SPELUNKING

Project Report

Authors:
Varchas Jakkaraju (590022789)
Shashwat Kashyap Tiwary (590026643)

Subject:
Game Development with C and Raylib

November 28, 2025

Contents

1 ABSTRACT

Spelunking is a 2D exploration game we built using C and Raylib. The main idea was to create a game with procedural rooms and enemy AI without relying on a heavy game engine. The player has to navigate through random maps, find treasure using a ”hot/cold” tracker, and avoid enemies like zombies and bats. We focused on writing clean C code, managing memory manually, and implementing grid-based collisions from scratch.

2 PROBLEM DEFINITION

2.1 Objective

We needed to develop a functional 2D game that handles real-time input and autonomous enemies. The challenge was to do this from the ground up, using only a basic library for drawing.

2.2 Scope

The project covers:

- **Rendering**: A 20x15 tile grid (16x16 pixels per tile).
- **Input**: Handling keys for moving, turning, digging, and attacking.
- **Logic**:
  - Generating a new room layout when the player leaves the screen.
  - Axis-Aligned Bounding Box (AABB) collision detection.
  - Calculating distance for the treasure tracker.
- **Constraints**: Written in C; Raylib used only for the window and graphics.

3 SYSTEM DESIGN

3.1 Architecture Overview

We split the code into modules to keep ‘main.c‘ clean and readable:

- **Core**: `main.c` (Entry point, game loop).
- **Entities**: `player.c`, `zombie.c`, `treasure.c` (Object logic).
- **Environment**: `room.c` (Map data and generation).
- **Utilities**: `utils.c` (Math), `hud.c` (UI).

3.2 Flowcharts & Algorithms

3.2.1 Algorithm 1: Main Game Loop

```
START
  Init Window, Player, