnable(GL\_LIGHTING);

glEnable(GL\_LIGHT0);

glClearColor(0.0f, 0.0f, 0.0f, 1.0f);

// Cargar modelo OBJ

loadOBJ("m4a1\_s.obj");

}

// Función de renderizado

void render() {

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

// Configurar la cámara

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

gluPerspective(45.0f, 1.0f, 0.1f, 100.0f);

// Ajustar la vista de la cámara para la perspectiva en primera persona

glMatrixMode(GL\_MODELVIEW);

glLoadIdentity();

gluLookAt(cameraX, cameraY, cameraZ, // Posición de la cámara

cameraX + sin(cameraAngleY), cameraY + sin(cameraAngleX), cameraZ - cos(cameraAngleY), // Punto hacia donde mira la cámara

0.0, 1.0, 0.0); // Vector de arriba

// Posicionar y rotar el modelo de la M4 (arma)

glPushMatrix();

glTranslatef(cameraX + weaponOffsetX, cameraY + weaponOffsetY, cameraZ + weaponOffsetZ); // Ajustar la posición relativa al jugador

glRotatef(rotationAngle, 0.0, 1.0, 0.0); // Ajustar la rotación inicial (180 grados alrededor del eje Y)

// Renderizar cada malla del modelo

for (const auto& mesh : meshes) {

glEnableClientState(GL\_VERTEX\_ARRAY);

glVertexPointer(3, GL\_FLOAT, 0, mesh.vertices.data());

glDrawElements(GL\_TRIANGLES, mesh.indices.size(), GL\_UNSIGNED\_INT, mesh.indices.data());

glDisableClientState(GL\_VERTEX\_ARRAY);

}

glPopMatrix(); // Restaurar la matriz de modelo-vista

glutSwapBuffers();

}

// Función de manejo de movimiento del ratón

void mouseMotion(int x, int y) {

static int lastX = x;

static int lastY = y;

// Calcula la diferencia de posición del ratón

int deltaX = x - lastX;

int deltaY = y - lastY;

// Ajusta el factor de sensibilidad para controlar la velocidad de rotación

const float sensitivity = 0.2f; // Ajusta este valor para controlar la velocidad de rotación

// Calcula el cambio en los ángulos de la cámara basado en la sensibilidad

cameraAngleY += deltaX \* sensitivity;

cameraAngleX += deltaY \* sensitivity;

// Limita el ángulo vertical de la cámara

if (cameraAngleX > 89.0f) cameraAngleX = 89.0f;

if (cameraAngleX < -89.0f) cameraAngleX = -89.0f;

// Calcula la dirección hacia donde mira la cámara

float radianAngleY = cameraAngleY \* 3.14159265 / 180.0f;

float radianAngleX = cameraAngleX \* 3.14159265 / 180.0f;

cameraZ = cameraY + 5.0f \* cos(radianAngleX);

cameraX = cameraY + 5.0f \* sin(radianAngleX) \* cos(radianAngleY);

// Actualiza las posiciones del último ratón

lastX = x;

lastY = y;

// Solicita que se vuelva a renderizar la escena

glutPostRedisplay();

}

// Función de manejo de eventos del ratón para rotar la cámara al hacer clic y arrastrar

void mouse(int button, int state, int x, int y) {

if (button == GLUT\_LEFT\_BUTTON && state == GLUT\_DOWN) {

// Guarda la posición inicial del clic

glutMotionFunc(mouseMotion);

}

else if (button == GLUT\_LEFT\_BUTTON && state == GLUT\_UP) {

// Detiene la rotación cuando se libera el clic

glutMotionFunc(nullptr);

}

}

int main(int argc, char\*\* argv) {

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGBA | GLUT\_DEPTH);

glutInitWindowSize(600, 600);

glutCreateWindow("OpenGL + Assimp + GLUT OBJ Loader");

init();

glutDisplayFunc(render);

// Registra funciones de manejo de eventos del ratón

glutMouseFunc(mouse);

glutMotionFunc(nullptr);

glutMainLoop();

return 0;

}