Informe de desarrollo de Pacman en C-AlgoritmosI

Tags: #Algoritmos_I #Proyecto

Date: 2024-11-19

Introducción

- Este informe documenta la creación de una versión simplificada del clásico juego Pac-Man, implementado en lenguaje C, y detalla las decisiones técnicas, desafíos enfrentados y aprendizajes obtenidos a lo largo del proyecto.

Propósito del Informe

Estas metas estructuran el contenido de este informe y guían su análisis:

1. Describir el proceso de desarrollo, destacando los principales retos técnicos y las soluciones adoptadas.
2. Documentar las herramientas y librerías utilizadas, junto con las fuentes de referencia consultadas.
3. Evaluar el impacto del uso de inteligencia artificial (IA) en aspectos específicos del proyecto.
4. Extraer conclusiones sobre los aprendizajes obtenidos durante la realización del trabajo.

Alcance del Proyecto

El objetivo del proyecto fue crear una versión funcional de Pac-Man que cumpliera con las siguientes características esenciales:

- Control del jugador sobre un personaje que recorre un laberinto.
- Recolección de puntos para progresar en el juego.
- Inteligencia artificial básica para los fantasmas que persiguen al jugador.
- Sistema de puntuación dinámico.
- Incremento gradual de dificultad mediante niveles progresivos.

Restricciones: Por limitaciones de tiempo y recursos, se priorizó una implementación básica y funcional, omitiendo algunas características avanzadas del juego original, como los *Power Pellets* y los comportamientos complejos de los fantasmas.

Análisis del Juego Original

Resumen de Historia del Desarrollo de PAC-MAN

lanzado en 1980 por Namco, no solo marcó una época en el ámbito de los videojuegos, sino que también introdujo varias mecánicas y algoritmos que aún hoy son objeto de estudio.

Mecánicas del Juego

Objetivos y reglas básicas:

- Pac-Man debe navegar por un laberinto, consumiendo puntos (pellets) mientras evita ser capturado por los fantasmas.
- Al consumir los Power Pellets (puntos grandes), los fantasmas se vuelven vulnerables temporalmente y pueden ser devorados por Pac-Man.
- El juego se desarrolla en niveles progresivos, donde aumenta la dificultad y la velocidad de los fantasmas.

Laberinto:

- El laberinto está compuesto por una grilla de 28x31 celdas.
- Hay caminos y obstáculos fijos, con túneles en los extremos del laberinto que permiten a Pac-Man y a los fantasmas teletransportarse al lado opuesto.
- Existen ciertas áreas, como la "casa de los fantasmas", donde los fantasmas reaparecen tras ser devorados.

Movimiento y Lógica de los Fantasmas

Una de las características más emblemáticas de Pac-Man es la IA de los fantasmas. Cada uno de los cuatro fantasmas tiene un comportamiento distinto, definido por un conjunto de reglas y patrones. Sus nombres en inglés son Blinky, Pinky, Inky y Clyde, y se comportan de la siguiente forma:

Fantasmas y sus estrategias:

1. **Blinky (Rojo) - El Perseguidor:**
   - Blinky sigue directamente a Pac-Man. Su objetivo es ir siempre hacia la posición actual de Pac-Man.
   - Tiene un comportamiento más agresivo a medida que Pac-Man come más puntos.
   - Cuando quedan pocos puntos, entra en modo "Cruise Elroy", aumentando su velocidad.
2. **Pinky (Rosa) - El Emboscador:**
   - Pinky intenta adelantarse a Pac-Man para anticipar su movimiento.

- Su objetivo es la celda que se encuentra **cuatro pasos** delante de la dirección en la que Pac-Man se está moviendo.
- Sin embargo, tiene un error en su lógica cuando Pac-Man se mueve hacia arriba, apuntando 4 casillas delante y 4 casillas a la izquierda.

3. **Inky (Azul) - El Inseguro:**
   - Inky combina la posición de Blinky y la posición de Pac-Man para determinar su movimiento.
   - Su objetivo es una celda que se encuentra en una posición relativa al punto medio entre Blinky y un punto que está dos pasos por delante de Pac-Man.
   - Es el fantasma más impredecible debido a la combinación de dos referencias.
4. **Clyde (Naranja) - El Aleatorio:**
   - Clyde se comporta de dos maneras: si está lejos de Pac-Man, intenta perseguirlo, pero si está cerca (a menos de 8 casillas), se mueve hacia la esquina inferior izquierda del mapa.
   - Su comportamiento es una mezcla de persecución y huida, lo que lo hace menos predecible.

Patrones de Movimiento de los Fantasmas:

- Los fantasmas alternan entre tres modos:
  - **Scatter Mode (Dispersión):** Los fantasmas se mueven hacia sus esquinas asignadas en el laberinto.
  - **Chase Mode (Persecución):** Los fantasmas persiguen activamente a Pac-Man según su estrategia individual.
  - **Frightened Mode (Asustados):** Activado por los Power Pellets. Los fantasmas huyen en direcciones aleatorias y pueden ser devorados.

Algoritmos Utilizados en Pac-Man

Algoritmo de Búsqueda (Pathfinding):

- Los fantasmas utilizan una variación del algoritmo de **Búsqueda en Profundidad (DFS)** o **Búsqueda en Amplitud (BFS)** para determinar su camino hacia el objetivo.
- El algoritmo más similar aplicado en este contexto es el **Algoritmo de A Estrella**, aunque no es implementado de forma explícita en el juego original. El Algoritmo de A* calcula el camino más corto basado en un coste (distancia) hacia el objetivo, lo que podría inspirar la lógica de movimiento predictivo de los fantasmas.

Algoritmo de Detección de Colisiones:

- Se utiliza para determinar si Pac-Man ha tocado a un fantasma o ha pasado por una celda con un punto.
- Implementado mediante el análisis de la posición en la grilla, verificando la superposición de las celdas.

Técnicas de Temporización (Timers):

- Pac-Man usa temporizadores para alternar entre los modos de los fantasmas (Scatter y Chase).
- Estos temporizadores permiten cambiar el comportamiento de los fantasmas para evitar que el jugador anticipe un único patrón, añadiendo variedad al juego.

Conclusion de la investigación

Al conocer las mecánicas del juego y disponiendo de un tiempo limitado para el desarrollo se llego a lo siguiente:

- No se introducirá la mecánica de distintas estrategias de fantasmas.
- No se introducirá los patrones de movimiento, solo quedara el Chase Mode (Persecución)
- No Existirán los Power Pellets (puntos grandes). Por lo que en ningún momento se comerán fantasmas

Estas decisiones están hechas con el objetivo de poder terminar el juego en un tiempo estimado, de ser necesario y tener mas tiempo se podrían haber agregado algunas de estas mecánicas.

Las mecánicas que se intentaron alcanzar en el desarrollo fueron:

- Persecución de los fantasmas
- movilidad de Pacman a través del laberinto
- Recolección de puntos.
- Niveles con cambios de dificultad.

Primera Etapa: Prototipado y Pruebas Iniciales

3.1 Elección del Renderizado Inicial

Al iniciar el proyecto se propuso renderizar el Pacman para jugarlo en la terminal y que solo funcione con texto, de esta elección surgió usar la librería `ncurses` y se trabajo gran parte del código con esto en mente.

3.2 Implementación Inicial(Mapa y movimiento de Pacman)

Primer Código

En la primera implementación se creo una lógica de renderizado de mapa en consola que constaba de una variable que funcionaba como una matriz que tenia los siguientes elementos:

- "#" para los obstáculos(casilla no transitable).
- " " para el espacio transitable(casilla transitable).
  La variable se veía de la siguiente forma y representaba el mapa:

```
char maze[10][10] = {
    { '#', '#', '#', '#', '#', '#', '#', '#', '#', '#' },
    { '#', ' ', ' ', ' ', '#', ' ', ' ', ' ', ' ', '#' },
    { '#', ' ', '#', ' ', '#', ' ', '#', '#', ' ', '#' },
    { '#', ' ', '#', ' ', ' ', ' ', ' ', '#', ' ', '#' },
    { '#', ' ', '#', '#', '#', '#', ' ', '#', ' ', '#' },
    { '#', ' ', ' ', ' ', ' ', '#', ' ', ' ', ' ', '#' },
    { '#', '#', '#', ' ', '#', '#', '#', '#', ' ', '#' },
    { '#', ' ', ' ', ' ', ' ', ' ', ' ', '#', ' ', '#' },
    { '#', ' ', '#', '#', '#', '#', ' ', ' ', ' ', '#' },
    { '#', '#', '#', '#', '#', '#', '#', '#', '#', '#' }

};
```

Junto a esta se desarrollo una procedimiento que, usando un bucles dibujaba el mapa en la terminal, de la siguiente forma:

```
void draw_maze() {
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 10; j++) {
            mvaddch(i, j, maze[i][j]);
        }
    }
    mvaddch(pacman_y, pacman_x, PACMAN);
}
```

Tambien se añadió la primera lógica de movimiento de Pacman en donde se usaba una variable que guardaba un evento(como el presionar una tecla) y en base a que tecla presionabas llamaba al procedimiento para mover a Pacman.
El procedimiento que movía a Pacman se veía de la siguiente manera:

```
void move_pacman(int new_y, int new_x) {
    if (maze[new_y][new_x] != WALL) { // Solo se mueve si no es una pared
        mvaddch(pacman_y, pacman_x, EMPTY); // Borra posición anterior
```

```
            pacman_y = new_y;
            pacman_x = new_x;
            mvaddch(pacman_y, pacman_x, PACMAN); // Dibuja nueva posición
        }

    }
```

El principal problema que se saco de esta implementación es que el movimiento de Pacman, si bien funcionaba, no era muy fluido ya que cada vez quisieras que cambie de posición deberías presionar la tecla una a una, mientras que en el juego original se mantenía yendo hacia la dirección indicada hasta que la cambies.
Tambien se pensó que seria mejor idea guardar el mapa en un documento aparte y luego traerlo con una función para así tener mas flexibilidad a la hora de cambiar o crear otro.

Segundo Código

Basado en los cambios que se buscaban incluir a partir de la primera implementación se realizo un nuevo código buscando satisfacer lo siguiente:

- Movimiento de Pacman mas fluido.
- Cargar el mapa desde un archivo.

Tambien se añadió el primer tipo de estructura de dato llamada `Posicion` que sirve para guardar el valor de la posición de un objeto en termino de eje X y eje Y.
El código final Posee una procedimiento de carga de mapa que recibe el nombre del archivo donde se almacena la forma del mapa(Existen variables donde se debe especificar el ancho y largo del mapa):

```
// Función para cargar el mapa desde un archivo
int cargar_mapa(const char *filename) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        perror("No se pudo abrir el archivo de mapa");
        return -1;
    }

    for (int i = 0; i < HEIGHT; i++) {
        if (fgets(maze[i], WIDTH + 2, file) == NULL) {  // +2 para leer el salto
de línea
            perror("Error al leer el mapa");
            fclose(file);
            return -1;
        }
    }
```

```
    fclose(file);
    return 0;
}
```

Tambien se añadió una pausa para que el bucle principal se repitiera de forma constante y con la misma lógica del anterior código de eventos se crea una variable que donde se guarda la dirección hacia donde va Pacman y se mantiene hasta que se detecte un evento que la cambie, el código final quedo de la siguiente manera:

```
#include <ncurses.h>
#include <stdlib.h>
#include <stdio.h>

#define WIDTH 22
#define HEIGHT 13
#define OBSTACLE '#'
#define PATH ' '
#define PACMAN 'P'

typedef struct {
    int x, y;
} Posicion;

char maze[HEIGHT][WIDTH + 1];

// Función para cargar el mapa desde un archivo
int cargar_mapa(const char *filename) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        perror("No se pudo abrir el archivo de mapa");
        return -1;
    }

    for (int i = 0; i < HEIGHT; i++) {
        if (fgets(maze[i], WIDTH + 2, file) == NULL) {  // +2 para leer el salto
de línea
            perror("Error al leer el mapa");
            fclose(file);
            return -1;
        }
    }

    fclose(file);
    return 0;
```

```
}

// Direcciones de movimiento (arriba, abajo, izquierda, derecha)
enum Direccion { ARRIBA, ABAJO, IZQUIERDA, DERECHA, NINGUNA };
Posicion direcciones[] = {{0, -1}, {0, 1}, {-1, 0}, {1, 0}};

// Función para verificar si una posición es transitable
int es_posicion_valida(Posicion pos) {
    return (pos.x >= 0 && pos.x < WIDTH && pos.y >= 0 && pos.y < HEIGHT &&
maze[pos.y][pos.x] != OBSTACLE);
}

// Dibujar el laberinto y la posición de Pacman
void draw_game(Posicion pacman) {
    for (int i = 0; i < HEIGHT; i++) {
        for (int j = 0; j < WIDTH; j++) {
            mvaddch(i, j, maze[i][j]);
        }
    }
    mvaddch(pacman.y, pacman.x, PACMAN);
}

int main() {
    Posicion pacman = {1, 1};  // Posición inicial de Pacman
    enum Direccion direccion_actual = DERECHA;  // Dirección inicial

    // Cargar el mapa desde el archivo
    if (cargar_mapa("mapa.txt") == -1) {
        return 1;
    }

    initscr();
    noecho();
    curs_set(FALSE);
    keypad(stdscr, TRUE);
    nodelay(stdscr, TRUE);

    int ch;
    while (1) {
        clear();
        draw_game(pacman);

        // Leer tecla y actualizar la dirección si se presiona una
        ch = getch();
        switch (ch) {
            case KEY_UP:    direccion_actual = ARRIBA; break;
```

```
            case KEY_DOWN:  direccion_actual = ABAJO; break;
            case KEY_LEFT:  direccion_actual = IZQUIERDA; break;
            case KEY_RIGHT: direccion_actual = DERECHA; break;
            case 'q':       endwin(); exit(0);  // Salir del juego
        }

        // Calcular la nueva posición de Pacman en la dirección actual
        Posicion nueva_posicion = {pacman.x + direcciones[direccion_actual].x,
pacman.y + direcciones[direccion_actual].y};

        // Mover Pacman si la nueva posición es válida
        if (es_posicion_valida(nueva_posicion)) {
            pacman = nueva_posicion;
        }

        refresh();
        napms(100);  // Pausa para que el movimiento sea visible
    }

    endwin();
    return 0;
}
```

Desarrollo de la Inteligencia Artificial para los Fantasmas

Investigación

El análisis de la IA en Pac-Man reveló que cada fantasma tiene un comportamiento único basado en patrones de movimiento. Aunque en el juego original se emplean técnicas específicas, el desarrollo de este proyecto adoptó el algoritmo *A* (A estrella)*, una técnica ampliamente utilizada para encontrar caminos óptimos en grafos y mapas. Este enfoque ofrece una solución flexible y eficiente para determinar la trayectoria de los fantasmas hacia el jugador.

¿Qué es el algoritmo A* y cómo funciona?

El algoritmo A *es una técnica de búsqueda heurística ampliamente utilizada para encontrar el camino más corto desde un punto de inicio a un punto objetivo en un espacio de búsqueda, como un mapa o grafo. A* combina las ventajas de la búsqueda por costo uniforme (Dijkstra) y la búsqueda heurística (Greedy Best-First Search). Su principal característica es que evalúa los nodos basándose en una función de costo:

$$f(n) = g(n) + h(n)$$

- $g(n)$: Representa el costo acumulado para llegar al nodo n desde el nodo inicial.
- $h(n)$: Es una estimación heurística del costo restante para llegar al objetivo desde n.
- $f(n)$: Es el costo total estimado si se pasa por el nodo n.

El algoritmo A* utiliza una estructura de datos llamada *lista abierta* para rastrear nodos que necesitan ser explorados y una *lista cerrada* para los nodos ya evaluados.

Pasos principales del algoritmo A*:

1. **Inicialización:**
   - Coloca el nodo inicial en la lista abierta.
   - Calcula los valores de $g(n), h(n), f(n)$ para el nodo inicial.
2. **Búsqueda iterativa:**
   - Selecciona el nodo con el menor $f(n)$ de la lista abierta.
   - Si el nodo seleccionado es el objetivo, termina la búsqueda y reconstruye el camino.
   - De lo contrario:
     - Mueve el nodo actual a la lista cerrada.
     - Explora sus vecinos (nodos adyacentes).
     - Si un vecino no está en la lista cerrada y mejora el camino actual, lo agrega o actualiza en la lista abierta.
3. **Terminar:**
   - Si la lista abierta se vacía sin encontrar el objetivo, significa que no hay camino posible.

Características clave:

- **Admisibilidad:** La heurística $h(n)$ debe ser optimista, es decir, nunca sobrestimar el costo real.
- **Completitud:** A* siempre encuentra un camino si existe uno.
- **Optimalidad:** Si $h(n)$ es admisible y consistente, A* garantiza el camino más corto.

Implementación del algoritmo A* en el código

El código implementa el algoritmo A* para encontrar un camino en un mapa bidimensional representado por una matriz. Aquí se explica cómo se implementaron los elementos clave del algoritmo:

1. Representación del mapa

- El mapa se carga desde un archivo usando la función `cargar_mapa`. Cada celda del mapa puede ser:
  - `OBSTACLE` (`#`): Celda inaccesible.

- `PLAYER (C)`: Posición inicial.
- `GOAL (G)`: Objetivo.
- `PATH (.)`: Camino recorrido.

2. Nodos y cálculo de costos

- Cada nodo se define mediante una estructura `Node` que contiene:
  - Posición actual `pos` (coordenadas).
  - Costos g, h, f.
  - Puntero al nodo padre para reconstruir el camino.
- La función `manhattan_distance` calcula $h(n)$ usando la distancia Manhattan, adecuada para movimientos ortogonales.

3. Listas abierta y cerrada

- La lista abierta es un array dinámico de nodos (`open_list`), donde se almacenan nodos candidatos para exploración.
- La lista cerrada es una matriz booleana (`closed_list`) que marca las celdas ya visitadas.

4. Proceso de búsqueda

1. El nodo inicial se agrega a la lista abierta con sus costos iniciales calculados.
2. En cada iteración:
   - Se selecciona el nodo con el menor $f(n)$.
   - Si es el nodo objetivo, se reconstruye el camino utilizando los nodos padre.
   - De lo contrario, se exploran sus vecinos (nodos adyacentes).
3. Los vecinos válidos se evalúan y, si tienen un mejor costo, se agregan a la lista abierta.

5. Reconstrucción del camino

- Una vez alcanzado el objetivo, el camino se reconstruye desde el nodo objetivo hasta el nodo inicial siguiendo la cadena de nodos padre.

6. Movimiento del personaje

- La función `move_character` simula el movimiento del personaje a lo largo del camino encontrado, actualizando y mostrando el mapa en cada paso.

Flujo principal

- En `main`, se inicializan las posiciones de inicio y objetivo.

- El algoritmo A* busca el camino y, si lo encuentra, se anima el movimiento del personaje sobre el mapa.

Código A* que Devuelve un Array con el Camino Mas Corto

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define FILAS 22
#define COLUMNAS 13
#define OBSTACULO '#'
#define LONGITUD_MAXIMA_CAMINO 100

// Estructura para las posiciones en el mapa
typedef struct {
    int x, y;
} Posicion;

// Estructura de nodo para A*
typedef struct {
    Posicion pos;
    int costo_g;
    int costo_h;
    int costo_f;
} Nodo;

// Mapa de ejemplo
char mapa[FILAS][COLUMNAS];

int cargar_mapa(const char *filename) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        perror("No se pudo abrir el archivo de mapa");
        return -1;
    }

    for (int i = 0; i < FILAS; i++) {
        if (fgets(mapa[i], COLUMNAS + 2, file) == NULL) {  // +2 para leer el
salto de línea
            perror("Error al leer el mapa");
            fclose(file);
            return -1;
        }
    }
```

```
    fclose(file);
    return 0;
}

// Direcciones de movimiento (arriba, abajo, izquierda, derecha)
Posicion direcciones[] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

// Función para calcular la distancia de Manhattan (heurística)
int distancia_manhattan(Posicion a, Posicion b) {
    return abs(a.x - b.x) + abs(a.y - b.y);
}

// Función para verificar si una posición está dentro de los límites y es
transitable
int es_posicion_valida(Posicion pos, char mapa_actual[FILAS][COLUMNAS]) {
    return (pos.x >= 0 && pos.x < FILAS && pos.y >= 0 && pos.y < COLUMNAS &&
mapa_actual[pos.x][pos.y] != OBSTACULO);
}

// Algoritmo A* encapsulado en una sola función
int a_estrella(Posicion inicio, Posicion objetivo, Posicion
camino[LONGITUD_MAXIMA_CAMINO]) {
    Nodo lista_abierta[FILAS * COLUMNAS];
    int contador_abierta = 0;
    int lista_cerrada[FILAS][COLUMNAS] = {0};
    int longitud_camino = 0;
    Posicion recorrido[FILAS][COLUMNAS] = {{-1, -1}};

    Nodo nodo_inicio = {inicio, 0, distancia_manhattan(inicio, objetivo),
distancia_manhattan(inicio, objetivo)};
    lista_abierta[contador_abierta++] = nodo_inicio;

    while (contador_abierta > 0) {
        // Buscar el nodo con el menor costo_f en lista_abierta
        int indice_min = 0;
        for (int i = 1; i < contador_abierta; i++) {
            if (lista_abierta[i].costo_f < lista_abierta[indice_min].costo_f) {
                indice_min = i;
            }
        }

        Nodo nodo_actual = lista_abierta[indice_min];
        lista_abierta[indice_min] = lista_abierta[--contador_abierta];
        lista_cerrada[nodo_actual.pos.x][nodo_actual.pos.y] = 1;

        if (nodo_actual.pos.x == objetivo.x && nodo_actual.pos.y == objetivo.y)
```

```
{
            Posicion pos = objetivo;
            while (pos.x != inicio.x || pos.y != inicio.y) {
                camino[longitud_camino++] = pos;
                pos = recorrido[pos.x][pos.y];
            }
            camino[longitud_camino++] = inicio;
            // Invertir el camino para obtenerlo en el orden correcto
            for (int i = 0; i < longitud_camino / 2; i++) {
                Posicion temp = camino[i];
                camino[i] = camino[longitud_camino - i - 1];
                camino[longitud_camino - i - 1] = temp;
            }
            return longitud_camino;
        }

        // Expandir vecinos
        for (int i = 0; i < 4; i++) {
            Posicion vecino_pos = {nodo_actual.pos.x + direcciones[i].x,
nodo_actual.pos.y + direcciones[i].y};
            if (es_posicion_valida(vecino_pos, mapa) &&
!lista_cerrada[vecino_pos.x][vecino_pos.y]) {
                int costo_g = nodo_actual.costo_g + 1;
                int costo_h = distancia_manhattan(vecino_pos, objetivo);
                int costo_f = costo_g + costo_h;

                int en_lista = 0;
                for (int j = 0; j < contador_abierta; j++) {
                    if (lista_abierta[j].pos.x == vecino_pos.x &&
lista_abierta[j].pos.y == vecino_pos.y) {
                        en_lista = 1;
                        if (costo_f < lista_abierta[j].costo_f) {
                            lista_abierta[j].costo_g = costo_g;
                            lista_abierta[j].costo_h = costo_h;
                            lista_abierta[j].costo_f = costo_f;
                            recorrido[vecino_pos.x][vecino_pos.y] =
nodo_actual.pos;
                        }
                        break;
                    }
                }

                if (!en_lista) {
                    Nodo nodo_vecino = {vecino_pos, costo_g, costo_h, costo_f};
                    lista_abierta[contador_abierta++] = nodo_vecino;
                    recorrido[vecino_pos.x][vecino_pos.y] = nodo_actual.pos;
```

```
                }
            }
        }
    }

    return -1;  // No se encontró un camino
}

int main() {
    Posicion inicio = {4, 0};
    Posicion objetivo = {1, 9};
    Posicion camino[LONGITUD_MAXIMA_CAMINO];
    int longitud_camino = a_estrella(inicio, objetivo, camino);
    // Cargar el mapa desde el archivo
    if (cargar_mapa("mapa.txt") == -1) {
        return 1;
    }


    if (longitud_camino != -1) {
        printf("Camino encontrado:\n");
        for (int i = 0; i < longitud_camino; i++) {
            printf("Paso %d: (%d, %d)\n", i, camino[i].x, camino[i].y);
        }
    } else {
        printf("No se encontró un camino.\n");
    }

    return 0;
}
```

Código A* que Recorre el Mapa

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <unistd.h>  // Para usleep (solo en UNIX)

#define COLUMNAS 22
#define FILAS 13
#define OBSTACLE '#'
#define PLAYER 'C'
#define GOAL 'G'
#define PATH '.'
```

```c
typedef struct {
    int x, y;
} Position;

typedef struct Node {
    Position pos;
    int g_cost;
    int h_cost;
    int f_cost;
    struct Node *parent;
} Node;

char mapa[FILAS][COLUMNAS] ;
int cargar_mapa(const char *filename) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        perror("No se pudo abrir el archivo de mapa");
        return -1;
    }

    for (int i = 0; i < FILAS; i++) {
        if (fgets(mapa[i], COLUMNAS + 2, file) == NULL) {  // +2 para leer el
salto de línea
            perror("Error al leer el mapa");
            fclose(file);
            return -1;
        }
    }

    fclose(file);
    return 0;
}

int manhattan_distance(Position a, Position b) {
    return abs(a.x - b.x) + abs(a.y - b.y);
}

int is_valid_position(Position pos) {
    return (pos.x >= 0 && pos.x < FILAS && pos.y >= 0 && pos.y < COLUMNAS &&
mapa[pos.x][pos.y] != OBSTACLE);
}

Position directions[] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

void print_map() {
```

```
    for (int i = 0; i < FILAS; i++) {
        for (int j = 0; j < COLUMNAS; j++) {
            printf("%c ", mapa[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}

Node* a_star(Position start, Position goal, Position path[], int *path_length) {
    Node *open_list[FILAS * COLUMNAS];
    int open_count = 0;
    int closed_list[FILAS][COLUMNAS] = {0};

    Node *start_node = (Node *)malloc(sizeof(Node));
    start_node->pos = start;
    start_node->g_cost = 0;
    start_node->h_cost = manhattan_distance(start, goal);
    start_node->f_cost = start_node->g_cost + start_node->h_cost;
    start_node->parent = NULL;

    open_list[open_count++] = start_node;

    while (open_count > 0) {
        int min_index = 0;
        for (int i = 1; i < open_count; i++) {
            if (open_list[i]->f_cost < open_list[min_index]->f_cost) {
                min_index = i;
            }
        }

        Node *current_node = open_list[min_index];
        open_list[min_index] = open_list[--open_count];
        closed_list[current_node->pos.x][current_node->pos.y] = 1;

        if (current_node->pos.x == goal.x && current_node->pos.y == goal.y) {
            *path_length = 0;
            Node *path_node = current_node;
            while (path_node != NULL) {
                path[(*path_length)++] = path_node->pos;
                path_node = path_node->parent;
            }
            return current_node;
        }

        for (int i = 0; i < 4; i++) {
```

```
            Position neighbor_pos = {current_node->pos.x + directions[i].x,
current_node->pos.y + directions[i].y};
            if (is_valid_position(neighbor_pos) && !closed_list[neighbor_pos.x]
[neighbor_pos.y]) {
                int g_cost = current_node->g_cost + 1;
                int h_cost = manhattan_distance(neighbor_pos, goal);
                int f_cost = g_cost + h_cost;

                Node *neighbor_node = (Node *)malloc(sizeof(Node));
                neighbor_node->pos = neighbor_pos;
                neighbor_node->g_cost = g_cost;
                neighbor_node->h_cost = h_cost;
                neighbor_node->f_cost = f_cost;
                neighbor_node->parent = current_node;

                open_list[open_count++] = neighbor_node;
            }
        }
    }
    printf("No se encontró un camino.\n");
    return NULL;
}

void move_character(Position path[], int path_length) {
    for (int i = path_length - 1; i >= 0; i--) {
        // Limpiar la posición anterior
        if (i < path_length - 1) {
            mapa[path[i + 1].x][path[i + 1].y] = PATH;
        }
        // Colocar el personaje en la nueva posición
        mapa[path[i].x][path[i].y] = PLAYER;
        print_map();
        usleep(300000);  // Pausa de 0.3 segundos para observar el movimiento
        system("CLS");
    }
}

int main() {
    Position start = {1, 1};
    Position goal = {9, 20};
    Position path[FILAS * COLUMNAS];
    int path_length;
    // Cargar el mapa desde el archivo
    if (cargar_mapa("mapa.txt") == -1) {
        return 1;
    }
```

```
    Node *end_node = a_star(start, goal, path, &path_length);

    if (end_node != NULL) {
        mapa[start.x][start.y] = PLAYER;
        mapa[goal.x][goal.y] = GOAL;
        print_map();
        printf("Recorriendo el camino encontrado:\n");
        move_character(path, path_length);
    }

    return 0;
}
```

