Bullet.cpp:

```
#include "bullet.h"
#include "enemy.h"
#include "collision.h"
#include "map.h"
#include <GL/freeglut.h>
#include <vector>
#include <algorithm>
#include <cmath>

// Lista de balas activas
std::vector<Bullet> bullets;
extern std::vector<Enemy> enemies;
extern int score;

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(); ) {
            if (bulletHitsEnemy(it->x, it->y, enemyIt->x, enemyIt->y)) {
                enemyIt->health--; // ? Restar vida al enemigo

                if (enemyIt->health <= 0) {
                    enemyIt = enemies.erase(enemyIt); // ? Eliminar enemigo si muere
                    score += 100;
                } else {
                    ++enemyIt; // ? Solo avanzar si no se eliminó
                }

                hit = true; // ? La bala impactó
                break; // ? Terminar búsqueda con esta bala
            } else {
                ++enemyIt;
            }
        }

        // 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);
}

bool bulletHitsEnemy(float bulletX, float bulletY, float enemyX, float enemyY) {
    float bulletSize = 2.0f;
    float enemySize = 10.0f;

    return
        bulletX + bulletSize > enemyX - enemySize &&
        bulletX - bulletSize < enemyX + enemySize &&
        bulletY + bulletSize > enemyY - enemySize &&
        bulletY - bulletSize < 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);



#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.h:

```
#ifndef DRAW_H
#define DRAW_H

void drawTutorialText();
void drawGameOverScreen();
void drawLoadingLevelText();

#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"

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;
                    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) {
    switch (symbol) {
        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;
        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;
                }
            }



        // 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); // Rojo
      break;
   case ENEMY_BASIC:
      glColor3f(1.0f, 1.0f, 0.0f); // Amarillo
      break;
   case ENEMY_SNIPER:
      glColor3f(0.3f, 0.3f, 1.0f); // Azul claro
      break;
   case ENEMY_FAST:
      glColor3f(0.0f, 1.0f, 0.0f); // Verde
      break;
   default:
      glColor3f(1.0f, 1.0f, 1.0f); // Blanco
}

// Cuerpo del enemigo (cuadrado de 20x20)
glBegin(GL_QUADS);
glVertex2f(e.x - 10, e.y - 10);
glVertex2f(e.x + 10, e.y - 10);
glVertex2f(e.x + 10, e.y + 10);
glVertex2f(e.x - 10, e.y + 10);
glEnd();

// Barra de vida (visible siempre que tenga vida)
if (e.health > 0) {
   float barWidth = 20.0f;
   float barHeight = 3.0f;
   float xLeft = e.x - barWidth / 2.0f;
```

```
            float yTop = e.y - 15.0f;

            // Fondo oscuro
            glColor3f(0.5f, 0.0f, 0.0f);
            glBegin(GL_QUADS);
            glVertex2f(xLeft, yTop);
            glVertex2f(xLeft + barWidth, yTop);
            glVertex2f(xLeft + barWidth, yTop - barHeight);
            glVertex2f(xLeft, yTop - barHeight);
            glEnd();

            // Vida 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() {
    enemies.erase(std::remove_if(enemies.begin(), enemies.end(),
        [](const Enemy& e) {
            // Eliminar enemigos con:
            // - Posiciones inválidas
            // - Salud inválida
```

```
        // - Tipos incorrectos
        return std::isnan(e.x) || std::isnan(e.y) ||
               e.x < -2000 || e.x > 2000 ||
               e.y < -2000 || e.y > 2000 ||
               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_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;
};

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;




#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 "enemy.h" // <--- asegúrate de tenerlo


extern bool keys[256];
extern int mouseX;
extern int mouseY;
```

```
void menuKeys(unsigned char key);


void procesarTecla(unsigned char key, int x, int y) {
        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) {
                 reinicioPendiente = true;
                 fadeAlpha = 0.0f;
                 enFade = true;
                 estadoFade = FADE_OUT;
              } else {
                 exit(0);
              }
           }
           return;
        }


        if (enPausa) {
     menuKeys(key);
     return;
   }
   keys[key] = true;


        if (key == 32) { // SPACE
           int windowWidth = 800;
           int windowHeight = 600;
           float worldMouseX = (mouseX - windowWidth / 2.0f) + cameraX;
```

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

            float offset = 65.0f * 0.6f;
            float bulletX = cameraX + cos(playerAngle) * offset;
            float bulletY = cameraY + sin(playerAngle) * offset;

                recoilTimer = 5;  // o 6 frames de retroceso
            shootBullet(bulletX, bulletY, worldMouseX, worldMouseY);
            playDisparo();
        }
        if (key == 27) { // Tecla ESC
    enPausa = !enPausa;

    // ====== Generar enemigos con teclas (modo prueba) ======
    if (key == 'n') {
        addEnemy(cameraX + 100, cameraY, ENEMY_BASIC);
    } else if (key == 'm') {
        addEnemy(cameraX + 100, cameraY, ENEMY_SNIPER);
    } else if (key == 'b') {
        addEnemy(cameraX + 100, cameraY, ENEMY_FAST);
    }


    return;
        }

    return; // No hacer nada más
}

void procesarTeclaUp(unsigned char key, int x, int y) {
    keys[key] = false;
```

```
}

void procesarTeclaEspecial(int key, int x, int y) {
    if (gameOverActivo) {
        if (key == GLUT_KEY_UP) {
            opcionGameOver = (opcionGameOver - 1 + 2) % 2;
        } else if (key == GLUT_KEY_DOWN) {
            opcionGameOver = (opcionGameOver + 1) % 2;
        }
    }

    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 (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;  // Tamaño arma * escala
```

```
        float bulletX = cameraX + cos(playerAngle) * offset;
        float bulletY = cameraY + sin(playerAngle) * offset;

        recoilTimer = 5;  // o 6 frames de retroceso
        shootBullet(bulletX, bulletY, worldMouseX, worldMouseY);
        playDisparo();
    }
}

void procesarMouseMove(int x, int y) {
    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);
}
```

