Quiero que me ayudes en algo dentro de mi proyecto, sucede que cuando agrego el bossfight con la letra K dentro de mi map3.txt no aparece en mi juego, el compila, pero no funciona es decir mi Bossfight nunca aparece, en los logs que tengo creados este tampoco aparece, también tienes que tener en cuenta de que mis map están todos bien escritos, como podría solucionar lo de la K y el bossfight, recuerda que mi map3.txt carga completo solo mi K no aparece, mis mapas están bien escrito, este es mi codigo

Bullet.cpp:

#include "bullet.h"

#include "enemy.h"

#include "collision.h"

#include "map.h"

#include <GL/freeglut.h>

#include <vector>

#include <algorithm>

#include <cmath>

#include "globals.h"

#include <iostream>

// Lista de balas activas

std::vector<Bullet> bullets;

extern std::vector<Enemy> enemies;

extern int score;

static void onBossDefeated(float x, float y) {

bossAlive = false;

bossExploding = true;

bossExplodeTimer = 90; // ~1.5s

bossDeathX = x;

bossDeathY = y;

// Portal se coloca al terminar la explosión (ver Main.timer)

}

void shootBullet(float startX, float startY, float targetX, float targetY) {

Bullet b;

b.x = startX;

b.y = startY;

float dx = targetX - startX;

float dy = targetY - startY;

float length = sqrt(dx \* dx + dy \* dy);

if (length != 0) {

b.vx = (dx / length) \* b.speed;

b.vy = (dy / length) \* b.speed;

} else {

b.vx = 0;

b.vy = b.speed;

}

bullets.push\_back(b);

}

void updateBullets() {

// Mover y revisar balas una por una

for (auto it = bullets.begin(); it != bullets.end(); ) {

it->x += it->vx;

it->y += it->vy;

// 1?? Verificar colisión con enemigos

bool hit = false;

for (auto enemyIt = enemies.begin(); enemyIt != enemies.end(); ) {

// dentro de updateBullets() donde recorres enemigos:

if (bulletHitsEnemy(it->x, it->y, enemyIt->x, enemyIt->y, enemyIt->size)) {

enemyIt->health--;

if (enemyIt->health <= 0) {

// === NUEVO: si era boss, disparar secuencia de final

if (enemyIt->type == ENEMY\_BOSS) {

onBossDefeated(enemyIt->x, enemyIt->y);

}

enemyIt = enemies.erase(enemyIt);

score += 1000; // recompensa mayor si gustas

} else {

++enemyIt;

}

hit = true;

break;

}

}

// 2?? Verificar colisión con pared

if (isWall(it->x, it->y)) {

hit = true;

}

// 3?? Eliminar la bala si impactó o salió del mundo

if (hit || it->y > 2000 || it->y < -2000 || it->x > 2000 || it->x < -2000) {

it = bullets.erase(it);

} else {

++it;

}

}

}

void drawBullets() {

glColor3f(1.0f, 1.0f, 0.0f); // Amarillo

for (auto& b : bullets) {

glRectf(b.x - 2, b.y - 2, b.x + 2, b.y + 2);

}

}

Bullet.h:

#ifndef BULLET\_H

#define BULLET\_H

#include <vector>

struct Bullet {

float x, y;

float vx, vy;

float speed = 15.0f;

};

void shootBullet(float startX, float startY, float targetX, float targetY);

void updateBullets();

void drawBullets();

extern std::vector<Bullet> bullets;

#endif

Collision.cpp:

#include "collision.h"

#include "map.h"

extern std::vector<std::string> mapData;

bool isWall(float worldX, float worldY) {

int tileSize = 40;

int mapHeight = mapData.size();

int mapWidth = mapData[0].size();

int col = static\_cast<int>((worldX / tileSize) + mapWidth / 2.0f);

int row = static\_cast<int>((mapHeight / 2.0f) - (worldY / tileSize));

if (row >= 0 && row < mapHeight && col >= 0 && col < mapWidth) {

return mapData[row][col] == '#';

}

return false;

}

// Verifica las 4 esquinas del jugador como un cuadrado de 30x30

bool canMoveTo(float centerX, float centerY) {

float halfSize = 15.0f; // Radio del jugador

return

!isWall(centerX - halfSize, centerY - halfSize) &&

!isWall(centerX + halfSize, centerY - halfSize) &&

!isWall(centerX - halfSize, centerY + halfSize) &&

!isWall(centerX + halfSize, centerY + halfSize);

}

// Collision.cpp

bool bulletHitsEnemy(float bulletX, float bulletY, float enemyX, float enemyY, float enemySize) {

float bulletHalf = 2.0f;

// enemySize es radio (mitad del ancho/alto)

return

bulletX + bulletHalf > enemyX - enemySize &&

bulletX - bulletHalf < enemyX + enemySize &&

bulletY + bulletHalf > enemyY - enemySize &&

bulletY - bulletHalf < enemyY + enemySize;

}

bool isOnPortal(float worldX, float worldY) {

int tileSize = 40;

int mapHeight = mapData.size();

int mapWidth = mapData[0].size();

int col = static\_cast<int>((worldX / tileSize) + mapWidth / 2.0f);

int row = static\_cast<int>((mapHeight / 2.0f) - (worldY / tileSize));

if (row >= 0 && row < mapHeight && col >= 0 && col < mapWidth) {

return mapData[row][col] == 'P';

}

return false;

}

Collision.h:

#ifndef COLLISION\_H

#define COLLISION\_H

// Verifica si hay una pared en esa posición

bool isWall(float worldX, float worldY);

// Verifica si el jugador puede moverse a la posición (considerando tamaño)

bool canMoveTo(float centerX, float centerY);

bool bulletHitsEnemy(float bulletX, float bulletY, float enemyX, float enemyY);

bool isOnPortal(float worldX, float worldY);

// Collision.h

bool bulletHitsEnemy(float bulletX, float bulletY, float enemyX, float enemyY, float enemySize);

#endif

Draw.cpp:

#include <GL/freeglut.h>

#include "globals.h"

#include <string>

extern int nivelActual;

extern bool cambioNivelPendiente;

void drawTutorialText() {

if (tutorialPaso >= 3) return;

const char\* mensaje = NULL;

if (tutorialPaso == 1)

mensaje = "Presiona WASD para moverte";

else if (tutorialPaso == 2)

mensaje = "Usa clic para disparar";

if (!mensaje) return;

// Proyección 2D

glMatrixMode(GL\_PROJECTION);

glPushMatrix();

glLoadIdentity();

gluOrtho2D(0, currentWindowWidth, 0, currentWindowHeight);

glMatrixMode(GL\_MODELVIEW);

glPushMatrix();

glLoadIdentity();

glEnable(GL\_BLEND);

glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA);

glColor4f(1.0f, 1.0f, 1.0f, tutorialAlpha); // Usar alpha

// Calcular ancho real del texto (en píxeles) usando ancho de fuente

int textWidth = 0;

for (int i = 0; mensaje[i] != '\0'; ++i)

textWidth += glutBitmapWidth(GLUT\_BITMAP\_HELVETICA\_18, mensaje[i]);

// Posicionar el texto centrado horizontalmente

glRasterPos2i((currentWindowWidth - textWidth) / 2, currentWindowHeight / 2 + 200);

for (int i = 0; mensaje[i] != '\0'; ++i)

glutBitmapCharacter(GLUT\_BITMAP\_HELVETICA\_18, mensaje[i]);

glDisable(GL\_BLEND);

glPopMatrix();

glMatrixMode(GL\_PROJECTION);

glPopMatrix();

glMatrixMode(GL\_MODELVIEW);

}

void drawGameOverScreen() {

if (!gameOverActivo) return;

// Fade-in

if (gameOverAlpha < 1.0f)

gameOverAlpha += 0.01f;

// Proyección 2D

glMatrixMode(GL\_PROJECTION);

glPushMatrix();

glLoadIdentity();

gluOrtho2D(0, currentWindowWidth, 0, currentWindowHeight);

glMatrixMode(GL\_MODELVIEW);

glPushMatrix();

glLoadIdentity();

glEnable(GL\_BLEND);

glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA);

// Fondo negro con transparencia

glColor4f(0, 0, 0, gameOverAlpha \* 0.7f);

glBegin(GL\_QUADS);

glVertex2i(0, 0);

glVertex2i(currentWindowWidth, 0);

glVertex2i(currentWindowWidth, currentWindowHeight);

glVertex2i(0, currentWindowHeight);

glEnd();

// Texto "GAME OVER"

const char\* texto = "GAME OVER";

int ancho = 0;

for (int i = 0; texto[i]; ++i)

ancho += glutBitmapWidth(GLUT\_BITMAP\_HELVETICA\_18, texto[i]);

glColor4f(1, 1, 1, gameOverAlpha);

glRasterPos2i((currentWindowWidth - ancho) / 2, currentWindowHeight / 2 + 80);

for (int i = 0; texto[i]; ++i)

glutBitmapCharacter(GLUT\_BITMAP\_HELVETICA\_18, texto[i]);

// Opciones

const char\* opciones[] = { "Reintentar", "Salir" };

for (int i = 0; i < 2; ++i) {

if (i == opcionGameOver)

glColor4f(1.0f, 1.0f, 0.0f, gameOverAlpha); // Amarillo

else

glColor4f(1.0f, 1.0f, 1.0f, gameOverAlpha);

int anchoOp = 0;

for (int j = 0; opciones[i][j]; ++j)

anchoOp += glutBitmapWidth(GLUT\_BITMAP\_HELVETICA\_18, opciones[i][j]);

glRasterPos2i((currentWindowWidth - anchoOp) / 2, currentWindowHeight / 2 - i \* 40);

for (int j = 0; opciones[i][j]; ++j)

glutBitmapCharacter(GLUT\_BITMAP\_HELVETICA\_18, opciones[i][j]);

}

glDisable(GL\_BLEND);

glPopMatrix();

glMatrixMode(GL\_PROJECTION);

glPopMatrix();

glMatrixMode(GL\_MODELVIEW);