Integración de Mecánicas Básicas

Integrar Movilidad de Pacman y Fantasmas

Con el código funcional para el movimiento de Pac-Man en el mapa y la implementación del algoritmo A* para los fantasmas, se trabajó en la integración de ambos elementos en un solo módulo. Para mejorar la jugabilidad, se añadió un retraso en la actualización de la posición de los fantasmas, reduciendo su velocidad respecto a la de Pac-Man. Además, se implementó un verificador que detecta si un fantasma ocupa la misma posición que Pac-Man, lo que provoca el fin del juego.

Código con los Cambios Implementados

```
#include <ncurses.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define COLUMNAS 22
#define FILAS 13
#define OBSTACLE '#'
#define PATH ' '
#define PACMAN 'P'
#define GHOST 'G'
#define LONGITUD_MAXIMA_CAMINO 100

typedef struct {
    int x, y;
```

```c
} Posicion;

bool bandera;
char maze[FILAS][COLUMNAS + 1];

// Direcciones de movimiento (arriba, abajo, izquierda, derecha)
enum Direccion { ARRIBA, ABAJO, IZQUIERDA, DERECHA, NINGUNA };
Posicion direcciones[] = {{0, -1}, {0, 1}, {-1, 0}, {1, 0}};

// Función para cargar el mapa desde un archivo
int cargar_mapa(const char *filename) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        perror("No se pudo abrir el archivo de mapa");
        return -1;
    }

    for (int i = 0; i < FILAS; i++) {
        if (fgets(maze[i], COLUMNAS + 2, file) == NULL) {  // +2 para leer el
salto de línea
            perror("Error al leer el mapa");
            fclose(file);
            return -1;
        }
    }

    fclose(file);
    return 0;
}

// Función para calcular la distancia de Manhattan (heurística)
int distancia_manhattan(Posicion a, Posicion b) {
    return abs(a.x - b.x) + abs(a.y - b.y);
}

// Algoritmo A* para encontrar el camino entre el fantasma y Pacman
int a_estrella(Posicion inicio, Posicion objetivo, Posicion
camino[LONGITUD_MAXIMA_CAMINO]) {
    int lista_cerrada[FILAS][COLUMNAS] = {0};
    int longitud_camino = 0;
    Posicion recorrido[FILAS][COLUMNAS] = {{-1, -1}};
    int encontrado = 0;

    typedef struct {
        Posicion pos;
        int g, h, f;
```

```
    } Nodo;

    Nodo lista_abierta[FILAS * COLUMNAS];
    int contador_abierta = 0;

    Nodo nodo_inicio = {inicio, 0, distancia_manhattan(inicio, objetivo),
distancia_manhattan(inicio, objetivo)};
    lista_abierta[contador_abierta++] = nodo_inicio;

    while (contador_abierta > 0 && !encontrado) {
        int indice_min = 0;
        for (int i = 1; i < contador_abierta; i++) {
            if (lista_abierta[i].f < lista_abierta[indice_min].f) {
                indice_min = i;
            }
        }

        Nodo nodo_actual = lista_abierta[indice_min];
        lista_abierta[indice_min] = lista_abierta[--contador_abierta];
        lista_cerrada[nodo_actual.pos.y][nodo_actual.pos.x] = 1;

        if (nodo_actual.pos.x == objetivo.x && nodo_actual.pos.y == objetivo.y)
{
            Posicion pos = objetivo;
            while (pos.x != inicio.x || pos.y != inicio.y) {
                camino[longitud_camino++] = pos;
                pos = recorrido[pos.y][pos.x];
            }
            camino[longitud_camino++] = inicio;
            for (int i = 0; i < longitud_camino / 2; i++) {
                Posicion temp = camino[i];
                camino[i] = camino[longitud_camino - i - 1];
                camino[longitud_camino - i - 1] = temp;
            }
            encontrado = 1;
            break;
        }

        for (int i = 0; i < 4; i++) {
            Posicion vecino = {nodo_actual.pos.x + direcciones[i].x,
nodo_actual.pos.y + direcciones[i].y};
            if (vecino.x >= 0 && vecino.x < COLUMNAS && vecino.y >= 0 &&
vecino.y < FILAS &&
                maze[vecino.y][vecino.x] != OBSTACLE && !lista_cerrada[vecino.y]
[vecino.x]) {
                int g = nodo_actual.g + 1;
```

```
                int h = distancia_manhattan(vecino, objetivo);
                int f = g + h;

                int en_lista = 0;
                for (int j = 0; j < contador_abierta; j++) {
                    if (lista_abierta[j].pos.x == vecino.x &&
lista_abierta[j].pos.y == vecino.y) {
                        en_lista = 1;
                        if (f < lista_abierta[j].f) {
                            lista_abierta[j].g = g;
                            lista_abierta[j].h = h;
                            lista_abierta[j].f = f;
                            recorrido[vecino.y][vecino.x] = nodo_actual.pos;
                        }
                        break;
                    }
                }

                if (!en_lista) {
                    Nodo nodo_vecino = {vecino, g, h, f};
                    lista_abierta[contador_abierta++] = nodo_vecino;
                    recorrido[vecino.y][vecino.x] = nodo_actual.pos;
                }
            }
        }
    }

    return encontrado ? longitud_camino : -1;
}

// Función para verificar si una posición es transitable
int es_posicion_valida(Posicion pos) {
    return (pos.x >= 0 && pos.x < COLUMNAS && pos.y >= 0 && pos.y < FILAS &&
maze[pos.y][pos.x] != OBSTACLE);
}

// Dibujar el laberinto, Pacman y el Fantasma
void draw_game(Posicion pacman, Posicion ghost) {
    for (int i = 0; i < FILAS; i++) {
        for (int j = 0; j < COLUMNAS; j++) {
            mvaddch(i, j, maze[i][j]);
        }
    }
    if ( pacman.y == ghost.y && pacman.x == ghost.x )
    {
        endwin();
```

```
        exit(0);
    }
    mvaddch(pacman.y, pacman.x, PACMAN);
    mvaddch(ghost.y, ghost.x, GHOST);
}

int main() {
    Posicion pacman = {1, 5};
    Posicion ghost = {8, 5};
    enum Direccion direccion_actual = DERECHA;

    if (cargar_mapa("mapa.txt") == -1) {
        return 1;
    }
    initscr();
    noecho();
    curs_set(FALSE);
    keypad(stdscr, TRUE);
    nodelay(stdscr, TRUE);
    int ch;
    Posicion camino[LONGITUD_MAXIMA_CAMINO];
    int longitud_camino, paso = 0;
    while (1) {
        clear();
        draw_game(pacman, ghost);
        ch = getch();
        switch (ch) {
            case KEY_UP:    direccion_actual = ARRIBA; break;
            case KEY_DOWN:  direccion_actual = ABAJO; break;
            case KEY_LEFT:  direccion_actual = IZQUIERDA; break;
            case KEY_RIGHT: direccion_actual = DERECHA; break;
            case 'q':       endwin(); exit(0);
        }
        Posicion nueva_posicion = {pacman.x + direcciones[direccion_actual].x,
pacman.y + direcciones[direccion_actual].y};
        if (es_posicion_valida(nueva_posicion)) {
            pacman = nueva_posicion;
        }

        longitud_camino = a_estrella(ghost, pacman, camino);
        if (bandera)
        {
            if (longitud_camino > 1) {
                ghost = camino[1];
            }
            bandera=false;
```

```
        }
        else
        {
            bandera=true;
        }
        refresh();
        napms(100);
    }
    endwin();
    return 0;
}
```

En este código se puede controlar a Pacman y existe un fantasma persiguiéndote

Recolección de Puntos

La mecánica de puntos fue una de las más sencillas de implementar. Consistió en verificar si la casilla ocupada por Pac-Man contenía un punto; en caso afirmativo, se incrementaba la puntuación global y la casilla se marcaba como vacía. Al cargar el mapa por primera vez, se contabilizaba el número total de puntos disponibles. El juego termina cuando la cantidad de puntos recolectados coincide con el total inicial, indicando que el nivel ha sido completado.

Código con los Cambios Implementados

```
#include <ncurses.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define COLUMNAS 22
#define FILAS 13
#define OBSTACLE '#'
#define PATH ' '
#define PACMAN 'P'
#define GHOST 'G'
#define POINT '.'
#define LONGITUD_MAXIMA_CAMINO 100

typedef struct {
    int x, y;
} Posicion;

bool bandera;
char maze[FILAS][COLUMNAS + 1];
```

```c
int puntaje = 0;
int puntos_totales = 0;

// Direcciones de movimiento (arriba, abajo, izquierda, derecha)
enum Direccion { ARRIBA, ABAJO, IZQUIERDA, DERECHA, NINGUNA };
Posicion direcciones[] = {{0, -1}, {0, 1}, {-1, 0}, {1, 0}};

// Función para cargar el mapa desde un archivo
int cargar_mapa(const char *filename) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        perror("No se pudo abrir el archivo de mapa");
        return -1;
    }

    for (int i = 0; i < FILAS; i++) {
        if (fgets(maze[i], COLUMNAS + 2, file) == NULL) {  // +2 para leer el
salto de línea
            perror("Error al leer el mapa");
            fclose(file);
            return -1;
        }
        for (int j = 0; j < COLUMNAS; j++) {
            if (maze[i][j] == POINT) {
                puntos_totales++;
            }
        }
    }

    fclose(file);
    return 0;
}

// Función para calcular la distancia de Manhattan (heurística)
int distancia_manhattan(Posicion a, Posicion b) {
    return abs(a.x - b.x) + abs(a.y - b.y);
}

// Algoritmo A* para encontrar el camino entre el fantasma y Pacman
int a_estrella(Posicion inicio, Posicion objetivo, Posicion
camino[LONGITUD_MAXIMA_CAMINO]) {
    int lista_cerrada[FILAS][COLUMNAS] = {0};
    int longitud_camino = 0;
    Posicion recorrido[FILAS][COLUMNAS] = {{-1, -1}};
    int encontrado = 0;
```

```
    typedef struct {
        Posicion pos;
        int g, h, f;
    } Nodo;

    Nodo lista_abierta[FILAS * COLUMNAS];
    int contador_abierta = 0;

    Nodo nodo_inicio = {inicio, 0, distancia_manhattan(inicio, objetivo),
distancia_manhattan(inicio, objetivo)};
    lista_abierta[contador_abierta++] = nodo_inicio;

    while (contador_abierta > 0 && !encontrado) {
        int indice_min = 0;
        for (int i = 1; i < contador_abierta; i++) {
            if (lista_abierta[i].f < lista_abierta[indice_min].f) {
                indice_min = i;
            }
        }

        Nodo nodo_actual = lista_abierta[indice_min];
        lista_abierta[indice_min] = lista_abierta[--contador_abierta];
        lista_cerrada[nodo_actual.pos.y][nodo_actual.pos.x] = 1;

        if (nodo_actual.pos.x == objetivo.x && nodo_actual.pos.y == objetivo.y)
{
            Posicion pos = objetivo;
            while (pos.x != inicio.x || pos.y != inicio.y) {
                camino[longitud_camino++] = pos;
                pos = recorrido[pos.y][pos.x];
            }
            camino[longitud_camino++] = inicio;
            for (int i = 0; i < longitud_camino / 2; i++) {
                Posicion temp = camino[i];
                camino[i] = camino[longitud_camino - i - 1];
                camino[longitud_camino - i - 1] = temp;
            }
            encontrado = 1;
            break;
        }

        for (int i = 0; i < 4; i++) {
            Posicion vecino = {nodo_actual.pos.x + direcciones[i].x,
nodo_actual.pos.y + direcciones[i].y};
            if (vecino.x >= 0 && vecino.x < COLUMNAS && vecino.y >= 0 &&
vecino.y < FILAS &&
```

```
                maze[vecino.y][vecino.x] != OBSTACLE && !lista_cerrada[vecino.y]
[vecino.x]) {
                int g = nodo_actual.g + 1;
                int h = distancia_manhattan(vecino, objetivo);
                int f = g + h;

                int en_lista = 0;
                for (int j = 0; j < contador_abierta; j++) {
                    if (lista_abierta[j].pos.x == vecino.x &&
lista_abierta[j].pos.y == vecino.y) {
                        en_lista = 1;
                        if (f < lista_abierta[j].f) {
                            lista_abierta[j].g = g;
                            lista_abierta[j].h = h;
                            lista_abierta[j].f = f;
                            recorrido[vecino.y][vecino.x] = nodo_actual.pos;
                        }
                        break;
                    }
                }

                if (!en_lista) {
                    Nodo nodo_vecino = {vecino, g, h, f};
                    lista_abierta[contador_abierta++] = nodo_vecino;
                    recorrido[vecino.y][vecino.x] = nodo_actual.pos;
                }
            }
        }
    }

    return encontrado ? longitud_camino : -1;
}

// Función para verificar si una posición es transitable
int es_posicion_valida(Posicion pos) {
    return (pos.x >= 0 && pos.x < COLUMNAS && pos.y >= 0 && pos.y < FILAS &&
maze[pos.y][pos.x] != OBSTACLE);
}

// Dibujar el laberinto, Pacman, el Fantasma y el puntaje
void draw_game(Posicion pacman, Posicion ghost) {
    for (int i = 0; i < FILAS; i++) {
        for (int j = 0; j < COLUMNAS; j++) {
            mvaddch(i, j, maze[i][j]);
        }
    }
```

```
    mvprintw(FILAS, 0, "Puntaje: %d", puntaje);

    // Terminar el juego si no quedan puntos
    if (puntos_totales == 0) {
        mvprintw(FILAS + 1, 0, "Ganaste! Presiona 'q' para salir.");
        getch();
        endwin();
        exit(0);
    }

    // Terminar el juego si Pacman y el fantasma se encuentran
    if (pacman.y == ghost.y && pacman.x == ghost.x) {
        mvprintw(FILAS + 1, 0, "Perdiste! Presiona 'q' para salir.");
        getch();
        endwin();
        exit(0);
    }

    mvaddch(pacman.y, pacman.x, PACMAN);
    mvaddch(ghost.y, ghost.x, GHOST);
}

int main() {
    Posicion pacman = {1, 5};
    Posicion ghost = {8, 5};
    enum Direccion direccion_actual = DERECHA;

    if (cargar_mapa("mapaConPuntos.txt") == -1) {
        return 1;
    }

    initscr();
    noecho();
    curs_set(FALSE);
    keypad(stdscr, TRUE);
    nodelay(stdscr, TRUE);

    int ch;
    Posicion camino[LONGITUD_MAXIMA_CAMINO];
    int longitud_camino, paso = 0;

    while (1) {
        clear();
        draw_game(pacman, ghost);
        ch = getch();
```

```
        switch (ch) {
            case KEY_UP:    direccion_actual = ARRIBA; break;
            case KEY_DOWN:  direccion_actual = ABAJO; break;
            case KEY_LEFT:  direccion_actual = IZQUIERDA; break;
            case KEY_RIGHT: direccion_actual = DERECHA; break;
            case 'q':       endwin(); exit(0);
        }
        Posicion nueva_posicion = {pacman.x + direcciones[direccion_actual].x,
pacman.y + direcciones[direccion_actual].y};
        if (es_posicion_valida(nueva_posicion)) {
            if (maze[nueva_posicion.y][nueva_posicion.x] == POINT) {
                puntaje++;
                puntos_totales--;
                maze[nueva_posicion.y][nueva_posicion.x] = PATH;
            }
            pacman = nueva_posicion;
        }
        longitud_camino = a_estrella(ghost, pacman, camino);
        if (bandera) {
            if (longitud_camino > 1) {
                ghost = camino[1];
            }
            bandera = false;
        } else {
            bandera = true;
        }
        refresh();
        napms(100);
    }
    endwin();
    return 0;
}
```

Transición a SDL2

Debido a las limitaciones visuales de `ncurses` y considerando el tiempo restante para completar el proyecto, se decidió migrar la lógica de renderizado a `SDL2`, una librería de `C` más avanzada y versátil para gráficos. Antes de integrar `SDL2` en el proyecto, se realizaron pruebas independientes con el objetivo de familiarizarse con sus funciones principales. Estas pruebas incluyeron tareas básicas como inicializar `SDL2` y renderizar colores estáticos en pantalla. Además, se planificó cuidadosamente la implementación para minimizar los cambios necesarios en el código existente.

Principales cambios realizados:

- Creación de un nuevo procedimiento para inicializar `SDL`.
- Desarrollo de un procedimiento para dibujar rectángulos.
- Implementación de un método para renderizar el mapa utilizando `SDL`.
- Adaptación de la lógica de eventos al sistema de entrada de `SDL`.

Adicionalmente, se aprovechó la migración para añadir más fantasmas al mapa, incrementando el nivel de desafío del juego.

Código con Cambios Implementados

```
#include <SDL.h>
#include <stdbool.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

// Dimensiones del mapa
#define COLUMNAS 22
#define FILAS 13
#define TILE_SIZE 30 // Tamaño de cada celda en píxeles

// Componentes del mapa
#define OBSTACLE '#'
#define POINT '.'
#define PATH ' '

// Pacman y fantasmas
#define PACMAN 'C'
#define GHOST 'G'
#define GHOST2 'H'
#define GHOST3 'I'

#define LONGITUD_MAXIMA_CAMINO 100

//Velocidades de Fantasmas
#define VELOCIDAD_FANTASMA 4
#define VELOCIDAD_FANTASMA2 2
#define VELOCIDAD_FANTASMA3 3

//Retardo de Salida
#define SALIDA_FANTASMA2 50
#define SALIDA_FANTASMA3 100
```

```c
// Registro compuesto por eje x y eje y
typedef struct {
    int x, y;
} Posicion;

// Variables globales de SDL
SDL_Window* window = NULL;
SDL_Renderer* renderer = NULL;
char maze[FILAS][COLUMNAS];

int puntaje = 0;
int puntos_totales = 0;

// Direcciones de movimiento (arriba, abajo, izquierda, derecha)
enum Direccion { ARRIBA, ABAJO, IZQUIERDA, DERECHA, NINGUNA };
Posicion direcciones[] = {{0, -1}, {0, 1}, {-1, 0}, {1, 0}};


// Colores
SDL_Color COLOR_OBSTACLE = {0, 0, 255, 255};    // Azul
SDL_Color COLOR_POINT = {255, 255, 255, 255};   // Blanco
SDL_Color COLOR_GHOST = {255, 0, 0, 255};       // Rojo
SDL_Color COLOR_PACMAN = {255, 255, 0, 255};    // Amarillo

// Función para cargar el mapa desde un archivo
int cargar_mapa(const char* filename) {
    FILE* file = fopen(filename, "r");
    if (!file) {
        perror("No se pudo abrir el archivo de mapa");
        return -1;
    }

    for (int i = 0; i < FILAS; i++) {
        if (fgets(maze[i], COLUMNAS + 2, file) == NULL) {  // +2 para leer el
salto de línea
            perror("Error al leer el mapa");
            fclose(file);
            return -1;
        }
        for (int j = 0; j < COLUMNAS; j++) {
            if (maze[i][j] == POINT) {
                puntos_totales++;
            }
        }
    }
```

```
    fclose(file);
    return 0;
}



// Función para inicializar SDL
int iniciar_SDL() {
    if (SDL_Init(SDL_INIT_VIDEO) < 0) {
        printf("Error al iniciar SDL: %s\n", SDL_GetError());
        return -1;
    }

    window = SDL_CreateWindow("Pacman con SDL2", SDL_WINDOWPOS_UNDEFINED,
SDL_WINDOWPOS_UNDEFINED, COLUMNAS * TILE_SIZE, FILAS * TILE_SIZE,
SDL_WINDOW_SHOWN);
    if (!window) {
        printf("Error al crear la ventana: %s\n", SDL_GetError());
        SDL_Quit();
        return -1;
    }

    renderer = SDL_CreateRenderer(window, -1, SDL_RENDERER_ACCELERATED);
    if (!renderer) {
        printf("Error al crear el renderizador: %s\n", SDL_GetError());
        SDL_DestroyWindow(window);
        SDL_Quit();
        return -1;
    }

    return 0;
}

// Función para dibujar un rectángulo con un color específico
void dibujar_rectangulo(int x, int y, int porcentaje, SDL_Color color) {
    SDL_SetRenderDrawColor(renderer, color.r, color.g, color.b, color.a);
    int size = TILE_SIZE * porcentaje / 100;
    int offset = (TILE_SIZE - size) / 2;
    SDL_Rect rect = {x * TILE_SIZE + offset, y * TILE_SIZE + offset, size,
size};
    SDL_RenderFillRect(renderer, &rect);
}

// Función para dibujar el mapa y los personajes, y contar puntos
void dibujar_juego(Posicion *pacman, Posicion ghost, Posicion ghost2, Posicion
```

```
ghost3) {
    SDL_SetRenderDrawColor(renderer, 0, 0, 0, 255);
    SDL_RenderClear(renderer);

    for (int i = 0; i < FILAS; i++) {
        for (int j = 0; j < COLUMNAS; j++) {
            switch (maze[i][j]) {
                case OBSTACLE:
                    dibujar_rectangulo(j, i, 100, COLOR_OBSTACLE);
                    break;
                case POINT:
                    // Verifica si Pac-Man está en esta posición
                    if (pacman->x == j && pacman->y == i) {
                        puntaje++; // Incrementa el puntaje
                        maze[i][j] = PATH; // Vacía la casilla
                    } else {
                        dibujar_rectangulo(j, i, 10, COLOR_POINT); // Dibuja el
punto
                    }
                    break;
                default:
                    break;
            }
        }
    }

    // Dibujar Pac-Man y los fantasmas
    dibujar_rectangulo(pacman->x, pacman->y, 70, COLOR_PACMAN);
    dibujar_rectangulo(ghost.x, ghost.y, 70, COLOR_GHOST);
    dibujar_rectangulo(ghost2.x, ghost2.y, 70, COLOR_GHOST);
    dibujar_rectangulo(ghost3.x, ghost3.y, 70, COLOR_GHOST);


    SDL_RenderPresent(renderer);
}

// Función para verificar si una posición es transitable
int es_posicion_valida(Posicion pos, Posicion ghost1, Posicion ghost2, Posicion
ghost3) {
    return (pos.x >= 0 && pos.x < COLUMNAS &&
            pos.y >= 0 && pos.y < FILAS &&
            maze[pos.y][pos.x] != OBSTACLE &&
            !(pos.x == ghost1.x && pos.y == ghost1.y) &&
            !(pos.x == ghost2.x && pos.y == ghost2.y) &&
            !(pos.x == ghost3.x && pos.y == ghost3.y));
}
```

```
// Función para calcular la distancia de Manhattan (heurística)
int distancia_manhattan(Posicion a, Posicion b) {
    return abs(a.x - b.x) + abs(a.y - b.y);
}

// Algoritmo A* para encontrar el camino entre el fantasma y Pacman
int a_estrella(Posicion inicio, Posicion objetivo, Posicion
camino[LONGITUD_MAXIMA_CAMINO]) {
    int lista_cerrada[FILAS][COLUMNAS] = {0};
    int longitud_camino = 0;
    Posicion recorrido[FILAS][COLUMNAS] = {{-1, -1}};
    int encontrado = 0;

    typedef struct
    {
        Posicion pos;
        int g, h, f;
    } Nodo;

    Nodo lista_abierta[FILAS * COLUMNAS];
    int contador_abierta = 0;

    Nodo nodo_inicio = {inicio, 0, distancia_manhattan(inicio, objetivo),
distancia_manhattan(inicio, objetivo)};
    lista_abierta[contador_abierta++] = nodo_inicio;

    while (contador_abierta > 0 && !encontrado) {
        int indice_min = 0;
        for (int i = 1; i < contador_abierta; i++) {
            if (lista_abierta[i].f < lista_abierta[indice_min].f) {
                indice_min = i;
            }
        }

        Nodo nodo_actual = lista_abierta[indice_min];
        lista_abierta[indice_min] = lista_abierta[--contador_abierta];
        lista_cerrada[nodo_actual.pos.y][nodo_actual.pos.x] = 1;

        if (nodo_actual.pos.x == objetivo.x && nodo_actual.pos.y == objetivo.y)
{
            Posicion pos = objetivo;
            while (pos.x != inicio.x || pos.y != inicio.y) {
                camino[longitud_camino++] = pos;
                pos = recorrido[pos.y][pos.x];
            }
```

```
            camino[longitud_camino++] = inicio;
            for (int i = 0; i < longitud_camino / 2; i++) {
                Posicion temp = camino[i];
                camino[i] = camino[longitud_camino - i - 1];
                camino[longitud_camino - i - 1] = temp;
            }
            encontrado = 1;
            break;
        }

        for (int i = 0; i < 4; i++) {
            Posicion vecino = {nodo_actual.pos.x + direcciones[i].x,
nodo_actual.pos.y + direcciones[i].y};
            if (vecino.x >= 0 && vecino.x < COLUMNAS && vecino.y >= 0 &&
vecino.y < FILAS &&
                maze[vecino.y][vecino.x] != OBSTACLE && !lista_cerrada[vecino.y]
[vecino.x]) {
                int g = nodo_actual.g + 1;
                int h = distancia_manhattan(vecino, objetivo);
                int f = g + h;

                int en_lista = 0;
                for (int j = 0; j < contador_abierta; j++) {
                    if (lista_abierta[j].pos.x == vecino.x &&
lista_abierta[j].pos.y == vecino.y) {
                        en_lista = 1;
                        if (f < lista_abierta[j].f) {
                            lista_abierta[j].g = g;
                            lista_abierta[j].h = h;
                            lista_abierta[j].f = f;
                            recorrido[vecino.y][vecino.x] = nodo_actual.pos;
                        }
                        break;
                    }
                }

                if (!en_lista) {
                    Nodo nodo_vecino = {vecino, g, h, f};
                    lista_abierta[contador_abierta++] = nodo_vecino;
                    recorrido[vecino.y][vecino.x] = nodo_actual.pos;
                }
            }
        }
    }

    return encontrado ? longitud_camino : -1;
```

```
}



int main(int argc, char *argv[]) {
    // Inicialización
    if (iniciar_SDL() == -1) {
        return -1;
    }

    if (cargar_mapa("mapaConPuntos.txt") == -1) {
        return 1;
    }

    Posicion pacman = {1, 6};
    Posicion ghost = {9, 5};
    Posicion ghost2 = {8, 5};
    Posicion ghost3 = {12, 5};

    enum Direccion direccion_actual = DERECHA;

    //Contadores para velocidad de fantasma
    int contador_fantasma = 0;
    int contador_fantasma2 = 0;  // Contador de velocidad para el segundo
fantasma
    int contador_fantasma3 = 0;

    //Contadores para retardo de salida
    int Contador_Salida_fantasma2 =0;
    int Contador_Salida_fantasma3 = 0;

    bool quit= false;

    SDL_Event e;


    int ch;


    Posicion camino[LONGITUD_MAXIMA_CAMINO];
    Posicion camino2[LONGITUD_MAXIMA_CAMINO];
    Posicion camino3[LONGITUD_MAXIMA_CAMINO];

    int longitud_camino, longitud_camino2, longitud_camino3;

    while (!quit) {
```

```
        dibujar_juego(&pacman, ghost, ghost2, ghost3);
        while (SDL_PollEvent(&e) != 0) {
            if (e.type == SDL_QUIT) {
                quit = true;
            }
            if (e.type == SDL_KEYDOWN) {
                switch (e.key.keysym.sym) {
                    case SDLK_UP:
                        direccion_actual = ARRIBA;
                        break;
                    case SDLK_DOWN:
                        direccion_actual = ABAJO;
                        break;
                    case SDLK_LEFT:
                        direccion_actual = IZQUIERDA;
                        break;
                    case SDLK_RIGHT:
                        direccion_actual = DERECHA;
                        break;
                }
            }
        }

        //Actualiza la Posicion de pacman
        Posicion nueva_posicion = {pacman.x + direcciones[direccion_actual].x,
pacman.y + direcciones[direccion_actual].y};
        if (es_posicion_valida(nueva_posicion, ghost, ghost2, ghost3)) {
            if (maze[nueva_posicion.y][nueva_posicion.x] == POINT) {
                puntaje++;
                maze[nueva_posicion.y][nueva_posicion.x] = PATH;
            }
            pacman = nueva_posicion;
        }

        // Mover el primer fantasma
        if (contador_fantasma >= VELOCIDAD_FANTASMA) {
            longitud_camino = a_estrella(ghost, pacman, camino);
            if (longitud_camino > 1 && es_posicion_valida(camino[1],ghost,
ghost2,ghost3)) {
                ghost = camino[1];
            }
            contador_fantasma = 0;
        } else {
            contador_fantasma++;
        }
```

```
        // Mover el segundo fantasma
        if (contador_fantasma2 >= VELOCIDAD_FANTASMA2 &&
SALIDA_FANTASMA2<=Contador_Salida_fantasma2) {
            longitud_camino2 = a_estrella(ghost2, pacman, camino2);
            if (longitud_camino2 > 1 && es_posicion_valida(camino2[1],
ghost,ghost2,ghost3)) {
                ghost2 = camino2[1];
            }
            contador_fantasma2 = 0;
        }
        else
        {
            if (SALIDA_FANTASMA2>Contador_Salida_fantasma2)
            {
                Contador_Salida_fantasma2++;
            }
            else
            {
                contador_fantasma2++;
            }


        }
        if (contador_fantasma3 >= VELOCIDAD_FANTASMA3 && SALIDA_FANTASMA3 <=
Contador_Salida_fantasma3) {
            longitud_camino3 = a_estrella(ghost3, pacman, camino3);
            if (longitud_camino3 > 1 && es_posicion_valida(camino3[1], ghost,
ghost2, ghost3)) {
                ghost3 = camino3[1];
            }
            contador_fantasma3 = 0;
        } else {
            if (SALIDA_FANTASMA3 > Contador_Salida_fantasma3) {
                Contador_Salida_fantasma3++;
            } else {
                contador_fantasma3++;
            }
        }

        if (puntaje==puntos_totales){
            quit = true;
        }
        else if (pacman.x==ghost.x && pacman.y==ghost.y)
        {
            quit = true;
```

```
        }
        else if (pacman.x==ghost2.x && pacman.y==ghost2.y)
        {
            quit = true;
        }
        else if (pacman.x==ghost3.x && pacman.y==ghost3.y)
        {
            quit = true;
        }
        SDL_Delay(100);
    }
    // Cerrar SDL
    SDL_DestroyRenderer(renderer);
    SDL_DestroyWindow(window);
    SDL_Quit();

    return 0;
}
```

Adiciones Finales y Refinamiento

Animación de Muerte

Se implementó una animación para el evento en que Pac-Man es alcanzado por un fantasma. Durante esta animación, Pac-Man desaparece gradualmente, reduciendo su tamaño visual en la pantalla mediante un procedimiento específico desarrollado para este propósito. Esta mecánica añade un toque visual que mejora la experiencia del jugador, marcando claramente el final del juego.

Código Procedimiento Animación de Muerte

```
// Función para dibujar Animacion de muerte de Pac-Man
void Animacion_Muerte(Mapa* mapa,Posicion pacman, Posicion ghost, Posicion
ghost2, Posicion ghost3) {
    printf("EJE x: %d", pacman.x);
    printf("EJE y: %d", pacman.y);
    dibujar_juego(mapa, &pacman, ghost, ghost2, ghost3);
    dibujar_rectangulo(pacman.x, pacman.y, 100, COLOR_FONDO);
    reproducir_y_liberar_sonido(muerteSound,false,0);
    for (int i = 90; i > 5; i -= 5) // Reduce el porcentaje en cada paso
    {
```

```
        SDL_SetRenderDrawColor(renderer, 255, 255, 0, 255); // Color de Pac-Man
(amarillo)
        dibujar_rectangulo(pacman.x, pacman.y, 100, COLOR_FONDO);
        dibujar_rectangulo(pacman.x, pacman.y, i, COLOR_PACMAN); // Llamada a tu
función de dibujo con tamaño basado en porcentaje
        SDL_RenderPresent(renderer); // Actualiza la pantalla para mostrar el
cambio
        SDL_Delay(50); // Espera para ver la animación (puedes ajustar el valor
según el ritmo)
    }
    // Limpiar el área de Pac-Man después de la animación (simulando su
desaparición)
    SDL_SetRenderDrawColor(renderer, 0, 0, 0, 255); // Color de fondo (negro)
    dibujar_rectangulo(pacman.x, pacman.y, 0, COLOR_FONDO); // Eliminar Pac-Man
    SDL_RenderPresent(renderer); // Actualiza la pantalla para reflejar la
eliminación
    mostrar_mensaje(mapa,"GAME OVER", COLOR_GHOST);

}
```

Mensaje en Pantalla

Para mejorar la comunicación visual durante el juego, se integró la sub-librería `SDL_ttf`, que permite renderizar texto utilizando fuentes personalizadas. Se descargó una fuente representativa del estilo de Pac-Man y se implementaron las funciones necesarias para mostrar mensajes en pantalla, como "Game Over" o "Nivel Completado". Este cambio aporta una capa adicional de interacción visual y contextualiza los eventos clave del juego.

Ejemplo de Procedimiento Para Mostrar Mensaje en Pantalla

```
//Función para mostrar un mensaje en pantalla
void mostrar_mensaje(Mapa* mapa,const char* mensaje, SDL_Color color) {
    // Inicializar TTF y verificar errores
    if (TTF_Init() == -1)
    {
        printf("Error al inicializar SDL_ttf: %s\n", TTF_GetError());
        return;
    }
    // Cargar la fuente (asegúrate de tener el archivo .ttf en el directorio)
    TTF_Font* font = TTF_OpenFont("Letras.ttf", 64);
    if (!font) {
        printf("Error al cargar la fuente: %s\n", TTF_GetError());
        return;
    }
```

```
    // Crea la superficie de texto
    SDL_Surface* surfaceMessage = TTF_RenderText_Solid(font, mensaje, color);
    if (!surfaceMessage) {
        printf("Error al crear la superficie de texto: %s\n", TTF_GetError());
        TTF_CloseFont(font);
        TTF_Quit();
        return;
    }

    // Convertir la superficie en una textura
    SDL_Texture* message = SDL_CreateTextureFromSurface(renderer,
surfaceMessage);
    SDL_FreeSurface(surfaceMessage); // Liberar la superficie ya que no se
necesita más

    // Obtener el tamaño del texto para posicionarlo en el centro
    int text_width, text_height;
    TTF_SizeText(font, "GAME OVER", &text_width, &text_height);
    SDL_Rect textRect = { ((mapa->columnas*TILE_SIZE) - text_width) / 2, (mapa-
>filas*TILE_SIZE) / 2, text_width, text_height };

    // Limpiar la pantalla, luego dibujar el texto
    SDL_SetRenderDrawColor(renderer, 0, 0, 0, 255); // Color de fondo negro
    SDL_RenderClear(renderer);
    SDL_RenderCopy(renderer, message, NULL, &textRect); // Renderizar el texto
    SDL_RenderPresent(renderer); // Mostrar en pantalla

    SDL_Delay(2000); // Mantener el texto visible durante 2 segundos

    // Limpiar recursos
    SDL_DestroyTexture(message);
    TTF_CloseFont(font);
    TTF_Quit();
}
```

Sonidos

Con el objetivo de enriquecer la experiencia auditiva del jugador, se integró la sub-librería `SDL_mixer`, que permite la reproducción de sonidos. Se buscaron y descargaron los efectos de sonido clásicos de Pac-Man, como el sonido al comer puntos y el de derrota. Además, se desarrollaron funciones específicas para gestionar y reproducir estos sonidos en los momentos apropiados, optimizando la inmersión del jugador.

Código Ejemplo de Implementación

```
//Funcion para cargar sonido de pacman
void cargar_sonido() {
    if (Mix_OpenAudio(44100, MIX_DEFAULT_FORMAT, 2, 2048) < 0) {
        printf("Error al inicializar SDL_mixer: %s\n", Mix_GetError());
    }

    pacmanSound = Mix_LoadWAV("pacman_chomp.wav"); // Archivo de sonido, coloca
el archivo .wav en tu directorio
    if (pacmanSound == NULL) {
        printf("Error al cargar el sonido: %s\n", Mix_GetError());
    }

    inicioSound = Mix_LoadWAV("Inicio.wav"); // Archivo de sonido, coloca el
archivo .wav en tu directorio
    if (inicioSound == NULL) {
        printf("Error al cargar el sonido: %s\n", Mix_GetError());
    }

    muerteSound = Mix_LoadWAV("Muerte.wav"); // Archivo de sonido, coloca el
archivo .wav en tu directorio
    if (muerteSound == NULL) {
        printf("Error al cargar el sonido: %s\n", Mix_GetError());
    }

}

//Funcion para reproducir sonido
void reproducir_y_liberar_sonido(Mix_Chunk *Sonido,bool delay,int
Duracion_delay) {
    if (Sonido != NULL) {
        Mix_PlayChannel(-1, Sonido, 0); // Reproducir el sonido una vez
        if (delay)
        {
            SDL_Delay(Duracion_delay); // Duracion del sonido
            Mix_FreeChunk(Sonido); // Liberar el recurso después de reproducir
        }
    }
}
```

Lógica de Niveles

La implementación de la lógica de niveles se realizó al final del proyecto, lo que planteó varios desafíos técnicos debido a la necesidad de ajustar partes fundamentales del código. Entre las principales modificaciones se incluyeron:

- Un nuevo bucle dedicado exclusivamente al manejo de niveles.
- Una nueva estructura de datos para gestionar mapas de manera eficiente.
- Cambios significativos en la lógica global para administrar mapas y posiciones de manera dinámica.
- Variables adicionales para controlar la velocidad tanto de Pac-Man como de los fantasmas.

Estos cambios, junto con otras adaptaciones globales, permitieron integrar un sistema de niveles progresivos con incrementos en la dificultad, completando así el desarrollo del juego.

Retos y Soluciones

Principales Retos Técnicos

A lo largo del desarrollo del proyecto, se identificaron diversos retos técnicos y de diseño. A continuación, se describen los principales desafíos enfrentados, junto con las soluciones adoptadas para resolverlos:

Implementación inicial en ncurses

- Reto: La decisión de usar `ncurses` como herramienta de renderizado inicial simplificó el desarrollo temprano, pero presentó limitaciones gráficas significativas:
  - Imposibilidad de representar colores y animaciones de manera atractiva.
  - Dificultad para gestionar eventos en tiempo real de manera fluida.
- Solución: Migrar a SDL2, una librería gráfica más versátil, que permitió:
  - Implementar gráficos en 2D con colores.
  - Añadir efectos visuales y sonoros para mejorar la experiencia del jugador.