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);
```

#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"

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;

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) {


        if (enPausa && !enFade) {
    glutPostRedisplay(); // Redibujar el menú aunque esté pausado
    glutTimerFunc(16, timer, 0);
    return;
        }

        if (reinicioPendiente && fadeAlpha >= 1.0f) {
    ReiniciarJuego();
    reinicioPendiente = false; // ? ya no hay reinicio pendiente
    estadoFade = FADE_IN;
        }
        else if (cambioNivelPendiente && fadeAlpha >= 1.0f) {
    nivelActual++;
    if (nivelActual > TOTAL_NIVELES) nivelActual = 1; // Opcional: volver al nivel 1

    std::string nombreMapa = "map" + std::to_string(nivelActual) + ".txt";
    loadMap(nombreMapa.c_str());
```

```
    initEnemiesFromMap();

    //cameraX = POS_INICIAL_X;
    //cameraY = POS_INICIAL_Y;
    playerHealth = 3;

    bullets.clear();
    enemyBullets.clear();

    estadoFade = FADE_IN;
    cambioNivelPendiente = false;
}



        if (enFade) {
    if (estadoFade == FADE_OUT) {
        fadeAlpha += 0.02f;
        if (fadeAlpha >= 1.0f) {
                        fadeAlpha = 1.0f;
                        estadoFade = FADE_IN;
                    }

    } else if (estadoFade == FADE_IN) {
        fadeAlpha -= 0.02f;
        if (fadeAlpha <= 0.0f) {
            fadeAlpha = 0.0f;
            enFade = false;
            estadoFade = NINGUNO;
            enPausa = false; // Salimos de la pausa
        }
```

```
    }
}
        if (!gameOverActivo && playerHealth <= 0) {
            gameOverActivo = true;
            gameOverAlpha = 0.0f;
            gameOverTimer = 0;
            opcionGameOver = 0;
            Mix_PauseMusic(); // ?? Pausar música al morir
        }

        if (recoilTimer > 0) {
    recoilTimer--;
        }
    float speed = playerSpeed;
    float nextX = cameraX;
    float nextY = cameraY;
    //std::cout << "cameraX: " << cameraX << ", cameraY: " << cameraY;

    if (keys['w']) nextY += speed;
    if (keys['s']) nextY -= speed;
    if (keys['a']) nextX -= speed;
    if (keys['d']) nextX += speed;

    // Verificar colisiones con el nuevo sistema
    if (canMoveTo(nextX, cameraY)) cameraX = nextX;
    if (canMoveTo(cameraX, nextY)) cameraY = nextY;

    if (invulnerableTimer > 0) {
    invulnerableTimer--;
        }
```

```
            // Lógica del 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) { // ~3 segundos de espera
         tutorialEstado = 2;
      }
   } else if (tutorialEstado == 2) { // FADE OUT
      tutorialAlpha -= 0.02f;
      if (tutorialAlpha <= 0.0f) {
         tutorialAlpha = 0.0f;
         tutorialPaso++;
         tutorialEstado = 0; // Empezar FADE IN del siguiente mensaje
      }
   }
}

if (!enFade && !cambioNivelPendiente && isOnPortal(cameraX, cameraY)) {
   cambioNivelPendiente = true;
   enFade = true;
   estadoFade = FADE_OUT;
   fadeAlpha = 0.0f;
}
```

```
    updateBullets();
    updateEnemyBullets();
    updateEnemies(cameraX, cameraY);

    if (gameOverActivo && !reinicioPendiente) {
    glutPostRedisplay();
    glutTimerFunc(16, timer, 0);
    return;
            }


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

void display() {
    glClear(GL_COLOR_BUFFER_BIT);

    // --- 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
    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 ---
    drawLoadingLevelText();
    drawFadeOverlay();
    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);

        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;
        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"

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";
    }
}


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.#

#.@..#.....#.....#

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

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

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

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

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

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

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

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

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

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

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

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;

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;
    }

    musicaFondo = Mix_LoadMUS("music/Contra III_song.MP3");
    if (!musicaFondo) {
        std::cout << "Error al cargar música de fondo: " << Mix_GetError() << std::endl;
        return false;
    }
```

```
    Mix_VolumeMusic(MIX_MAX_VOLUME / 4); // Volumen bajo
    Mix_VolumeChunk(sonidoDisparo, 14); // 48 = volumen medio-bajo
    return true;
}

void playDisparo() {
    Mix_PlayChannel(-1, sonidoDisparo, 0);
}

void playMusic() {
    if (musicaFondo)
        Mix_PlayMusic(musicaFondo, -1); // -1 = bucle infinito
}

void cleanSounds() {
    Mix_FreeChunk(sonidoDisparo);
    Mix_FreeMusic(musicaFondo);
    Mix_CloseAudio();
    SDL_Quit();
}
```

Sound.h:

```
#ifndef SOUND_H
#define SOUND_H

bool initSounds();        // Inicia SDL_mixer y carga todo
void playDisparo();       // Reproduce el sonido del disparo
void playMusic();         // Reproduce la música de fondo
void cleanSounds();       // Libera recursos

#endif
```