}

void drawLoadingLevelText() {

if (!cambioNivelPendiente) return;

std::string texto = "1-" + std::to\_string(nivelActual);

glMatrixMode(GL\_PROJECTION);

glPushMatrix();

glLoadIdentity();

gluOrtho2D(0, currentWindowWidth, 0, currentWindowHeight);

glMatrixMode(GL\_MODELVIEW);

glPushMatrix();

glLoadIdentity();

glColor4f(1, 1, 1, 1);

int ancho = 0;

for (char c : texto)

ancho += glutBitmapWidth(GLUT\_BITMAP\_HELVETICA\_18, c);

glRasterPos2i((currentWindowWidth - ancho) / 2, currentWindowHeight / 2);

for (char c : texto)

glutBitmapCharacter(GLUT\_BITMAP\_HELVETICA\_18, c);

glPopMatrix();

glMatrixMode(GL\_PROJECTION);

glPopMatrix();

glMatrixMode(GL\_MODELVIEW);

}

// Draw.cpp (al final, similar a las otras pantallas)

void drawCreditsScreen() {

// Fondo 2D

glMatrixMode(GL\_PROJECTION);

glPushMatrix(); glLoadIdentity(); gluOrtho2D(0, currentWindowWidth, 0, currentWindowHeight);

glMatrixMode(GL\_MODELVIEW);

glPushMatrix(); glLoadIdentity();

glEnable(GL\_BLEND);

glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA);

// fondo

glColor4f(0, 0, 0, 0.85f);

glBegin(GL\_QUADS);

glVertex2i(0,0); glVertex2i(currentWindowWidth,0);

glVertex2i(currentWindowWidth,currentWindowHeight); glVertex2i(0,currentWindowHeight);

glEnd();

// título

const char\* titulo = "CREDITOS";

int w=0; for (const char\* p=titulo; \*p; ++p) w += glutBitmapWidth(GLUT\_BITMAP\_HELVETICA\_18, \*p);

glColor3f(1,1,1);

glRasterPos2i((currentWindowWidth - w)/2, currentWindowHeight - 80);

for (const char\* p=titulo; \*p; ++p) glutBitmapCharacter(GLUT\_BITMAP\_HELVETICA\_18, \*p);

// lista (pon lo tuyo)

const char\* lines[] = {

"Programacion: Tu Nombre",

"Arte y Sprites: Tu Nombre",

"Musica: (SDL\_mixer)",

"Motor: FreeGLUT + OpenGL",

"Agradecimientos: Familia & Amigos",

"",

"Presiona ENTER o ESC para volver al Menu"

};

int y = currentWindowHeight - 140;

for (int i=0;i<7;++i) {

int ww=0; for (const char\* p=lines[i]; \*p; ++p) ww += glutBitmapWidth(GLUT\_BITMAP\_HELVETICA\_18, \*p);

glRasterPos2i((currentWindowWidth - ww)/2, y - i\*30);

for (const char\* p=lines[i]; \*p; ++p) glutBitmapCharacter(GLUT\_BITMAP\_HELVETICA\_18, \*p);

}

glDisable(GL\_BLEND);

glPopMatrix();

glMatrixMode(GL\_PROJECTION); glPopMatrix();

glMatrixMode(GL\_MODELVIEW);

}

Draw.h:

#ifndef DRAW\_H

#define DRAW\_H

void drawTutorialText();

void drawGameOverScreen();

void drawLoadingLevelText();

// Draw.h

void drawCreditsScreen();

#endif

Enemy.cpp:

#include "enemy.h"

#include "collision.h"

#include "globals.h"

#include <GL/freeglut.h>

#include <cmath>

#include <iostream>

#include <algorithm>

#include "enemy\_bullet.h"

#include "map.h" // arriba de Enemy.cpp

std::vector<Enemy> enemies;

void addEnemy(float x, float y, EnemyType type) {

Enemy e;

e.x = x;

e.y = y;

e.type = type; // Usar 'type' no 'tipo'

// Configuración basada en tipo

switch(type) {

case ENEMY\_MELEE:

e.speed = 1.5f;

e.health = 2;

break;

case ENEMY\_RANGED:

e.speed = 0.8f;

e.health = 1;

break;

case ENEMY\_BASIC:

e.speed = 1.0f;

e.health = 3;

break;

case ENEMY\_SNIPER:

e.speed = 0.6f;

e.health = 2;

break;

case ENEMY\_FAST:

e.speed = 2.5f;

e.health = 1;

break;

// en addEnemy(...)

case ENEMY\_BOSS:

e.speed = 0.7f;

e.health = 40; // vida del boss

e.maxHealth = 40;

e.size = 20.0f; // radio -> sprite 40x40 aprox

e.phase = 0;

e.phaseTimer = 0;

e.spawnCooldown2 = 180;

break;

default:

std::cerr << "Tipo de enemigo desconocido, usando valores por defecto\n";

e.type = ENEMY\_MELEE;

e.speed = 1.5f;

e.health = 2;

}

e.maxHealth = e.health;

e.activo = false;

enemies.push\_back(e);

}

bool addEnemyFromMapChar(char symbol, float x, float y) {

unsigned char us = static\_cast<unsigned char>(symbol);

char s = static\_cast<char>(std::toupper(us)); // normaliza a MAYÚSCULA

switch (s) {

case 'E': addEnemy(x,y,ENEMY\_MELEE); return true;

case 'B': addEnemy(x,y,ENEMY\_BASIC); return true;

case 'S': addEnemy(x,y,ENEMY\_SNIPER); return true;

case 'F': addEnemy(x,y,ENEMY\_FAST); return true;

case 'K':

std::cout << "[DEBUG] Boss colocado en (" << x << "," << y << ")\n";

addEnemy(x,y,ENEMY\_BOSS);

bossAlive = true;

return true;

default:

return false;

}

}

void updateEnemies(float playerX, float playerY) {

const float visionRange = 300.0f;

bool playerDamagedThisFrame = false;

for (auto& e : enemies) {

if (e.health <= 0 || e.type >= ENEMY\_INVALID) continue;

float dx = playerX - e.x;

float dy = playerY - e.y;

float distance = sqrt(dx\*dx + dy\*dy);

// Debug de activación

if (!e.activo && distance < visionRange) {

std::cout << "Activando enemigo en (" << e.x << "," << e.y

<< ") - Tipo: " << e.type << std::endl;

e.activo = true;

}

// Activación por proximidad

if (!e.activo && distance < visionRange) {

e.activo = true;

// Asegurar valores correctos al activarse

e.health = std::max(1, e.health);

if (e.type == ENEMY\_MELEE) {

e.speed = 1.5f;

} else {

e.speed = 0.8f;

}

}

// Movimiento solo para melee activos

if (e.activo && e.type == ENEMY\_MELEE) {

if (distance > 20.0f) {

float dirX = dx/distance;

float dirY = dy/distance;

// Movimiento con colisión

float nextX = e.x + dirX \* e.speed;

if (!isWall(nextX, e.y)) e.x = nextX;

float nextY = e.y + dirY \* e.speed;

if (!isWall(e.x, nextY)) e.y = nextY;

}

}

else if (e.activo && e.type == ENEMY\_SNIPER) {

float dx = playerX - e.x;

float dy = playerY - e.y;

float distance = sqrt(dx\*dx + dy\*dy);

// Dispara cada cierto tiempo si ve al jugador

if (distance < 500.0f) {

if (e.shootCooldown <= 0) {

shootEnemyBullet(e.x, e.y, playerX, playerY);

e.shootCooldown = 120; // 2 segundos entre disparos (60 FPS)

} else {

e.shootCooldown--;

}

}

}

else if (e.activo && e.type == ENEMY\_FAST) {

if (distance > 20.0f) {

float dirX = dx / distance;

float dirY = dy / distance;

float nextX = e.x + dirX \* e.speed;

if (!isWall(nextX, e.y)) e.x = nextX;

float nextY = e.y + dirY \* e.speed;

if (!isWall(e.x, nextY)) e.y = nextY;

}

}

// dentro de updateEnemies(...)

else if (e.activo && e.type == ENEMY\_BOSS) {

// 1) El boss se mueve lento hacia/alejado del jugador con un vaivén

float wiggle = sin(e.phaseTimer \* 0.05f) \* 0.8f;

float dirX = dx / std::max(1.0f, distance);

float dirY = dy / std::max(1.0f, distance);

// Mantener cierta distancia media (kiteo simple)

float desired = 220.0f;

float push = (distance > desired ? 1.0f : -1.0f);

float nextX = e.x + (dirX \* e.speed \* push) + wiggle;

float nextY = e.y + (dirY \* e.speed \* push) - wiggle;

if (!isWall(nextX, e.y)) e.x = nextX;

if (!isWall(e.x, nextY)) e.y = nextY;

// 2) Ciclo de fases (cada ~6s cambia)

e.phaseTimer++;

if (e.phaseTimer > 360) {

e.phaseTimer = 0;

e.phase = (e.phase + 1) % 3; // 0: minions, 1: radial, 2: sniper burst

}

// 3) Acciones por fase

if (e.phase == 0) {

// Spawn de minions rápidos cada 3s

if (e.spawnCooldown2 <= 0) {

// genera 2-3 FAST alrededor

addEnemy(e.x + 60, e.y, ENEMY\_FAST);

addEnemy(e.x - 60, e.y, ENEMY\_FAST);

if (rand() % 2) addEnemy(e.x, e.y + 80, ENEMY\_FAST);

e.spawnCooldown2 = 180;

} else e.spawnCooldown2--;

}

else if (e.phase == 1) {

// Ráfaga radial de balas cada 45 frames

if (e.shootCooldown <= 0) {

const int N = 12;

for (int i=0;i<N;++i) {

float ang = (2.0f \* 3.1415926f \* i) / N;

float tx = e.x + cos(ang) \* 200.0f;

float ty = e.y + sin(ang) \* 200.0f;

shootEnemyBullet(e.x, e.y, tx, ty);

}

e.shootCooldown = 45;

} else e.shootCooldown--;

}

else if (e.phase == 2) {

// Disparos dirigidos en ráfagas de 3

if (e.shootCooldown <= 0) {

for (int k=0;k<3;++k) {

shootEnemyBullet(e.x, e.y, playerX, playerY);

}

e.shootCooldown = 60;

} else e.shootCooldown--;

}

}

// Daño al jugador

if (e.activo && distance < 20.0f && invulnerableTimer == 0 && !playerDamagedThisFrame) {

playerHealth--;

invulnerableTimer = 60;

playerDamagedThisFrame = true;

// Empujar al enemigo

e.x += (e.x < playerX ? -40 : 40);

e.y += (e.y < playerY ? -40 : 40);

}

}

}

void drawEnemies() {

static int frameCounter = 0;

frameCounter++;

for (auto& e : enemies) {

// Validación general

if (e.health <= 0 || std::isnan(e.x) || std::isnan(e.y)) continue;

// Color por tipo de enemigo

switch (e.type) {

case ENEMY\_MELEE: glColor3f(1.0f, 0.0f, 0.0f); break;

case ENEMY\_BASIC: glColor3f(1.0f, 1.0f, 0.0f); break;

case ENEMY\_SNIPER: glColor3f(0.3f, 0.3f, 1.0f); break;

case ENEMY\_FAST: glColor3f(0.0f, 1.0f, 0.0f); break;

case ENEMY\_BOSS: glColor3f(0.9f, 0.3f, 0.9f); break; // morado

default: glColor3f(1.0f, 1.0f, 1.0f); break;

}

// Cuerpo del enemigo (cuadrado de 20x20)

float r = e.size;

glBegin(GL\_QUADS);

glVertex2f(e.x - r, e.y - r);

glVertex2f(e.x + r, e.y - r);

glVertex2f(e.x + r, e.y + r);

glVertex2f(e.x - r, e.y + r);

glEnd();

// Barra de vida (visible siempre que tenga vida)

if (e.health > 0) {

float barWidth = r \* 2.0f;

float barHeight = 4.0f;

float xLeft = e.x - barWidth / 2.0f;

float yTop = e.y + r + 6.0f;

// fondo

glColor3f(0.2f, 0.0f, 0.0f);

glBegin(GL\_QUADS);

glVertex2f(xLeft, yTop);

glVertex2f(xLeft + barWidth, yTop);

glVertex2f(xLeft + barWidth, yTop - barHeight);

glVertex2f(xLeft, yTop - barHeight);

glEnd();

// actual

float healthRatio = (float)e.health / e.maxHealth;

glColor3f(0.0f, 1.0f, 0.0f);

glBegin(GL\_QUADS);

glVertex2f(xLeft, yTop);

glVertex2f(xLeft + barWidth \* healthRatio, yTop);

glVertex2f(xLeft + barWidth \* healthRatio, yTop - barHeight);

glVertex2f(xLeft, yTop - barHeight);

glEnd();

}

}

}

void cleanupInvalidEnemies() {

const int tile = 40;

const int H = (int)mapData.size();

const int W = H ? (int)mapData[0].size() : 0;

const float maxX = (W/2.0f + 2) \* tile; // margen

const float maxY = (H/2.0f + 2) \* tile;

enemies.erase(std::remove\_if(enemies.begin(), enemies.end(),

[&](const Enemy& e) {

return std::isnan(e.x) || std::isnan(e.y) ||

e.x < -maxX || e.x > maxX ||

e.y < -maxY || e.y > maxY ||

e.health <= 0 || e.health > 100 ||

e.type >= ENEMY\_INVALID;

}),

enemies.end());

}

Enemy.h:

#ifndef ENEMY\_H

#define ENEMY\_H

#include <vector>

enum EnemyType {

ENEMY\_MELEE = 0,

ENEMY\_RANGED = 1,

ENEMY\_BASIC = 2,

ENEMY\_SNIPER = 3,

ENEMY\_FAST = 4,

ENEMY\_BOSS = 5,

ENEMY\_INVALID = 99

};

struct Enemy {

float x = 0.0f;

float y = 0.0f;

float speed = 1.5f;

bool activo = false;

int health = 2;

int maxHealth = 2;

EnemyType type = ENEMY\_MELEE;

int shootCooldown = 0;

int damageTimer = 0;

// === NUEVO (para BOSS y afinados)

float size = 10.0f; // colisión/dibujo

int phase = 0; // patrón actual del boss

int phaseTimer = 0; // cronómetro del patrón

int spawnCooldown2 = 0;// cooldown para generar minions

};

extern std::vector<Enemy> enemies;

void updateEnemies(float playerX, float playerY);

void drawEnemies();

void cleanupInvalidEnemies();

void addEnemy(float x, float y, EnemyType type);

bool addEnemyFromMapChar(char symbol, float x, float y);

#endif

Enemy\_bullet.cpp:

#include "enemy\_bullet.h"

#include "globals.h"

#include "collision.h"

#include <cmath>

#include <GL/freeglut.h>

std::vector<EnemyBullet> enemyBullets;

void shootEnemyBullet(float startX, float startY, float targetX, float targetY) {

EnemyBullet b;

b.x = startX;

b.y = startY;

float dx = targetX - startX;

float dy = targetY - startY;

float length = std::sqrt(dx \* dx + dy \* dy);

if (length != 0) {

b.vx = (dx / length) \* b.speed;

b.vy = (dy / length) \* b.speed;

} else {

b.vx = 0;

b.vy = b.speed;

}

enemyBullets.push\_back(b);

}

void updateEnemyBullets() {

for (auto it = enemyBullets.begin(); it != enemyBullets.end(); ) {

it->x += it->vx;

it->y += it->vy;

// Colisión con jugador

float dx = it->x - cameraX;

float dy = it->y - cameraY;

float dist = sqrt(dx \* dx + dy \* dy);

if (dist < 15.0f && invulnerableTimer == 0) {

playerHealth--;

invulnerableTimer = 60;

it = enemyBullets.erase(it);

continue;

}

// Colisión con pared o fuera de límites

if (isWall(it->x, it->y) || std::abs(it->x) > 2000 || std::abs(it->y) > 2000) {

it = enemyBullets.erase(it);

} else {

++it;

}

}

}

void drawEnemyBullets() {

glColor3f(1.0f, 0.3f, 0.3f); // rojizo

for (auto& b : enemyBullets) {

glRectf(b.x - 3, b.y - 3, b.x + 3, b.y + 3);

}

}

Enemy\_bullet.h:

#ifndef ENEMY\_BULLET\_H

#define ENEMY\_BULLET\_H

#include <vector>

struct EnemyBullet {

float x, y;

float vx, vy;

float speed = 7.0f;

};

extern std::vector<EnemyBullet> enemyBullets;

void shootEnemyBullet(float startX, float startY, float targetX, float targetY);

void updateEnemyBullets();

void drawEnemyBullets();

#endif

Globals.h:

// globals.h

#ifndef GLOBALS\_H

#define GLOBALS\_H

extern int playerHealth; // ?? Declaración externa

extern float cameraX;

extern float cameraY;

extern int score;

extern int invulnerableTimer; // En frames (~60 por segundo)

extern float playerAngle;

extern int currentWindowWidth;

extern int currentWindowHeight;

extern int recoilTimer;

extern bool enPausa;

extern int opcionSeleccionada;

extern const float POS\_INICIAL\_X;

extern const float POS\_INICIAL\_Y;

extern float fadeAlpha;

extern bool enFade;

enum FadeEstado { NINGUNO, FADE\_OUT, FADE\_IN };

extern FadeEstado estadoFade;

extern int tutorialPaso; // 1 = primer texto, 2 = segundo texto, 3 = terminado

extern float tutorialAlpha; // transparencia del texto (0.0 a 1.0)

extern int tutorialEstado; // 0 = fade-in, 1 = espera, 2 = fade-out

extern int tutorialTimer; // cuenta frames para la espera

extern bool gameOverActivo;

extern float gameOverAlpha;

extern int opcionGameOver; // 0 = Reintentar, 1 = Salir

extern int gameOverTimer;

extern bool reinicioPendiente;

extern bool isFullScreen;

extern int windowedWidth;

extern int windowedHeight;

extern int windowPosX;

extern int windowPosY;

extern int nivelActual; // 1, 2, 3, etc.

extern const int TOTAL\_NIVELES; // Cantidad total de niveles

extern bool cambioNivelPendiente;

enum GameState { STATE\_MENU = 0, STATE\_SETTINGS = 1, STATE\_GAME = 2 };

extern GameState gameState;

extern int mainMenuIndex; // 0..2 (Iniciar, Ajustes, Salir)

extern int settingsMenuIndex; // 0..1 (Volumen, Volver)

extern int masterVolumePercent; // 0..100

extern bool startGamePending; // para hacer fade del menú -> juego

// globals.h (agrega estas extern nuevas)

extern bool bossAlive; // ¿existe boss en este nivel?

extern bool bossExploding; // ¿mostrando animación de explosión?

extern int bossExplodeTimer; // frames de explosión

extern float bossDeathX, bossDeathY; // dónde murió

// estado de créditos

extern bool creditsActive;

extern int creditsTimer;

#endif

Input.cpp:

// input.cpp

#include <GL/freeglut.h>

#include "input.h"

#include "globals.h"

#include "bullet.h"

#include "sound.h"

#include "collision.h"

#include <cmath>

#include <cstdlib>

#include <algorithm> // <-- necesario para std::max / std::min

#include "enemy.h"

#include <SDL.h>

#include <SDL\_mixer.h>

// Está definida en Main.cpp

extern bool keys[256];

extern int mouseX;

extern int mouseY;

// ---------------------- MENÚ PRINCIPAL ----------------------

void mainMenuKeys(unsigned char key) {

if (key == 'w') {

mainMenuIndex = (mainMenuIndex + 3 - 1) % 3; // 0..2

} else if (key == 's') {

mainMenuIndex = (mainMenuIndex + 1) % 3;

} else if (key == 13) { // ENTER

if (mainMenuIndex == 0) { // Iniciar juego

startGamePending = true;

enFade = true;

estadoFade = FADE\_OUT;

fadeAlpha = 0.0f;

} else if (mainMenuIndex == 1) { // Ajustes

gameState = STATE\_SETTINGS;

} else if (mainMenuIndex == 2) { // Salir

exit(0);

}

} else if (key == 27) { // ESC en menú: salir

exit(0);

}

}

// ---------------------- AJUSTES ----------------------

void settingsMenuKeys(unsigned char key) {

if (key == 'w') {

settingsMenuIndex = (settingsMenuIndex + 2 - 1) % 2; // 0..1

} else if (key == 's') {

settingsMenuIndex = (settingsMenuIndex + 1) % 2;

} else if (key == 'a' && settingsMenuIndex == 0) {

masterVolumePercent = std::max(0, masterVolumePercent - 5);

setMasterVolume(masterVolumePercent);

} else if (key == 'd' && settingsMenuIndex == 0) {

masterVolumePercent = std::min(100, masterVolumePercent + 5);

setMasterVolume(masterVolumePercent);

} else if (key == 13) { // ENTER

if (settingsMenuIndex == 1) { // Volver

gameState = STATE\_MENU;

}

} else if (key == 27) { // ESC = volver

gameState = STATE\_MENU;

}

}

void procesarTecla(unsigned char key, int x, int y) {

// --- Créditos: solo se manejan aquí (Enter/Esc normales) ---

if (creditsActive) {

if (key == 13 || key == 27) { // ENTER o ESC

creditsActive = false;

gameState = STATE\_MENU;

enFade = true;

estadoFade = FADE\_OUT;

fadeAlpha = 0.0f; // transición suave

playMenuMusic();

}

return;

}

// ----- Ruteo por estado global -----

if (gameState == STATE\_MENU) {

mainMenuKeys(key);

return;

} else if (gameState == STATE\_SETTINGS) {

settingsMenuKeys(key);

return;

}

// ===== De aquí en adelante: STATE\_GAME =====

if (gameOverActivo) {

if (key == 'w') {

opcionGameOver = (opcionGameOver - 1 + 2) % 2;

} else if (key == 's') {

opcionGameOver = (opcionGameOver + 1) % 2;

} else if (key == 13) { // Enter

if (opcionGameOver == 0) { // Reintentar

reinicioPendiente = true;

fadeAlpha = 0.0f;

enFade = true;

estadoFade = FADE\_OUT;

} else {

exit(0);

}

}

return;

}

if (enPausa) {

// Menú de pausa

pauseMenuKeys(key);

return;

}

keys[key] = true;

if (key == 32) { // SPACE: disparo

float worldMouseX = (mouseX - currentWindowWidth / 2.0f) + cameraX;

float worldMouseY = -(mouseY - currentWindowHeight / 2.0f) + cameraY;

float offset = 65.0f \* 0.6f;

float bulletX = cameraX + cos(playerAngle) \* offset;

float bulletY = cameraY + sin(playerAngle) \* offset;

recoilTimer = 5;

shootBullet(bulletX, bulletY, worldMouseX, worldMouseY);

playDisparo();

}

if (key == 27) { // ESC: toggle pausa SOLO en juego

enPausa = !enPausa;

return;

}

}

void procesarTeclaUp(unsigned char key, int x, int y) {

keys[key] = false;

}

void procesarTeclaEspecial(int key, int x, int y) {

// OJO: GLUT NO entrega Enter/Esc aquí. Solo flechas/F1..F12/etc.

// Navegación con flechas en menús

if (gameState == STATE\_MENU) {

if (key == GLUT\_KEY\_UP) mainMenuIndex = (mainMenuIndex + 3 - 1) % 3;

if (key == GLUT\_KEY\_DOWN) mainMenuIndex = (mainMenuIndex + 1) % 3;

return;

}

if (gameState == STATE\_SETTINGS) {

if (key == GLUT\_KEY\_UP) settingsMenuIndex = (settingsMenuIndex + 2 - 1) % 2;

if (key == GLUT\_KEY\_DOWN) settingsMenuIndex = (settingsMenuIndex + 1) % 2;

if (settingsMenuIndex == 0) {

if (key == GLUT\_KEY\_LEFT) {

masterVolumePercent = std::max(0, masterVolumePercent - 5);

setMasterVolume(masterVolumePercent);

} else if (key == GLUT\_KEY\_RIGHT) {

masterVolumePercent = std::min(100, masterVolumePercent + 5);

setMasterVolume(masterVolumePercent);

}

}

return;

}

// ===== De aquí en adelante: STATE\_GAME =====

if (gameOverActivo) {

if (key == GLUT\_KEY\_UP) opcionGameOver = (opcionGameOver - 1 + 2) % 2;

if (key == GLUT\_KEY\_DOWN) opcionGameOver = (opcionGameOver + 1) % 2;

return;

}

if (key == GLUT\_KEY\_F12) {

if (isFullScreen) {

glutReshapeWindow(windowedWidth, windowedHeight);

glutPositionWindow(windowPosX, windowPosY);

isFullScreen = false;

} else {

glutFullScreen();

isFullScreen = true;

}

}

}

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

if (gameState != STATE\_GAME) return;

if (gameOverActivo) return;

if (enPausa) return;

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

float worldMouseX = (x - currentWindowWidth / 2.0f) + cameraX;

float worldMouseY = -(y - currentWindowHeight / 2.0f) + cameraY;

float offset = 65.0f \* 0.6f;

float bulletX = cameraX + cos(playerAngle) \* offset;

float bulletY = cameraY + sin(playerAngle) \* offset;

recoilTimer = 5;

shootBullet(bulletX, bulletY, worldMouseX, worldMouseY);

playDisparo();

}

}

void procesarMouseMove(int x, int y) {

if (gameState != STATE\_GAME) return;

if (gameOverActivo) return;

if (enPausa) return;

mouseX = x;

mouseY = y;

float worldMouseX = (x - currentWindowWidth / 2.0f) + cameraX;

float worldMouseY = -(y - currentWindowHeight / 2.0f) + cameraY;

float dx = worldMouseX - cameraX;

float dy = worldMouseY - cameraY;

playerAngle = atan2(dy, dx);

}

void pauseMenuKeys(unsigned char key) {

if (key == 'w') {

opcionSeleccionada--;

if (opcionSeleccionada < 0) opcionSeleccionada = 3; // totalOpciones-1

} else if (key == 's') {

opcionSeleccionada++;

if (opcionSeleccionada > 3) opcionSeleccionada = 0; // wrap

} else if (key == 13) { // ENTER

if (opcionSeleccionada == 0) { // Continuar

enPausa = false;

} else if (opcionSeleccionada == 1) { // Reiniciar nivel

reinicioPendiente = true;

enFade = true;

estadoFade = FADE\_OUT;

fadeAlpha = 0.0f;

} else if (opcionSeleccionada == 2) { // Música ON/OFF

if (Mix\_PausedMusic()) {

Mix\_ResumeMusic();

} else {

Mix\_PauseMusic();

}

} else if (opcionSeleccionada == 3) { // Salir

exit(0);

}

}

}

Input.h:

#ifndef INPUT\_H

#define INPUT\_H

void procesarTecla(unsigned char key, int x, int y);

void procesarTeclaUp(unsigned char key, int x, int y);

void procesarTeclaEspecial(int key, int x, int y);

void procesarMouseClick(int button, int state, int x, int y);

void procesarMouseMove(int x, int y);

void pauseMenuKeys(unsigned char key);

// ===== NUEVO: handlers de menús =====

void settingsMenuKeys(unsigned char key);

#endif

Main.cpp:

#include <GL/freeglut.h>

#include "player.h"

#include "map.h"

#include "bullet.h"

#include <iostream>

#include "collision.h"

#include "enemy.h"

#include "globals.h"

#include <cmath>

#include <SDL.h>

#include <SDL\_mixer.h> // o solo <SDL\_mixer.h> si lo dejaste en include raíz

#include "sound.h"

#include "draw.h"

#include "input.h"

#include "enemy\_bullet.h"

#include "sprite\_logo.h"

void ReiniciarJuego(); // ?? Declaración anticipada

void drawGameOverMenu();

int score = 0;

int playerHealth = 3; // Comienza con 3 vidas

int invulnerableTimer = 0;

int recoilTimer = 0;

bool enPausa = false;

int opcionSeleccionada = 0; // 0 = Continuar, 1 = Reiniciar, etc.

const char\* opcionesMenu[] = {

"Continuar",

"Reiniciar nivel",

"Musica: Activada",

"Salir del juego"

};

const int totalOpciones = 4;

float fadeAlpha = 0.0f;

bool enFade = false;

FadeEstado estadoFade = NINGUNO;

int tutorialPaso = 1; // Comienza mostrando el primer mensaje

int tutorialTimer = 0; // Empieza en 0

float tutorialAlpha = 0.0f;

int tutorialEstado = 0;

bool gameOverActivo = false;

float gameOverAlpha = 0.0f;

int opcionGameOver = 0;

int gameOverTimer = 0;

bool reinicioPendiente = false;

bool bossAlive = false;

bool bossExploding = false;

int bossExplodeTimer = 0;

float bossDeathX = 0.0f, bossDeathY = 0.0f;

bool creditsActive = false;

int creditsTimer = 0;

GameState gameState = STATE\_MENU;

int mainMenuIndex = 0; // 0 Iniciar, 1 Ajustes, 2 Salir

int settingsMenuIndex = 0; // 0 Volumen, 1 Volver

int masterVolumePercent = 25;

bool startGamePending = false;

int nivelActual = 1;

const int TOTAL\_NIVELES = 3;

bool cambioNivelPendiente = false;

bool isFullScreen = true;

int windowedWidth = 800;

int windowedHeight = 600;

int windowPosX = 100;

int windowPosY = 100;

int currentWindowWidth = 800;

int currentWindowHeight = 600;

// Cámara

float cameraX = 0.0f;

float cameraY = 0.0f;

float playerSpeed = 4.0f;

float playerAngle = 0.0f;

const float POS\_INICIAL\_X = -1512;

const float POS\_INICIAL\_Y = 164;

// Teclas presionadas

bool keys[256] = { false };

int mouseX = 0;

int mouseY = 0;

void drawFadeOverlay() {

if (!enFade) return;

glDisable(GL\_TEXTURE\_2D);

glMatrixMode(GL\_PROJECTION);

glPushMatrix();

glLoadIdentity();

gluOrtho2D(0, currentWindowWidth, 0, currentWindowHeight);

glMatrixMode(GL\_MODELVIEW);

glPushMatrix();

glLoadIdentity();

glEnable(GL\_BLEND);

glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA);

glColor4f(0.0f, 0.0f, 0.0f, fadeAlpha);

glBegin(GL\_QUADS);

glVertex2f(0, 0);

glVertex2f(currentWindowWidth, 0);

glVertex2f(currentWindowWidth, currentWindowHeight);

glVertex2f(0, currentWindowHeight);

glEnd();

glDisable(GL\_BLEND);

glPopMatrix();

glMatrixMode(GL\_PROJECTION);

glPopMatrix();

glMatrixMode(GL\_MODELVIEW);

glEnable(GL\_TEXTURE\_2D);

}

void reshape(int width, int height) {

currentWindowWidth = width;

currentWindowHeight = height;

glViewport(0, 0, width, height);

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

// Mantener la relación 4:3 sin deformar (800x600)

float aspect = width / (float)height;

if (aspect >= (800.0f / 600.0f)) {

// Pantalla más ancha que 4:3

glOrtho(-400.0f \* aspect \* (600.0f / 800.0f), 400.0f \* aspect \* (600.0f / 800.0f),

-300.0f, 300.0f, -1.0f, 1.0f);

} else {

// Pantalla más alta que 4:3

glOrtho(-400.0f, 400.0f,

-300.0f / aspect \* (800.0f / 600.0f), 300.0f / aspect \* (800.0f / 600.0f),

-1.0f, 1.0f);

}

glMatrixMode(GL\_MODELVIEW);

glLoadIdentity();

}

void menuKeys(unsigned char key) {

if (key == 'w') {

opcionSeleccionada--;

if (opcionSeleccionada < 0) opcionSeleccionada = totalOpciones - 1;

} else if (key == 's') {

opcionSeleccionada++;

if (opcionSeleccionada >= totalOpciones) opcionSeleccionada = 0;

} else if (key == 13) { // ENTER

if (opcionSeleccionada == 0) {

enPausa = false; // Continuar

} else if (opcionSeleccionada == 1) {

reinicioPendiente = true; // <- FALTABA ESTO

enFade = true;

estadoFade = FADE\_OUT;

fadeAlpha = 0.0f;

}

else if (opcionSeleccionada == 2) {

if (Mix\_PausedMusic()) {

Mix\_ResumeMusic();

opcionesMenu[2] = "Musica: Activada";

} else {

Mix\_PauseMusic();

opcionesMenu[2] = "Musica: Desactivada";

}

} else if (opcionSeleccionada == 3) {

exit(0);

}

}

}

void drawHUD() {

// Cambia a proyección 2D para HUD

glMatrixMode(GL\_PROJECTION);

glPushMatrix();

glLoadIdentity();

gluOrtho2D(0, currentWindowWidth, 0, currentWindowHeight); // Coordenadas de pantalla

glMatrixMode(GL\_MODELVIEW);

glPushMatrix();

glLoadIdentity();

// Dibujar el HUD en la esquina superior izquierda

glColor3f(1, 1, 1); // Blanco

glRasterPos2i(10, currentWindowHeight - 20); // 10 px a la derecha, 20 px debajo del top

std::string hudText = "Score: " + std::to\_string(score);

for (char c : hudText)

glutBitmapCharacter(GLUT\_BITMAP\_9\_BY\_15, c);

// Restaurar la proyección original

glPopMatrix(); // MODELVIEW

glMatrixMode(GL\_PROJECTION);

glPopMatrix();

glMatrixMode(GL\_MODELVIEW); // Volver a modo normal

}

// FPS / Lógica de movimiento

void timer(int value) {

// === 1) PAUSA (solo aplica en juego y si no hay fade) =====================

if (gameState == STATE\_GAME && enPausa && !enFade) {

glutPostRedisplay(); // Redibujar el menú de pausa

glutTimerFunc(16, timer, 0); // ~60 FPS

return;

}

// === 2) REINICIO / CAMBIO DE NIVEL (cuando el fade está en negro) ========

if (reinicioPendiente && fadeAlpha >= 1.0f) {

ReiniciarJuego();

reinicioPendiente = false;

estadoFade = FADE\_IN;

}

else if (gameState == STATE\_GAME && cambioNivelPendiente && fadeAlpha >= 1.0f) {

if (nivelActual > TOTAL\_NIVELES) nivelActual = 1;

std::string nombreMapa = "map" + std::to\_string(nivelActual) + ".txt";

loadMap(nombreMapa.c\_str());

initEnemiesFromMap();

playerHealth = 3;

bullets.clear();

enemyBullets.clear();

estadoFade = FADE\_IN;

cambioNivelPendiente = false;

}

// === 3) FADES (menu<->juego y transiciones) ==============================

if (enFade) {

if (estadoFade == FADE\_OUT) {

fadeAlpha += 0.02f;

if (fadeAlpha >= 1.0f) {

fadeAlpha = 1.0f;

// Transición MENÚ ? JUEGO

if (startGamePending && gameState == STATE\_MENU) {

nivelActual = 1;

loadMap("map.txt");

initEnemiesFromMap();

playerHealth = 3;

bullets.clear();

enemyBullets.clear();

// Música de juego

playGameMusic();

gameState = STATE\_GAME;

startGamePending = false;

}

estadoFade = FADE\_IN;

}

} else if (estadoFade == FADE\_IN) {

fadeAlpha -= 0.02f;

if (fadeAlpha <= 0.0f) {

fadeAlpha = 0.0f;

enFade = false;

estadoFade = NINGUNO;

// Al terminar un fade, quitamos pausa si estábamos en juego

if (gameState == STATE\_GAME) enPausa = false;

}

}

}

// === 4) GAME OVER (solo cuando estamos jugando) ==========================

if (gameState == STATE\_GAME && !gameOverActivo && playerHealth <= 0) {

gameOverActivo = true;

gameOverAlpha = 0.0f;

gameOverTimer = 0;

opcionGameOver = 0;

Mix\_PauseMusic();

}

// === 5) SIMULACIÓN DEL JUEGO (solo en STATE\_GAME) ========================

if (gameState == STATE\_GAME) {

if (recoilTimer > 0) recoilTimer--;

float speed = playerSpeed;

float nextX = cameraX;

float nextY = cameraY;

if (keys['w']) nextY += speed;

if (keys['s']) nextY -= speed;

if (keys['a']) nextX -= speed;

if (keys['d']) nextX += speed;

// Colisiones

if (canMoveTo(nextX, cameraY)) cameraX = nextX;

if (canMoveTo(cameraX, nextY)) cameraY = nextY;

if (invulnerableTimer > 0) invulnerableTimer--;

// Tutorial con fade

if (tutorialPaso <= 2) {

if (tutorialEstado == 0) { // FADE IN

tutorialAlpha += 0.02f;

if (tutorialAlpha >= 1.0f) {

tutorialAlpha = 1.0f;

tutorialEstado = 1;

tutorialTimer = 0;

}

} else if (tutorialEstado == 1) { // ESPERA

tutorialTimer++;

if (tutorialTimer >= 180) { // ~3s

tutorialEstado = 2;

}

} else if (tutorialEstado == 2) { // FADE OUT

tutorialAlpha -= 0.02f;

if (tutorialAlpha <= 0.0f) {

tutorialAlpha = 0.0f;

tutorialPaso++;

tutorialEstado = 0;

}

}

}

// Portal -> siguiente paso

if (!enFade && !cambioNivelPendiente && isOnPortal(cameraX, cameraY)) {

if (nivelActual == 3) {

// FINAL: activar créditos

creditsActive = true;

creditsTimer = 0;

stopMusic();

enFade = true;

estadoFade = FADE\_OUT;

fadeAlpha = 0.0f;

// OJO: NO cambiar gameState aquí

} else {

nivelActual++;

if (nivelActual > TOTAL\_NIVELES) nivelActual = 1;

cambioNivelPendiente = true;

enFade = true;

estadoFade = FADE\_OUT;

fadeAlpha = 0.0f;

}

}

// Actualizaciones de juego

updateBullets();

updateEnemyBullets();

updateEnemies(cameraX, cameraY);

if (gameOverActivo && !reinicioPendiente) {

glutPostRedisplay();

glutTimerFunc(16, timer, 0);

return;

}

// === Explosión del boss y drop de portal ===

if (bossExploding) {

bossExplodeTimer--;

if (bossExplodeTimer <= 0) {

bossExploding = false;

// Colocar 'P' en la celda del boss

int tileSize = 40;

int mapHeight = (int)mapData.size();

int mapWidth = (int)mapData[0].size();

int col = (int)((bossDeathX / tileSize) + mapWidth / 2.0f);

int row = (int)((mapHeight / 2.0f) - (bossDeathY / tileSize));

if (row >= 0 && row < mapHeight && col >= 0 && col < mapWidth) {

mapData[row][col] = 'P';

}

}

}

}

// === 6) Redibujar y rearmar timer =======================================

glutPostRedisplay();

glutTimerFunc(16, timer, 0); // ~60 FPS

}

void drawCenteredText(const char\* txt, int y) {

int w = 0;

for (const char\* p = txt; \*p; ++p) w += glutBitmapWidth(GLUT\_BITMAP\_HELVETICA\_18, \*p);

glRasterPos2i((currentWindowWidth - w) / 2, y);

for (const char\* p = txt; \*p; ++p) glutBitmapCharacter(GLUT\_BITMAP\_HELVETICA\_18, \*p);

}

void drawMainMenu() {

glMatrixMode(GL\_PROJECTION);

glPushMatrix(); glLoadIdentity(); gluOrtho2D(0, currentWindowWidth, 0, currentWindowHeight);

glMatrixMode(GL\_MODELVIEW);

glPushMatrix(); glLoadIdentity();

// 1) DIBUJAR LOGO CENTRADO ARRIBA

// - Queremos que el sprite aparezca centrado, y su BASE esté ~200 px por encima del centro

// - Ajusta estos números si quieres el logo más cerca/lejos de las opciones

int centerX = currentWindowWidth / 2;

int baseY = currentWindowHeight - 600; // base del logo

int maxW = (int)(currentWindowWidth \* 1.2f); // ancho máximo 60% pantalla

int maxH = (int)(currentWindowHeight \* 0.90f); // alto máximo 35% pantalla

drawSpriteLogoCentered(centerX, baseY, maxW, maxH);

// Título

//glColor3f(1,1,1);

//drawCenteredText("== CONTRA TOP-DOWN ==", currentWindowHeight - 120);

const char\* items[] = {"Iniciar juego", "Ajustes", "Salir"};

for (int i=0;i<3;++i) {

if (i == mainMenuIndex) glColor3f(1,1,0);

else glColor3f(1,1,1);

drawCenteredText(items[i], currentWindowHeight/2 - i\*30);

}

glPopMatrix();

glMatrixMode(GL\_PROJECTION);

glPopMatrix();

glMatrixMode(GL\_MODELVIEW);