Diseño del mapa

- Reto: La representación del mapa como una matriz estática dificultaba la flexibilidad para crear o modificar niveles.
- Solución:
  - Implementar una función para cargar mapas desde archivos externos, permitiendo personalizar y escalar el juego.
  - Añadir validaciones para asegurar que los mapas cumplan con los requisitos del diseño del juego.

Movimiento de Pac-Man

- Reto: Inicialmente, el movimiento de Pac-Man requería presionar teclas repetidamente, lo cual era poco fluido y alejado de la experiencia original.
- Solución:
  - Crear una lógica para mantener el movimiento continuo en la última dirección indicada por el jugador.

Algoritmo de búsqueda para los fantasmas

- Reto: Implementar una inteligencia artificial básica para los fantasmas que permitiera perseguir a Pac-Man sin afectar el rendimiento del juego.
- Solución:
  - Utilizar el algoritmo A* para calcular el camino más corto hacia Pac-Man en cada iteración.
  - Introducir limitaciones en la frecuencia de cálculo para mantener un buen rendimiento y variar los movimientos de los fantasmas.

Integración de múltiples fantasmas

- Reto: Sincronizar los movimientos de varios fantasmas en un mismo mapa sin interferencias o colisiones entre ellos.
- Solución:
  - Asignar velocidades diferentes a cada fantasma para generar variedad.
  - Incorporar retardo en la salida de algunos fantasmas al inicio de cada nivel, simulando la mecánica del juego original.

Gestión de sonidos y animaciones

- Reto: Añadir sonidos y animaciones sin interrumpir el flujo del juego.
- Solución:
  - Integrar `SDL_mixer` para manejar los efectos sonoros de manera eficiente.
  - Diseñar animaciones para eventos clave, como la muerte de Pac-Man, reduciendo su tamaño de forma progresiva para dar un efecto visual impactante.

Escalabilidad y dificultad progresiva

- Reto: Diseñar una mecánica que aumente la dificultad del juego de forma escalonada y coherente entre niveles.
- Solución:
  - Implementar cambios de velocidad en los fantasmas y una mayor densidad de obstáculos en niveles avanzados.

Tiempo limitado para desarrollo

- Reto: El tiempo restringido dificultó la implementación de algunas mecánicas originales, como los Power Pellets o los comportamientos avanzados de los fantasmas.
- Solución:
  - Priorizar las funcionalidades esenciales, como el movimiento, los puntos, y la IA básica, dejando espacio para mejoras futuras.

Librerías y compilación

- Reto: La configuración del entorno de trabajo presentó múltiples desafíos:
  - No se reconocían las rutas donde se encontraban las librerías necesarias, lo que impedía la compilación.
  - La falta de documentación clara para integrar SDL2 con el entorno dificultó el avance.
- Solución:
  - Se realizaron múltiples ajustes en los archivos de configuración, incluyendo variables de entorno y rutas absolutas para las librerías.

Contribuciones de la IA al Proyecto

La inteligencia artificial desempeñó un papel clave como herramienta de soporte en diversas áreas del desarrollo del proyecto. Entre sus principales contribuciones se destacan:

Integración del Algoritmo A*

- Desafío: Comprender y adaptar el Algoritmo de búsqueda A* al contexto del juego, asegurando su funcionamiento en tiempo real con la posición dinámica de Pac-Man.
- Contribución de la IA: Proporcionó explicaciones claras sobre el funcionamiento del algoritmo y ejemplos prácticos.

Selección y uso de librerías de renderizado

- Desafío: No se tenía conocimiento previo sobre librerías gráficas adecuadas para renderizar el juego ni sobre las funciones específicas que ofrecían.
- Contribución de la IA: Facilitó información detallada sobre librerías como `SDL2`, ayudando a comprender su funcionalidad y guiando en la implementación de renderizados básicos y avanzados.

Resolución de problemas en el entorno de trabajo

- Desafío: La configuración del entorno presentó múltiples errores, desde rutas mal configuradas hasta incompatibilidades con las librerías utilizadas.
- Contribución de la IA: Brindó soluciones rápidas y precisas para los errores, desde ajustes en las variables de entorno hasta recomendaciones para depurar problemas de compilación.

Soporte general en el desarrollo

- Aunque de manera menos significativa, la IA también ofreció apoyo en áreas como:
  - Resolución de errores específicos en el código.
  - Generación de ejemplos prácticos para probar nuevas funcionalidades.
  - Sugerencias para mejorar la estructura del código.

Reflexión sobre el uso de la IA

Si bien la IA fue una herramienta fundamental para superar retos técnicos y acelerar el desarrollo, su papel fue de soporte. La toma de decisiones clave y la lógica central del juego se desarrollaron a partir de investigación propia, con la IA actuando como un complemento para resolver dudas específicas y optimizar procesos.

Resultados Finales

El juego terminado incluye los siguientes componentes clave, que fueron probados y funcionan correctamente:

1. **Movimiento de Pac-Man**:
   - Control continuo y fluido a través del mapa, siguiendo la dirección indicada por el jugador.
2. **Persecución por parte de los fantasmas**:
   - Implementación de un algoritmo de búsqueda (A*) que permite a los fantasmas seguir a Pac-Man de manera dinámica y adaptativa.
3. **Recolección de puntos**:
   - Sistema de puntuación que contabiliza los puntos recolectados por Pac-Man al limpiar el mapa.
4. **Niveles con dificultad progresiva**:
   - Aumento en la velocidad de los fantasmas, mayor complejidad en los mapas y densidad de obstáculos al avanzar entre niveles.

Funcionalidades adicionales

Aunque no eran esenciales para el funcionamiento básico del juego, se añadieron las siguientes características para enriquecer la experiencia del jugador:

- **Carga de mapas personalizables**:
  - Implementación de un sistema que permite cargar mapas desde archivos externos, facilitando la creación de nuevos niveles.

- Música y efectos sonoros:
  - Integración de sonidos clásicos del juego, como el consumo de puntos y la muerte de Pac-Man, utilizando la librería `SDL_mixer`.
- Mensajes en pantalla:
  - Uso de la librería `SDL_ttf` para mostrar mensajes contextuales, como "Juego terminado" o "Nivel Completado", con fuentes personalizadas.
- Renderizado avanzado:
  - Migración de `ncurses` a `SDL2`, mejorando considerablemente los gráficos del juego y la experiencia visual del jugador.

Estos logros reflejan un balance entre la implementación de los componentes esenciales y la integración de funcionalidades opcionales que enriquecen la jugabilidad.

Conclusión

El desarrollo de este proyecto permitió no solo afianzar conceptos fundamentales de programación en C, sino también explorar temas avanzados y enfrentar retos técnicos significativos. Entre los aprendizajes más destacados se incluyen:

Conceptos reforzados:

- Matrices y punteros:
  - Uso intensivo para la representación del mapa y el control de objetos en el juego.
- Algoritmos de búsqueda:
  - Implementación práctica del algoritmo A* y su adaptación al contexto de un juego en tiempo real.

Librerías estudiadas e implementadas:

1. `ncurses`: Para el prototipado inicial y la lógica básica del juego en la terminal.
2. `SDL2`: Para el renderizado gráfico avanzado y la gestión de eventos en tiempo real.
3. `SDL_ttf`: Para la implementación de fuentes personalizadas y mensajes en pantalla.
4. `SDL_mixer`: Para la reproducción de música y efectos sonoros, mejorando la experiencia auditiva.

Reflexión final:

Este proyecto sirvió como una excelente práctica para consolidar habilidades en programación y entender el potencial de las librerías externas. Con más tiempo, se podrían añadir:

- Mecánicas faltantes del juego original, como los Power Pellets y las estrategias avanzadas de los fantasmas.

- Mejores gráficos y animaciones para una experiencia más inmersiva.

En resumen, el proyecto no solo cumplió con los objetivos iniciales, sino que también dejó bases sólidas para continuar explorando el desarrollo de juegos en C.

Código Fuente

```c
// ========================== Librerías ==========================
#include <SDL.h>        // Librería para manejo de gráficos y eventos
#include <stdbool.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <SDL_ttf.h>    // Librería para renderizado de fuentes
#include <SDL_mixer.h>  // Librería para manejo de audio

// ========================== Definiciones ==========================

// Dimensiones máximas del mapa
#define COLUMNAS 50
#define FILAS 50
char maze[FILAS][COLUMNAS]; // Mapa del laberinto

// Tamaño de cada celda en píxeles (cada tile mide 30x30 píxeles)
#define TILE_SIZE 34

// Caracteres que representan componentes del mapa
#define OBSTACLE '#'   // Obstáculo o pared
#define POINT '.'      // Punto comestible
#define PATH ' '       // Camino libre

// Caracteres que representan los personajes
#define PACMAN 'C'
#define GHOST 'G'
#define GHOST2 'H'
#define GHOST3 'I'

// Longitud máxima del camino calculado (A*), usado para IA de fantasmas
#define LONGITUD_MAXIMA_CAMINO 100

// Velocidades de los personajes (menor valor, más rápido)
#define VELOCIDAD_PACMAN 100
#define VELOCIDAD_FANTASMA 400
#define VELOCIDAD_FANTASMA2 200
#define VELOCIDAD_FANTASMA3 300
```

```c
// Cantidad total de niveles en el juego
#define cant_tot_niv 3

// Tiempo de retardo para la salida de fantasmas adicionales (en milisegundos)
#define SALIDA_FANTASMA2 4000  // Fantasma 2 sale después de 4 segundos
#define SALIDA_FANTASMA3 7500  // Fantasma 3 sale después de 7.5 segundos

// ========================== Estructuras de Datos ==========================

// Estructura para representar un mapa
typedef struct {
    char maze[FILAS][COLUMNAS]; // Matriz que representa el laberinto
    char* nombre;               // Nombre del archivo del mapa
    int columnas, filas;        // Número de columnas y filas del mapa
} Mapa;

// Estructura para representar una posición en el mapa
typedef struct {
    int x, y; // Coordenadas X e Y
} Posicion;

// ========================== Variables Globales ==========================

// Arreglo de mapas para los niveles
Mapa mapas[] = {
    {.nombre = "Mapa_1.txt", .filas = 13, .columnas = 22}, // Nivel 1
    {.nombre = "Mapa_1.txt", .filas = 13, .columnas = 22}, // Nivel 2
    {.nombre = "Mapa_1.txt", .filas = 13, .columnas = 22}, // Nivel 3
};

// Nivel actual del juego (comienza en 0)
int nivel_actual = 0;

// ========================== Variables SDL ==========================

// Ventana y renderizador de SDL
SDL_Window* window = NULL;
SDL_Renderer* renderer = NULL;

// ========================== Variables de Juego ==========================

// Variables para puntaje
int puntaje = 0;         // Puntaje acumulado del jugador
int puntos_totales = 0;  // Puntos necesarios para completar el nivel
```

```
// ========================== Movimientos ==========================

// Enum para direcciones de movimiento
enum Direccion { ARRIBA, ABAJO, IZQUIERDA, DERECHA, NINGUNA };

// Arreglo que define los desplazamientos para cada dirección
Posicion direcciones[] = {
    {0, -1},  // ARRIBA
    {0, 1},   // ABAJO
    {-1, 0},  // IZQUIERDA
    {1, 0}    // DERECHA
};

// ========================== Colores ==========================

// Definición de colores para los diferentes elementos del juego
SDL_Color COLOR_OBSTACLE = {0, 0, 255, 255};    // Azul para obstáculos
SDL_Color COLOR_POINT = {255, 255, 255, 255};   // Blanco para puntos
SDL_Color COLOR_GHOST = {255, 0, 0, 255};       // Rojo para fantasmas
SDL_Color COLOR_PACMAN = {255, 255, 0, 255};    // Amarillo para Pac-Man
SDL_Color COLOR_FONDO = {0, 0, 0, 255};         // Negro para el fondo

// ========================== Variables de Audio ==========================

// Variables para controlar la reproducción del sonido
int ultimaX = -1, ultimaY = -1; // Posiciones anteriores de Pac-Man
Uint32 ultimoTiempoSonido = 0;  // Tiempo del último sonido reproducido
Uint32 delaySonido = 100;       // Delay de 100 ms entre sonidos

// Punteros a los efectos de sonido
Mix_Chunk *pacmanSound = NULL;  // Sonido de comer puntos
Mix_Chunk *inicioSound = NULL;  // Sonido de inicio del juego
Mix_Chunk *muerteSound = NULL;  // Sonido de muerte de Pac-Man



// ========================== Función cargar_mapa ==========================
// Función para cargar el mapa desde un archivo y contar los puntos
// Recibe un puntero a Mapa (estructura que contiene el nombre del archivo,
filas y columnas)
int cargar_mapa(Mapa* mapa) {
    // Abre el archivo de texto que contiene el diseño del mapa
    FILE* file = fopen(mapa->nombre, "r");
    if (!file) {
        // Si no se pudo abrir el archivo, muestra un error y retorna -1
        perror("No se pudo abrir el archivo de mapa");
```

```
        return -1;
    }

    puntos_totales = 0; // Inicializa el contador de puntos

    // Lee el archivo línea por línea y guarda el contenido en la matriz del
mapa
    for (int i = 0; i < mapa->filas; i++) {
        // fgets lee una línea del archivo y la almacena en el arreglo
correspondiente
        if (fgets(mapa->maze[i], mapa->columnas + 2, file) == NULL) {
            // Si hay un error al leer una línea, muestra un error, cierra el
archivo y retorna -1
            perror("Error al leer el mapa");
            fclose(file);
            return -1;
        }

        // Cuenta el número de puntos ('.') en el mapa para calcular el puntaje
total
        for (int j = 0; j < mapa->columnas; j++) {
            if (mapa->maze[i][j] == POINT) {
                puntos_totales++;
            }
        }
    }

    fclose(file); // Cierra el archivo después de cargar el mapa
    return 0; // Retorna 0 si se cargó correctamente
}



// ========================== Función cargar_sonido ==========================
// Función para cargar los efectos de sonido utilizando SDL_mixer
void cargar_sonido() {
    // Inicializa SDL_mixer con frecuencia de audio de 44100 Hz, formato de
audio predeterminado
    if (Mix_OpenAudio(44100, MIX_DEFAULT_FORMAT, 2, 2048) < 0) {
        printf("Error al inicializar SDL_mixer: %s\n", Mix_GetError());
    }

    // Carga el sonido de Pac-Man al comer
    pacmanSound = Mix_LoadWAV("pacman_chomp.wav");
    if (pacmanSound == NULL) {
        printf("Error al cargar el sonido: %s\n", Mix_GetError());
```

```
    }

    // Carga el sonido de inicio del juego
    inicioSound = Mix_LoadWAV("Inicio.wav");
    if (inicioSound == NULL) {
        printf("Error al cargar el sonido: %s\n", Mix_GetError());
    }

    // Carga el sonido de la muerte de Pac-Man
    muerteSound = Mix_LoadWAV("Muerte.wav");
    if (muerteSound == NULL) {
        printf("Error al cargar el sonido: %s\n", Mix_GetError());
    }
}



// ====================== Función reproducir_y_liberar_sonido
======================
// Función para reproducir un efecto de sonido y liberar el recurso si se
especifica
// Recibe el sonido a reproducir, un booleano para indicar si se debe esperar
(delay) y la duración del delay
void reproducir_y_liberar_sonido(Mix_Chunk *Sonido, bool delay, int
Duracion_delay) {
    if (Sonido != NULL) {
        // Reproduce el sonido en el canal -1 (primer canal disponible)
        Mix_PlayChannel(-1, Sonido, 0);

        // Si se especifica, espera la duración indicada y libera el recurso de
sonido
        if (delay) {
            SDL_Delay(Duracion_delay);   // Espera la duración del sonido
            Mix_FreeChunk(Sonido);       // Libera el recurso de sonido
        }
    }
}



// ========================= Función mostrar_mensaje ========================
// Función para mostrar un mensaje de texto en pantalla, centrado
// Recibe el mapa, el mensaje de texto, y el color del texto
void mostrar_mensaje(Mapa* mapa, const char* mensaje, SDL_Color color) {
    // Inicializa SDL_ttf para manejo de fuentes y verifica errores
    if (TTF_Init() == -1) {
```

```
        printf("Error al inicializar SDL_ttf: %s\n", TTF_GetError());
        return;
    }

    // Carga la fuente desde el archivo .ttf con un tamaño de 64 puntos
    TTF_Font* font = TTF_OpenFont("Letras.ttf", 64);
    if (!font) {
        printf("Error al cargar la fuente: %s\n", TTF_GetError());
        return;
    }

    // Crea una superficie con el texto renderizado de forma sólida
    SDL_Surface* surfaceMessage = TTF_RenderText_Solid(font, mensaje, color);
    if (!surfaceMessage) {
        printf("Error al crear la superficie de texto: %s\n", TTF_GetError());
        TTF_CloseFont(font);
        TTF_Quit();
        return;
    }

    // Convierte la superficie de texto a una textura para renderizarla
    SDL_Texture* message = SDL_CreateTextureFromSurface(renderer,
surfaceMessage);
    SDL_FreeSurface(surfaceMessage); // Libera la superficie ya que no se
necesita más

    // Calcula el tamaño y posición del texto para centrarlo en pantalla
    int text_width, text_height;
    TTF_SizeText(font, mensaje, &text_width, &text_height);
    SDL_Rect textRect = {
        ((mapa->columnas * TILE_SIZE) - text_width) / 2,  // Posición X centrada
        (mapa->filas * TILE_SIZE) / 2,                    // Posición Y centrada
        text_width,
        text_height
    };

    // Limpia la pantalla y establece el color de fondo a negro
    SDL_SetRenderDrawColor(renderer, 0, 0, 0, 255);
    SDL_RenderClear(renderer);

    // Renderiza el texto en pantalla
    SDL_RenderCopy(renderer, message, NULL, &textRect);
    SDL_RenderPresent(renderer); // Muestra el renderizado en pantalla

    SDL_Delay(2000); // Pausa la pantalla por 2 segundos para que el mensaje sea
visible
```

```
    // Libera los recursos utilizados
    SDL_DestroyTexture(message);
    TTF_CloseFont(font);
    TTF_Quit();
}



// ========================== Función iniciar_SDL ==========================
// Función para inicializar SDL y crear la ventana y el renderizador
// Recibe un parámetro "mapa" de tipo Mapa para ajustar el tamaño de la ventana
int iniciar_SDL(Mapa mapa) {

    // Inicializar SDL con el subsistema de video
    if (SDL_Init(SDL_INIT_VIDEO) < 0) {
        // Si falla la inicialización, muestra el error y retorna -1
        printf("Error al iniciar SDL: %s\n", SDL_GetError());
        return -1;
    }

    // Crear una ventana para el juego Pac-Man con las dimensiones basadas en el
mapa
    window = SDL_CreateWindow(
        "Pacman con SDL2",                // Título de la ventana
        SDL_WINDOWPOS_UNDEFINED,          // Posición X de la ventana (centrada)
        SDL_WINDOWPOS_UNDEFINED,          // Posición Y de la ventana (centrada)
        mapa.columnas * TILE_SIZE,        // Ancho de la ventana (columnas *
tamaño de celda)
        mapa.filas * TILE_SIZE,           // Alto de la ventana (filas * tamaño
de celda)
        SDL_WINDOW_SHOWN                  // Mostrar ventana al crearla
    );

    // Verificar si la ventana se creó correctamente
    if (!window) {
        // Si falla la creación de la ventana, muestra el error, cierra SDL y
retorna -1
        printf("Error al crear la ventana: %s\n", SDL_GetError());
        SDL_Quit();
        return -1;
    }

    // Crear el renderizador para dibujar en la ventana
    renderer = SDL_CreateRenderer(
```

```
        window,                          // Ventana asociada
        -1,                              // Índice del driver (-1 para usar el
primero disponible)
        SDL_RENDERER_ACCELERATED         // Utilizar renderizado acelerado por
hardware
    );

    // Verificar si el renderizador se creó correctamente
    if (!renderer) {
        // Si falla la creación del renderizador, muestra el error, destruye la
ventana, cierra SDL y retorna -1
        printf("Error al crear el renderizador: %s\n", SDL_GetError());
        SDL_DestroyWindow(window);       // Destruir la ventana creada
previamente
        SDL_Quit();                      // Cerrar SDL
        return -1;
    }

    // Si todo se inicializó correctamente, retorna 0
    return 0;
}



// ======================== Función dibujar_rectangulo ======================
// Función para dibujar un rectángulo en la posición especificada con un color
dado
// Parámetros:
//   x, y: Coordenadas del rectángulo en el mapa
//   porcentaje: Tamaño del rectángulo como un porcentaje del tamaño de una
celda (TILE_SIZE)
//   color: Color del rectángulo (estructura SDL_Color)
void dibujar_rectangulo(int x, int y, int porcentaje, SDL_Color color) {
    // Establece el color para el renderizador
    SDL_SetRenderDrawColor(renderer, color.r, color.g, color.b, color.a);

    // Calcula el tamaño del rectángulo basado en el porcentaje
    int size = (TILE_SIZE * porcentaje) / 100;
    int offset = (TILE_SIZE - size) / 2; // Calcula el desplazamiento para
centrar el rectángulo

    // Define el rectángulo a dibujar
    SDL_Rect rect = {x * TILE_SIZE + offset, y * TILE_SIZE + offset, size,
size};

    // Dibuja el rectángulo relleno
```

```
    SDL_RenderFillRect(renderer, &rect);
}



// ======================== Función dibujar_juego ========================
// Función para dibujar el mapa, Pac-Man, los fantasmas y contar puntos
void dibujar_juego(Mapa* mapa, Posicion* pacman, Posicion ghost, Posicion
ghost2, Posicion ghost3) {
    // Limpia la pantalla y establece el color de fondo a negro
    SDL_SetRenderDrawColor(renderer, 0, 0, 0, 255);
    SDL_RenderClear(renderer);

    // Obtiene el tiempo actual para futuras animaciones (aunque aquí no se
utiliza)
    Uint32 tiempoActual = SDL_GetTicks();

    // Itera sobre las filas y columnas del mapa
    for (int i = 0; i < mapa->filas; i++) {
        for (int j = 0; j < mapa->columnas; j++) {
            switch (mapa->maze[i][j]) {
                case OBSTACLE:
                    // Dibuja un obstáculo (bloque) en el mapa
                    dibujar_rectangulo(j, i, 100, COLOR_OBSTACLE);
                    break;
                case POINT:
                    // Si Pac-Man está en la posición del punto, elimina el
punto
                    if (pacman->x == j && pacman->y == i) {
                        mapa->maze[i][j] = PATH; // Marca la celda como vacía
                    } else {
                        // Dibuja el punto con un tamaño más pequeño (10% de la
celda)
                        dibujar_rectangulo(j, i, 10, COLOR_POINT);
                    }
                    break;
                default:
                    break;
            }
        }
    }

    // Dibuja a Pac-Man con un tamaño del 85% de la celda
    dibujar_rectangulo(pacman->x, pacman->y, 85, COLOR_PACMAN);

    // Dibuja los tres fantasmas con un tamaño del 75% de la celda
```

```
    dibujar_rectangulo(ghost.x, ghost.y, 75, COLOR_GHOST);
    dibujar_rectangulo(ghost2.x, ghost2.y, 75, COLOR_GHOST);
    dibujar_rectangulo(ghost3.x, ghost3.y, 75, COLOR_GHOST);

    // Muestra el renderizado en pantalla
    SDL_RenderPresent(renderer);
}



// ======================= Función Animacion_Muerte =======================
// Función para mostrar la animación de la muerte de Pac-Man
void Animacion_Muerte(Mapa* mapa, Posicion pacman, Posicion ghost, Posicion
ghost2, Posicion ghost3) {

    // Dibuja el estado actual del juego
    dibujar_juego(mapa, &pacman, ghost, ghost2, ghost3);

    // Limpia la posición de Pac-Man
    dibujar_rectangulo(pacman.x, pacman.y, 100, COLOR_FONDO);

    // Reproduce el sonido de muerte de Pac-Man
    reproducir_y_liberar_sonido(muerteSound, false, 0);

    // Animación de reducción del tamaño de Pac-Man
    for (int i = 90; i > 5; i -= 5) {
        // Establece el color de Pac-Man (amarillo)
        SDL_SetRenderDrawColor(renderer, 255, 255, 0, 255);

        // Limpia la posición de Pac-Man
        dibujar_rectangulo(pacman.x, pacman.y, 100, COLOR_FONDO);

        // Dibuja Pac-Man con un tamaño decreciente
        dibujar_rectangulo(pacman.x, pacman.y, i, COLOR_PACMAN);