}

void drawSettingsMenu() {

glMatrixMode(GL\_PROJECTION);

glPushMatrix(); glLoadIdentity(); gluOrtho2D(0, currentWindowWidth, 0, currentWindowHeight);

glMatrixMode(GL\_MODELVIEW);

glPushMatrix(); glLoadIdentity();

glColor3f(1,1,1);

drawCenteredText("== AJUSTES ==", currentWindowHeight - 120);

// Opción 0: Volumen

if (settingsMenuIndex == 0) glColor3f(1,1,0); else glColor3f(1,1,1);

std::string vol = "Volumen: " + std::to\_string(masterVolumePercent) + "% (A/D o ?/?)";

drawCenteredText(vol.c\_str(), currentWindowHeight/2);

// Opción 1: Volver

if (settingsMenuIndex == 1) glColor3f(1,1,0); else glColor3f(1,1,1);

drawCenteredText("Volver", currentWindowHeight/2 - 40);

glPopMatrix();

glMatrixMode(GL\_PROJECTION);

glPopMatrix();

glMatrixMode(GL\_MODELVIEW);

}

static void drawBossExplosion() {

if (!bossExploding) return;

// Proyección mundo (ya estás en mundo al llamar)

int rings = 5;

float t = (float)bossExplodeTimer / 90.0f; // 1 -> 0

float maxR = 180.0f;

for (int i=0;i<rings;i++) {

float f = (i+1)/(float)rings;

float r = (1.0f - t) \* maxR \* f;

glColor4f(1.0f, 0.6f \* (1.0f - f), 0.1f, 0.8f \* (1.0f - t));

glBegin(GL\_LINE\_LOOP);

const int seg = 36;

for (int k=0;k<seg;k++){

float ang = (2.0f\*3.1415926f\*k)/seg;

glVertex2f(bossDeathX + cos(ang)\*r, bossDeathY + sin(ang)\*r);

}

glEnd();

}

}

void display() {

glClear(GL\_COLOR\_BUFFER\_BIT);

// --- Créditos en pantalla (tiene prioridad sobre el juego) ---

if (creditsActive) {

drawCreditsScreen();

drawFadeOverlay();

glutSwapBuffers();

return;

}

if (gameState == STATE\_MENU) {

drawMainMenu();

drawFadeOverlay();

glutSwapBuffers();

return;

} else if (gameState == STATE\_SETTINGS) {

drawSettingsMenu();

drawFadeOverlay();

glutSwapBuffers();

return;

}

// --- Siempre dibujar el mundo ---

glPushMatrix();

glTranslatef(-cameraX, -cameraY, 0); // Mover el mundo

drawMap(); // Suelo y paredes

drawBullets(); // Disparos

drawEnemyBullets();

drawPlayer(); // Jugador

drawHealthBar(); // Barra de vida

drawEnemies(); // Enemigos

drawBossExplosion();

drawHUD(); // Puntaje

drawGameOverScreen(); // ya lo tienes

drawGameOverMenu(); // <-- agrégalo justo después

glPopMatrix(); // Fin del modo mundo

// --- Mostrar menú de pausa solo si no estamos en fade ---

if (enPausa && !enFade) {

glMatrixMode(GL\_PROJECTION);

glPushMatrix();

glLoadIdentity();

gluOrtho2D(0, currentWindowWidth, 0, currentWindowHeight);

glMatrixMode(GL\_MODELVIEW);

glPushMatrix();

glLoadIdentity();

int anchoRecuadro = 300;

int altoRecuadro = totalOpciones \* 40 + 40;

int xCentro = currentWindowWidth / 2;

int yCentro = currentWindowHeight / 2;

int xRecuadro = xCentro - anchoRecuadro / 2;

int yRecuadro = yCentro + altoRecuadro / 2;

// Fondo del recuadro

glColor3f(0.05f, 0.05f, 0.05f);

glBegin(GL\_QUADS);

glVertex2i(xRecuadro, yRecuadro);

glVertex2i(xRecuadro + anchoRecuadro, yRecuadro);

glVertex2i(xRecuadro + anchoRecuadro, yRecuadro - altoRecuadro);

glVertex2i(xRecuadro, yRecuadro - altoRecuadro);

glEnd();

// Borde

glColor3f(0.6f, 0.6f, 0.6f);

glBegin(GL\_LINE\_LOOP);

glVertex2i(xRecuadro, yRecuadro);

glVertex2i(xRecuadro + anchoRecuadro, yRecuadro);

glVertex2i(xRecuadro + anchoRecuadro, yRecuadro - altoRecuadro);

glVertex2i(xRecuadro, yRecuadro - altoRecuadro);

glEnd();

// Título

glColor3f(1.0f, 1.0f, 1.0f);

glRasterPos2i(xCentro - 60, yRecuadro - 30);

const char\* titulo = "== PAUSA ==";

for (int i = 0; titulo[i] != '\0'; ++i)

glutBitmapCharacter(GLUT\_BITMAP\_HELVETICA\_18, titulo[i]);

// Opciones

int yTexto = yRecuadro - 70;

for (int i = 0; i < totalOpciones; i++) {

if (i == opcionSeleccionada)

glColor3f(1.0f, 1.0f, 0.0f); // Amarillo

else

glColor3f(1.0f, 1.0f, 1.0f); // Blanco

glRasterPos2i(xCentro - 100, yTexto - i \* 30);

const char\* texto = opcionesMenu[i];

for (int j = 0; texto[j] != '\0'; ++j)

glutBitmapCharacter(GLUT\_BITMAP\_HELVETICA\_18, texto[j]);

}

glPopMatrix(); // MODELVIEW

glMatrixMode(GL\_PROJECTION);

glPopMatrix();

glMatrixMode(GL\_MODELVIEW);

}

// --- Dibujar overlay de fade si está activo ---

drawFadeOverlay();

drawLoadingLevelText();

drawTutorialText();

glutSwapBuffers();

}

void drawGameOverMenu() {

if (!gameOverActivo) return;

glMatrixMode(GL\_PROJECTION);

glPushMatrix();

glLoadIdentity();

gluOrtho2D(0, currentWindowWidth, 0, currentWindowHeight);

glMatrixMode(GL\_MODELVIEW);

glPushMatrix();

glLoadIdentity();

int xCentro = currentWindowWidth / 2;

int yCentro = currentWindowHeight / 2;

const char\* titulo = "GAME OVER";

const char\* opciones[] = { "Reintentar", "Salir" };

// Fondo oscuro del menú

glColor4f(0.0f, 0.0f, 0.0f, 0.7f);

glBegin(GL\_QUADS);

glVertex2f(0, 0);

glVertex2f(currentWindowWidth, 0);

glVertex2f(currentWindowWidth, currentWindowHeight);

glVertex2f(0, currentWindowHeight);

glEnd();

// Texto de título

glColor3f(1, 0, 0);

glRasterPos2i(xCentro - 50, yCentro + 50);

for (const char\* c = titulo; \*c; c++)

glutBitmapCharacter(GLUT\_BITMAP\_HELVETICA\_18, \*c);

// Opciones

for (int i = 0; i < 2; i++) {

if (i == opcionGameOver)

glColor3f(1, 1, 0); // Amarillo para opción seleccionada

else

glColor3f(1, 1, 1); // Blanco normal

glRasterPos2i(xCentro - 40, yCentro - i \* 30);

for (const char\* c = opciones[i]; \*c; c++)

glutBitmapCharacter(GLUT\_BITMAP\_HELVETICA\_18, \*c);

}

glPopMatrix();

glMatrixMode(GL\_PROJECTION);

glPopMatrix();

glMatrixMode(GL\_MODELVIEW);

}

void ReiniciarJuego() {

// Restaurar jugador

enemies.clear();

cameraX = POS\_INICIAL\_X;

cameraY = POS\_INICIAL\_Y;

playerHealth = 3;

score = 0;

// Limpiar y recargar enemigos

loadMap("map.txt");

initEnemiesFromMap();

// Limpiar balas

bullets.clear();

// Reiniciar estados visuales (no desactivar GameOver aún)

gameOverAlpha = 0.0f;

opcionGameOver = 0;

// Tutorial si se desea

tutorialPaso = 1;

tutorialAlpha = 0.0f;

tutorialEstado = 0;

tutorialTimer = 0;

// Reanudar música

Mix\_ResumeMusic();

// ? Aquí sí desactivamos Game Over

gameOverActivo = false;

}

// Al soltar una tecla

void keyboardUp(unsigned char key, int x, int y) {

keys[key] = false;

}

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

glutInit(&argc, argv);

glutInitWindowSize(800, 600);

glutInitDisplayMode(GLUT\_RGB | GLUT\_DOUBLE);

glutCreateWindow("Contra Top-Down");

glutFullScreen(); // ?? Esto activa pantalla completa

// Proyección ortográfica

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

glOrtho(-400, 400, -300, 300, -1, 1);

glMatrixMode(GL\_MODELVIEW);

glLoadIdentity();

// Callbacks

glutDisplayFunc(display);

// Ruta al .txt del logo del menú (ajústala a tu archivo)

cargarSpriteLogo("Sprite\_inicio/contra3\_logo\_300x300\_transparent\_bg\_redwhite.txt");

glutKeyboardFunc(procesarTecla);

glutKeyboardUpFunc(procesarTeclaUp);

glutSpecialFunc(procesarTeclaEspecial);

glutMouseFunc(procesarMouseClick);

glutPassiveMotionFunc(procesarMouseMove);

glutReshapeFunc(reshape);

glutTimerFunc(0, timer, 0);

cargarSpriteJugador("personajes/Personaje\_principal.txt");

// Cargar mapa

//loadMap("map.txt");

//initEnemiesFromMap();

//std::cout << "Tile inicial: " << isWall(cameraX, cameraY) << std::endl;

// Centrar la cámara

//cameraX = POS\_INICIAL\_X;

//cameraY = POS\_INICIAL\_Y;

if (!initSounds()) return 1;

// ===== NUEVO: arrancar en MENÚ con música de menú

gameState = STATE\_MENU;

masterVolumePercent = 25; // volumen inicial

setMasterVolume(masterVolumePercent);

playMenuMusic();

//playMusic();

glutMainLoop();

cleanSounds();

return 0;

}

Map.cpp:

// === FILE: Map.cpp ===

#include "map.h"

#include "enemy.h"

#include <fstream>

#include <iostream>

#include <vector>

#include <GL/freeglut.h>

#include <cmath> // Necesario para sin/cos

#include "globals.h"

#include <algorithm>

const int tileSize = 40;

float nextSpawnX = POS\_INICIAL\_X;

float nextSpawnY = POS\_INICIAL\_Y;

std::vector<std::string> mapData;

int maxColumns = 0;

void loadMap(const char\* filename) {

mapData.clear();

std::ifstream file(filename);

if (!file.is\_open()) {

std::cout << "ERROR: No se pudo abrir el archivo " << filename << std::endl;

return;

}

std::string line;

size\_t maxLength = 0;

std::vector<std::string> temp;

// Leer líneas y buscar '@' al mismo tiempo

int spawnRow = -1, spawnCol = -1;

while (std::getline(file, line)) {

if (line.length() > maxLength)

maxLength = line.length();

temp.push\_back(line);

}

file.close();

for (int row = 0; row < temp.size(); ++row) {

while (temp[row].length() < maxLength)

temp[row] += '.';

for (int col = 0; col < maxLength; ++col) {

if (temp[row][col] == '@') {

spawnRow = row;

spawnCol = col;

temp[row][col] = '.';

}

}

mapData.push\_back(temp[row]);

}

maxColumns = maxLength;

std::cout << "Mapa cargado con " << mapData.size() << " filas y " << maxColumns << " columnas.\n";

if (spawnRow != -1 && spawnCol != -1) {

nextSpawnX = (spawnCol - maxColumns / 2.0f) \* tileSize;

nextSpawnY = (mapData.size() / 2.0f - spawnRow) \* tileSize;

cameraX = nextSpawnX;

cameraY = nextSpawnY;

std::cout << "[DEBUG] Punto de aparición: (" << nextSpawnX << ", " << nextSpawnY << ")\n";

} else {

std::cout << "[ADVERTENCIA] No se encontró '@' en el mapa. Usando spawn por defecto.\n";

}

int countK = 0, countk = 0;

for (auto &row : mapData) {

countK += std::count(row.begin(), row.end(), 'K');

countk += std::count(row.begin(), row.end(), 'k'); // por si viniera en minúscula

}

std::cout << "[DEBUG] K en mapa tras load: " << countK

<< " | k: " << countk << std::endl;

}

void initEnemiesFromMap() {

enemies.clear();

const int tileSize = 40;

if (mapData.empty()) {

std::cerr << "Error: Mapa vacío al generar enemigos\n";

return;

}

const int mapHeight = mapData.size();

const int mapWidth = mapData[0].size();

for (int row = 0; row < mapHeight; ++row) {

if (row >= mapData.size()) {

std::cerr << "Error: Fila " << row << " fuera de rango\n";

continue;

}

for (int col = 0; col < mapWidth; ++col) {

char symbol = mapData[row][col];

float worldX = (col - mapWidth / 2.0f) \* tileSize;

float worldY = (mapHeight / 2.0f - row) \* tileSize;

if (addEnemyFromMapChar(symbol, worldX, worldY)) {

mapData[row][col] = '.'; // Marcar como procesado

}

}

}

cleanupInvalidEnemies(); // Limpiar cualquier enemigo inválido

}

void drawMap() {

int tileSize = 40;

int mapHeight = mapData.size();

int mapWidth = mapData[0].size();

for (int row = 0; row < mapHeight; ++row) {

for (int col = 0; col < mapWidth; ++col) {

char tile = mapData[row][col];

float posX = (col - mapWidth / 2.0f) \* tileSize;

float posY = (mapHeight / 2.0f - row) \* tileSize;

// Dibujar piso para todas las celdas no-pared

if (tile != '#') {

// Color de piso original (gris oscuro)

glColor3f(0.15f, 0.15f, 0.15f);

glBegin(GL\_QUADS);

glVertex2f(posX, posY);

glVertex2f(posX + tileSize, posY);

glVertex2f(posX + tileSize, posY - tileSize);

glVertex2f(posX, posY - tileSize);

glEnd();

}

// Dibujar paredes

if (tile == '#') {

glColor3f(0.4f, 0.4f, 0.4f);

glBegin(GL\_QUADS);

glVertex2f(posX, posY);

glVertex2f(posX + tileSize, posY);

glVertex2f(posX + tileSize, posY - tileSize);

glVertex2f(posX, posY - tileSize);

glEnd();

}

if (tile == 'P') {

glColor3f(1.0f, 0.5f, 0.0f); // Naranja

glBegin(GL\_QUADS);

glVertex2f(posX, posY);

glVertex2f(posX + tileSize, posY);

glVertex2f(posX + tileSize, posY - tileSize);

glVertex2f(posX, posY - tileSize);

glEnd();

}

}

}

}

Map.h:

#ifndef MAP\_H

#define MAP\_H

#include <vector>

#include <string>

extern std::vector<std::string> mapData;

void loadMap(const char\* filename);

void drawMap();

void initEnemiesFromMap();

#endif

Map.txt:

################################## ################################

#....#......#.........E........E.# #............................S.#

#....#......#....................# #...E..........###########.....#

#....#......#...........#......E.# #..............#...............#

#....#......#...........#........# #..#############...E...........#

##..##......##########..#...#########################..#...E.......................#

#...........#...........#....................F.........#.......E.......E...........#

#...........#...........#....................E.........#...#########################

#...........#...........#....................F.........#...........#.....E.........#

#...........#####....####...####################################...#...............#

#...........#...........#........# #..........E...########.E......#

#...............E.......#....E...# #............................S.#

#...........#...........#E......E# #........E.........F.....E.....#

################################## ############################...#

##............#

##............#

###################............#

#................##............#

#..S.............##............#

#################################.......E....##..##............#

#.....F...........E..........................##..##............#

#.........S.......................E..........##..##............#

#####################.....................................F......##..##............#

#...................#...............######...########............##..##............#

P...................................#......F........#............##................#

P.......@................S..........#......F........#...F........##.......S..S..S..#

#...................#...............#...............#............##................#

####################################################################################

Map2.txt:

##################################

#....#................E........E.# #######################

#....#.......F...................# #.....................#

#....#.........................E.# #.....E....E.......S..#

#....#.............E.............#########################.....................######

##..##...................................S..........................................#

#......................E..E....E.........S.......E..........F.............F.........#

#...................E........F.E....................................................#

#......................F..F......################################################...#

#.............F.....E..F..E..E...# #...#

#...................F.....F.F....# #...#

#...............E......F.....E...# #...#

#........................E...E..E# #...#

################################## #############################...#

#.............E...S.........#...#

#..E...F....................#...#

#..F.....E....F.........E...#...#

#....############...F.......#...#

#..F.# #...........#...#

#....# #...........#.F.#

#....# #.....F.....#FSF#

#....# #...............#

#..E.# #....F..........#

#....# #...........E...#

#....# #################

#....#

#################################....#

#...............................#....#

#......@....#################...#..E.#

#...........# #...#.E..#

#...........# #...#.E..#

#####PP###### #...#..E.#

############# #.E......#

#...E..E.#

#........#

##########

Map3.txt:

##################################

#....#................E........E.# #######################

#....#.......F...................# #.....................#

#....#.........................E.# #.....E....E.......S..#

#....#.............E.............#########################.....................######

##..##...................................S..........................................#

#......................E..E....E.........S.......E..........F.............F.........#

#...................E........F.E....................................................#

#......................F..F......################################################...#

#.............F.....E..F..E..E...# #...#

#...................F.....F.F....# #...#

#...............E......F.....E...# #...#

#........................E...E..E# #...#

################################## #############################...#

#.............E...S.........#...#

#..E...F....................#...#

#..F.....E....F.........E...#...#

#....############...F.......#...#

#..F.# #...........#...#

#....# #...........#.F.#

###################################### #....# #.....F.....#FSF#

#....................................# #....# #...............#

#....................................# #..E.# #....F..........#

#....................................# #....# #...........E...#

#........K...........................# #....# #################

#....................................# #....#

#...........................#########################....#

#...................................................#....#

#...............................#################...#....#

#...............................# #...#....#

#...............................# #...#....#

################################# #...#....#

############# #........#

#...@....#

#........#

##########

Player.cpp:

#include <GL/freeglut.h>

#include "player.h"

#include "globals.h"

#include <cmath>

#include <fstream>

#include <string>

#include <vector>

#ifndef M\_PI

#define M\_PI 3.14159265358979323846

#endif

extern float cameraX;

extern float cameraY;

extern int mouseX;

extern int mouseY;

std::vector<std::string> spriteJugador;

void cargarSpriteJugador(const char\* path) {

spriteJugador.clear();

std::ifstream archivo(path);

std::string linea;

while (std::getline(archivo, linea)) {

spriteJugador.push\_back(linea);

}

}

void drawPlayer() {

if (invulnerableTimer > 0 && (invulnerableTimer / 5) % 2 == 0)

return;

float px = cameraX;

float py = cameraY;

float angle = playerAngle \* 180.0f / M\_PI;

float recoilOffset = (recoilTimer > 0) ? -3.0f : 0.0f;

glPushMatrix();

glTranslatef(px, py, 0);

glRotatef(playerAngle \* 180.0f / M\_PI, 0, 0, 1);

glTranslatef(0, recoilOffset, 0); // aplicar recoil hacia atrás

const int CENTRO\_FILA = 8; // línea 9

const int CENTRO\_COLUMNA = 12; // columna 13

float cellSize = 1.6f; // tamaño de cada celda en unidades del mundo

float ancho = spriteJugador[0].length();

float alto = spriteJugador.size();

float origenX = -CENTRO\_COLUMNA \* cellSize;

float origenY = CENTRO\_FILA \* cellSize;

for (size\_t fila = 0; fila < spriteJugador.size(); ++fila) {

for (size\_t col = 0; col < spriteJugador[fila].length(); ++col) {

char c = spriteJugador[fila][col];

if (c == '.') continue;