        // Muestra el cambio en pantalla
        SDL_RenderPresent(renderer);

        // Espera un breve periodo para crear el efecto de animación
        SDL_Delay(50);
    }

    // Limpia la pantalla después de la animación
    SDL_SetRenderDrawColor(renderer, 0, 0, 0, 255);
    dibujar_rectangulo(pacman.x, pacman.y, 0, COLOR_FONDO);
    SDL_RenderPresent(renderer);
```

```
    // Muestra el mensaje "GAME OVER"
    mostrar_mensaje(mapa, "PERDISTE", COLOR_GHOST);
}



// ====================== Función es_posicion_valida ======================
// Verifica si una posición es válida para moverse (no hay obstáculos ni
fantasmas)
// Parámetros:
//   mapa: Estructura del mapa
//   pos: Posición a verificar
//   ghost1, ghost2, ghost3: Posiciones de los fantasmas
int es_posicion_valida(Mapa mapa, Posicion pos, Posicion ghost1, Posicion
ghost2, Posicion ghost3) {
    // Verifica que la posición esté dentro de los límites del mapa y no haya
obstáculos ni fantasmas
    return (pos.x >= 0 && pos.x < mapa.columnas &&
            pos.y >= 0 && pos.y < mapa.filas &&
            mapa.maze[pos.y][pos.x] != OBSTACLE &&
            !(pos.x == ghost1.x && pos.y == ghost1.y) &&
            !(pos.x == ghost2.x && pos.y == ghost2.y) &&
            !(pos.x == ghost3.x && pos.y == ghost3.y));
}

// ================== Función distancia_manhattan ==================
// Calcula la distancia de Manhattan (heurística) entre dos posiciones
// Parámetros:
//   a, b: Dos posiciones del mapa
// Retorna: La distancia de Manhattan entre las posiciones
int distancia_manhattan(Posicion a, Posicion b) {
    // Distancia de Manhattan: |x1 - x2| + |y1 - y2|
    return abs(a.x - b.x) + abs(a.y - b.y);
}



// ================ Algoritmo A* ================
// Función para encontrar el camino más corto entre un fantasma y Pac-Man
// Parámetros:
//   mapa: Estructura del mapa
//   inicio: Posición inicial del fantasma
//   objetivo: Posición de Pac-Man
//   camino: Arreglo para almacenar el camino encontrado
// Retorna: La longitud del camino encontrado o -1 si no hay camino
int a_estrella(Mapa mapa, Posicion inicio, Posicion objetivo, Posicion
```

```
camino[LONGITUD_MAXIMA_CAMINO]) {

    // Inicializa la lista cerrada para marcar las posiciones visitadas
    int lista_cerrada[FILAS][COLUMNAS] = {0};

    // Variable para almacenar la longitud del camino
    int longitud_camino = 0;

    // Arreglo para registrar el recorrido del camino
    Posicion recorrido[FILAS][COLUMNAS] = {{-1, -1}};

    // Variable que indica si se ha encontrado el camino
    int encontrado = 0;

    // Estructura de datos para los nodos del algoritmo A*
    typedef struct {
        Posicion pos; // Posición del nodo
        int g; // Costo desde el nodo inicial hasta este nodo
        int h; // Heurística: Distancia estimada desde este nodo hasta el
objetivo
        int f; // Costo total: g + h
    } Nodo;

    // Lista abierta para almacenar los nodos por explorar
    Nodo lista_abierta[mapa.filas * mapa.columnas];
    int contador_abierta = 0;

    // Inicializa el nodo de inicio
    Nodo nodo_inicio = {inicio, 0, distancia_manhattan(inicio, objetivo),
distancia_manhattan(inicio, objetivo)};

    // Agrega el nodo inicial a la lista abierta
    lista_abierta[contador_abierta++] = nodo_inicio;

    // Bucle principal del algoritmo A*
    while (contador_abierta > 0 && !encontrado) {
        // Encuentra el nodo con el menor costo total (f) en la lista abierta
        int indice_min = 0;
        for (int i = 1; i < contador_abierta; i++) {
            if (lista_abierta[i].f < lista_abierta[indice_min].f) {
                indice_min = i;
            }
        }

        // Extrae el nodo con el menor costo de la lista abierta
        Nodo nodo_actual = lista_abierta[indice_min];
```

```
        lista_abierta[indice_min] = lista_abierta[--contador_abierta]; // Reduce
el tamaño de la lista abierta

        // Marca el nodo actual como visitado en la lista cerrada
        lista_cerrada[nodo_actual.pos.y][nodo_actual.pos.x] = 1;

        // Si se ha alcanzado el objetivo, se reconstruye el camino
        if (nodo_actual.pos.x == objetivo.x && nodo_actual.pos.y == objetivo.y)
{
            Posicion pos = objetivo;
            while (pos.x != inicio.x || pos.y != inicio.y) {
                camino[longitud_camino++] = pos; // Agrega la posición al camino
                pos = recorrido[pos.y][pos.x]; // Sigue el rastro del recorrido
            }
            camino[longitud_camino++] = inicio; // Agrega la posición inicial al
final del camino

            // Invierte el camino para que vaya del inicio al objetivo
            for (int i = 0; i < longitud_camino / 2; i++) {
                Posicion temp = camino[i];
                camino[i] = camino[longitud_camino - i - 1];
                camino[longitud_camino - i - 1] = temp;
            }

            encontrado = 1; // Indica que se ha encontrado el camino
            break; // Sale del bucle
        }

        // Explora los vecinos del nodo actual (arriba, abajo, izquierda,
derecha)
        for (int i = 0; i < 4; i++) {
            // Calcula la posición del vecino
            Posicion vecino = {nodo_actual.pos.x + direcciones[i].x,
nodo_actual.pos.y + direcciones[i].y};

            // Verifica si el vecino es válido (dentro del mapa, no es un
obstáculo y no ha sido visitado)
            if (vecino.x >= 0 && vecino.x < mapa.columnas &&
                vecino.y >= 0 && vecino.y < mapa.filas &&
                mapa.maze[vecino.y][vecino.x] != OBSTACLE &&
                !lista_cerrada[vecino.y][vecino.x]) {

                // Calcula los costos del vecino
                int g = nodo_actual.g + 1; // Costo desde el inicio hasta el
vecino
                int h = distancia_manhattan(vecino, objetivo); // Heurística
```

```
(distancia estimada al objetivo)
                int f = g + h; // Costo total

                // Verifica si el vecino ya está en la lista abierta
                int en_lista = 0;
                for (int j = 0; j < contador_abierta; j++) {
                    if (lista_abierta[j].pos.x == vecino.x &&
lista_abierta[j].pos.y == vecino.y) {
                        en_lista = 1;
                        // Si el nuevo costo total es menor, actualiza el nodo
en la lista abierta
                        if (f < lista_abierta[j].f) {
                            lista_abierta[j].g = g;
                            lista_abierta[j].h = h;
                            lista_abierta[j].f = f;
                            recorrido[vecino.y][vecino.x] = nodo_actual.pos; //
Registra el recorrido
                        }
                        break;
                    }
                }

                // Si el vecino no estaba en la lista abierta, lo agrega
                if (!en_lista) {
                    Nodo nodo_vecino = {vecino, g, h, f};
                    lista_abierta[contador_abierta++] = nodo_vecino;
                    recorrido[vecino.y][vecino.x] = nodo_actual.pos;
                }
            }
        }
    }

    // Retorna la longitud del camino encontrado o -1 si no se encontró un
camino
    return encontrado ? longitud_camino : -1;
}

int main(int argc, char* argv[]) {
    // ========================= Configuración Inicial
=========================

    // Nivel de dificultad inicial
    int nivel_actual = 0;

    // Inicializar las rutas de los fantasmas usando el algoritmo A*
    Posicion camino[LONGITUD_MAXIMA_CAMINO];
```

```
    Posicion camino2[LONGITUD_MAXIMA_CAMINO];
    Posicion camino3[LONGITUD_MAXIMA_CAMINO];

    // Variables para almacenar la longitud del camino calculado por cada
fantasma
    int longitud_camino, longitud_camino2, longitud_camino3;

    // Cargar sonido del juego
    cargar_sonido();

    // Indicador para salir del juego
    bool quit_game = false;

    // ========================== Bucle Principal del Juego
==========================
    while (!quit_game) {

        // Iniciar SDL y cargar el mapa del nivel actual
        if (iniciar_SDL(mapas[nivel_actual]) == -1 ||
cargar_mapa(&mapas[nivel_actual]) == -1) {
            perror("Error al inicializar SDL o cargar el mapa.");
            return -1;
        }

        // Mensaje de inicio en el nivel 0
        if (nivel_actual == 0) {
            mostrar_mensaje(&mapas[nivel_actual], "Pac-Man", COLOR_PACMAN);
            reproducir_y_liberar_sonido(inicioSound, true, 4200);
        }

        // Configuración de velocidades basadas en la dificultad
        int velocidad_pacman = VELOCIDAD_PACMAN - (20 * nivel_actual);
        int velocidad_fantasma = VELOCIDAD_FANTASMA - (25 * nivel_actual);
        int velocidad_fantasma2 = VELOCIDAD_FANTASMA2 - (25 * nivel_actual);
        int velocidad_fantasma3 = VELOCIDAD_FANTASMA3 - (30 * nivel_actual);

        // Posiciones iniciales de Pac-Man y los fantasmas
        Posicion pacman = {1, 6};
        Posicion ghost = {9, 5};
        Posicion ghost2 = {8, 5};
        Posicion ghost3 = {12, 5};
        enum Direccion direccion_actual = DERECHA;

        // Variables para controlar el ciclo del nivel
        bool quit_level = false;
        int contador_pacman = 0, contador_fantasma = 0, contador_fantasma2 = 0,
```

```
contador_fantasma3 = 0;

        // Contadores para el retardo de salida de los fantasmas
        int Contador_Salida_fantasma2 = 0;
        int Contador_Salida_fantasma3 = 0;

        // Reiniciar puntaje del nivel
        puntaje = 0;
        mostrar_mensaje(&mapas[nivel_actual], "Comenzando", COLOR_POINT);

        // ========================== Bucle del Nivel ==========================
        while (!quit_level) {
            // Renderizar el estado actual del juego
            dibujar_juego(&mapas[nivel_actual], &pacman, ghost, ghost2, ghost3);

            // Manejo de eventos SDL
            SDL_Event e;
            while (SDL_PollEvent(&e) != 0) {
                if (e.type == SDL_QUIT) {
                    quit_game = true;
                    quit_level = true;
                } else if (e.type == SDL_KEYDOWN) {
                    switch (e.key.keysym.sym) {
                        case SDLK_UP: direccion_actual = ARRIBA; break;
                        case SDLK_DOWN: direccion_actual = ABAJO; break;
                        case SDLK_LEFT: direccion_actual = IZQUIERDA; break;
                        case SDLK_RIGHT: direccion_actual = DERECHA; break;
                        case SDLK_e: // Saltar nivel
                            mostrar_mensaje(&mapas[nivel_actual], "SALTAR
NIVEL", COLOR_POINT);
                            nivel_actual++;
                            quit_level = true;
                            break;
                        case SDLK_q: // Salir del juego
                            mostrar_mensaje(&mapas[nivel_actual], "SALIENDO DEL
JUEGO", COLOR_PACMAN);
                            quit_game = true;
                            quit_level = true;
                            break;
                    }
                }
            }

            // ========================== Movimiento de Pac-Man
==========================
            if (contador_pacman >= velocidad_pacman) {
```

```
                Posicion nueva_posicion = {
                    pacman.x + direcciones[direccion_actual].x,
                    pacman.y + direcciones[direccion_actual].y
                };
                if (es_posicion_valida(mapas[nivel_actual], nueva_posicion,
ghost, ghost2, ghost3)) {
                    if (mapas[nivel_actual].maze[nueva_posicion.y]
[nueva_posicion.x] == POINT) {
                        puntaje++;
                        if (pacmanSound != NULL && !Mix_Playing(-1)) {
                            Mix_PlayChannel(-1, pacmanSound, 0);
                        }
                        mapas[nivel_actual].maze[nueva_posicion.y]
[nueva_posicion.x] = PATH;
                    }
                    pacman = nueva_posicion;
                }
                contador_pacman = 0;
            } else {
                contador_pacman++;
            }

            // ========================== Movimiento de Fantasmas
==========================
            // Fantasma 1
            if (contador_fantasma >= velocidad_fantasma) {
                longitud_camino = a_estrella(mapas[nivel_actual], ghost, pacman,
camino);
                if (longitud_camino > 1 &&
es_posicion_valida(mapas[nivel_actual], camino[1], ghost, ghost2, ghost3)) {
                    ghost = camino[1];
                }
                contador_fantasma = 0;
            } else {
                contador_fantasma++;
            }

            // Fantasma 2
            if (contador_fantasma2 >= velocidad_fantasma2 && SALIDA_FANTASMA2 <=
Contador_Salida_fantasma2) {
                longitud_camino2 = a_estrella(mapas[nivel_actual], ghost2,
pacman, camino2);
                if (longitud_camino2 > 1 &&
es_posicion_valida(mapas[nivel_actual], camino2[1], ghost, ghost2, ghost3)) {
                    ghost2 = camino2[1];
                }
```

```
                contador_fantasma2 = 0;
            } else {
                if (SALIDA_FANTASMA2 > Contador_Salida_fantasma2) {
                    Contador_Salida_fantasma2++;
                } else {
                    contador_fantasma2++;
                }
            }

            // Fantasma 3
            if (contador_fantasma3 >= velocidad_fantasma3 && SALIDA_FANTASMA3 <=
Contador_Salida_fantasma3) {
                longitud_camino3 = a_estrella(mapas[nivel_actual], ghost3,
pacman, camino3);
                if (longitud_camino3 > 1 &&
es_posicion_valida(mapas[nivel_actual], camino3[1], ghost, ghost2, ghost3)) {
                    ghost3 = camino3[1];
                }
                contador_fantasma3 = 0;
            } else {
                if (SALIDA_FANTASMA3 > Contador_Salida_fantasma3) {
                    Contador_Salida_fantasma3++;
                } else {
                    contador_fantasma3++;
                }
            }

            // ========================== Verificar Condiciones de Fin de Nivel
==========================
            if (puntaje == puntos_totales) {
                mostrar_mensaje(&mapas[nivel_actual], "NIVEL COMPLETADO",
COLOR_PACMAN);
                nivel_actual++;
                if (nivel_actual-1 == cant_tot_niv) {
                    mostrar_mensaje(&mapas[nivel_actual], "GANASTE!!",
COLOR_PACMAN);
                    quit_game = true;
                }
                quit_level = true;
            }

            // Verificar colisión de Pac-Man con los fantasmas
            if ((pacman.x == ghost.x && pacman.y == ghost.y) ||
                (pacman.x == ghost2.x && pacman.y == ghost2.y) ||
                (pacman.x == ghost3.x && pacman.y == ghost3.y)) {
                Animacion_Muerte(&mapas[nivel_actual], pacman, ghost, ghost2,
```

```
ghost3);
                quit_game = true;
                quit_level = true;
            }

            SDL_Delay(1);
        }

        // Destruir recursos SDL del nivel actual
        SDL_DestroyRenderer(renderer);
        SDL_DestroyWindow(window);
    }

    // ========================= Finalizar Juego =========================
    mostrar_mensaje(&mapas[nivel_actual - 1], "JUEGO TERMINADO", COLOR_PACMAN);
    SDL_Delay(2000);

    SDL_Quit();
    Mix_CloseAudio();
    return 0;
}
```

Bibliografía

- SDL Wiki. *"Introduction to SDL2."* Disponible en: https://wiki.libsdl.org
- Videos sobre comportamiento del algoritmo A*:
  - https://www.youtube.com/watch?v=1gszEk8rUS4 (Recomendado)
  - https://www.youtube.com/watch?v=yxN6yR_7yJM&t=542s
  - https://www.youtube.com/watch?v=UVw_sX7AMG0
- Lenguaje C(pdf):https://informatica.uv.es/estguia/ATD/apuntes/laboratorio/Lenguaje-C.pdf