// --- COLORES ---

if (c == 'X') glColor3f(0.0f, 0.0f, 0.0f);

else if (c == 'H') glColor3f(0.23f, 0.16f, 0.15f);

else if (c == 'C') glColor3f(0.69f, 0.43f, 0.24f);

else if (c == 'V') glColor3f(0.95f, 0.74f, 0.64f);

else if (c == 'P') glColor3f(0.70f, 0.49f, 0.49f);

else if (c == 'G') glColor3f(0.27f, 0.27f, 0.27f);

else if (c == 'W') glColor3f(1.0f, 1.0f, 1.0f);

else if (c == 'R') glColor3f(1.0f, 0.0f, 0.0f);

else if (c == 'B') glColor3f(0.0f, 0.3f, 1.0f);

else if (c == 'L') glColor3f(0.7f, 0.7f, 0.7f);

else if (c == 'Y') glColor3f(1.0f, 1.0f, 0.0f);

else if (c == 'M') glColor3f(1.0f, 0.0f, 1.0f);

else if (c == 'T') glColor3f(0.0f, 0.5f, 0.0f);

else if (c == 'Z') glColor3f(0.5f, 1.0f, 0.5f);

else if (c == 'O') glColor3f(1.0f, 0.5f, 0.0f);

else if (c == 'A') glColor3f(0.0f, 0.0f, 0.5f);

else glColor3f(1.0f, 1.0f, 1.0f); // por defecto

float x = origenX + col \* cellSize;

float y = origenY - fila \* cellSize;

glBegin(GL\_QUADS);

glVertex2f(x, y);

glVertex2f(x + cellSize, y);

glVertex2f(x + cellSize, y - cellSize);

glVertex2f(x, y - cellSize);

glEnd();

}

}

glPopMatrix();

}

void drawHealthBar() {

float barWidth = 30.0f;

float barHeight = 5.0f;

float px = cameraX;

float py = cameraY;

float xLeft = px - barWidth / 2.0f;

float yTop = py - 20.0f; // Justo debajo del jugador

// Fondo de la barra (rojo)

glColor3f(0.3f, 0.0f, 0.0f);

glBegin(GL\_QUADS);

glVertex2f(xLeft, yTop);

glVertex2f(xLeft + barWidth, yTop);

glVertex2f(xLeft + barWidth, yTop - barHeight);

glVertex2f(xLeft, yTop - barHeight);

glEnd();

// Parte verde proporcional a la vida

float healthRatio = playerHealth / 3.0f; // 3 es la vida máxima

glColor3f(0.0f, 1.0f, 0.0f);

glBegin(GL\_QUADS);

glVertex2f(xLeft, yTop);

glVertex2f(xLeft + barWidth \* healthRatio, yTop);

glVertex2f(xLeft + barWidth \* healthRatio, yTop - barHeight);

glVertex2f(xLeft, yTop - barHeight);

glEnd();

}

Player.h:

#ifndef PLAYER\_H

#define PLAYER\_H

void drawPlayer();

void drawHealthBar();

void cargarSpriteJugador(const char\* path);

#endif

Sound.cpp:

#include "sound.h"

#include <SDL.h>

#include <SDL\_mixer.h>

#include <iostream>

Mix\_Chunk\* sonidoDisparo = NULL;

Mix\_Music\* musicaFondo = NULL; // in-game

Mix\_Music\* musicaMenu = NULL; // ===== NUEVO: música del menú

// default 25% al iniciar; lo ajustamos luego desde main

static int g\_masterVolumePercent = 25;

static int clamp(int v, int lo, int hi) { return v < lo ? lo : (v > hi ? hi : v); }

static int toSDLVol(int percent) { return (MIX\_MAX\_VOLUME \* clamp(percent,0,100)) / 100; }

bool initSounds() {

if (SDL\_Init(SDL\_INIT\_AUDIO) < 0) {

std::cout << "Error al iniciar SDL: " << SDL\_GetError() << std::endl;

return false;

}

if (Mix\_OpenAudio(44100, MIX\_DEFAULT\_FORMAT, 2, 2048) < 0) {

std::cout << "Error al iniciar SDL\_mixer: " << Mix\_GetError() << std::endl;

return false;

}

sonidoDisparo = Mix\_LoadWAV("music/pistol.ogg");

if (!sonidoDisparo) {

std::cout << "Error al cargar sonido de disparo: " << Mix\_GetError() << std::endl;

return false;

}

// ===== CAMBIO: pista del juego (ya la tenías)

musicaFondo = Mix\_LoadMUS("music/Contra III\_song.MP3");

if (!musicaFondo) {

std::cout << "Error al cargar música de juego: " << Mix\_GetError() << std::endl;

return false;

}

// ===== NUEVO: pista del menú principal

musicaMenu = Mix\_LoadMUS("music/DARKWOOD.MP3"); // <-- agrega este archivo

if (!musicaMenu) {

std::cout << "Error al cargar música de menú: " << Mix\_GetError() << std::endl;

return false;

}

setMasterVolume(g\_masterVolumePercent); // aplica % inicial

return true;

}

void setMasterVolume(int percent) {

g\_masterVolumePercent = clamp(percent,0,100);

int vol = toSDLVol(g\_masterVolumePercent);

Mix\_VolumeMusic(vol);

// Ajusta los chunks (disparo) ligeramente más bajo que la música

Mix\_VolumeChunk(sonidoDisparo, (vol \* 3) / 4);

}

void playDisparo() {

Mix\_PlayChannel(-1, sonidoDisparo, 0);

}

void stopMusic() {

Mix\_HaltMusic();

}

void playMenuMusic() {

if (musicaMenu) {

Mix\_HaltMusic();

Mix\_PlayMusic(musicaMenu, -1);

}

}

void playGameMusic() {

if (musicaFondo) {

Mix\_HaltMusic();

Mix\_PlayMusic(musicaFondo, -1);

}

}

void cleanSounds() {

Mix\_FreeChunk(sonidoDisparo);

Mix\_FreeMusic(musicaFondo);

Mix\_FreeMusic(musicaMenu); // ===== NUEVO

Mix\_CloseAudio();

SDL\_Quit();

}

Sound.h:

#ifndef SOUND\_H

#define SOUND\_H

bool initSounds();

void playDisparo();

void playMenuMusic();

void playGameMusic();

void stopMusic();

void setMasterVolume(int percent);

void cleanSounds();

#endif

Sprite\_logo.cpp:

#include "sprite\_logo.h"

#include <vector>

#include <string>

#include <fstream>

#include <GL/freeglut.h>

static std::vector<std::string> g\_logo;

void cargarSpriteLogo(const char\* path) {

g\_logo.clear();

std::ifstream f(path);

std::string line;

while (std::getline(f, line)) {

g\_logo.push\_back(line);

}

}

static void setColorForChar(char c) {

// === Mapeo de colores EXACTO al que usas en tus sprites ===

if (c == 'X') glColor3f(0.0f, 0.0f, 0.0f); // Negro

else if (c == 'H') glColor3f(0.23f, 0.16f, 0.15f); // Marrón oscuro

else if (c == 'C') glColor3f(0.69f, 0.43f, 0.24f); // Marrón claro

else if (c == 'V') glColor3f(0.95f, 0.74f, 0.64f); // Piel clara

else if (c == 'P') glColor3f(0.70f, 0.49f, 0.49f); // Sombra de piel

else if (c == 'G') glColor3f(0.27f, 0.27f, 0.27f); // Gris oscuro

// Colores adicionales

else if (c == 'W') glColor3f(1.0f, 1.0f, 1.0f); // Blanco

else if (c == 'R') glColor3f(1.0f, 0.0f, 0.0f); // Rojo

else if (c == 'B') glColor3f(0.0f, 0.3f, 1.0f); // Azul

else if (c == 'L') glColor3f(0.7f, 0.7f, 0.7f); // Gris claro

else if (c == 'Y') glColor3f(1.0f, 1.0f, 0.0f); // Amarillo

else if (c == 'M') glColor3f(1.0f, 0.0f, 1.0f); // Magenta

else if (c == 'T') glColor3f(0.0f, 0.5f, 0.0f); // Verde oscuro

else if (c == 'Z') glColor3f(0.5f, 1.0f, 0.5f); // Verde claro

else if (c == 'O') glColor3f(1.0f, 0.5f, 0.0f); // Naranja

else if (c == 'A') glColor3f(0.0f, 0.0f, 0.5f); // Azul oscuro

else glColor3f(1.0f, 1.0f, 1.0f); // Default

}

void drawSpriteLogoCentered(int centerX, int bottomY, int maxWidthPx, int maxHeightPx) {

if (g\_logo.empty()) return;

const int rows = (int)g\_logo.size();

const int cols = (int)g\_logo[0].size();

if (rows == 0 || cols == 0) return;

// Calcula el tamaño de celda para que quepa dentro de (maxWidthPx x maxHeightPx)

float cellSize = (float)maxWidthPx / (float)cols;

float cellSizeH = (float)maxHeightPx / (float)rows;

if (cellSizeH < cellSize) cellSize = cellSizeH; // respetar la más restrictiva

// Tamaño final del sprite en píxeles de pantalla

float spriteW = cols \* cellSize;

float spriteH = rows \* cellSize;

// Queremos centrar horizontalmente y apoyar la base en bottomY

float x0 = centerX - spriteW / 2.0f;

float y0 = (float)bottomY; // base

float yTop = y0 + spriteH; // comenzamos a dibujar desde la fila superior

// Dibuja en coordenadas de HUD (ya deben estar activadas fuera)

for (int r = 0; r < rows; ++r) {

float y = yTop - r \* cellSize;

for (int c = 0; c < cols; ++c) {

char ch = g\_logo[r][c];

if (ch == '.') continue; // transparente

setColorForChar(ch);

float x = x0 + c \* cellSize;

glBegin(GL\_QUADS);

glVertex2f(x, y);

glVertex2f(x+cellSize, y);

glVertex2f(x+cellSize, y - cellSize);

glVertex2f(x, y - cellSize);

glEnd();

}

}

}

Sprite\_logo.h:

#ifndef SPRITE\_LOGO\_H

#define SPRITE\_LOGO\_H

#include <string>

// Carga el sprite desde un .txt (con letras de color)

void cargarSpriteLogo(const char\* path);

// Dibuja el sprite centrado en pantalla usando coordenadas de HUD (0..ancho, 0..alto).

// - centerX: centro horizontal (normalmente currentWindowWidth/2)

// - bottomY: coordenada Y donde se apoyará la base del sprite (píxeles de pantalla)

// - maxWidthPx / maxHeightPx: tamaño máximo permitido (para escalar el sprite y que quepa)

void drawSpriteLogoCentered(int centerX, int bottomY, int maxWidthPx, int maxHeightPx);

#endif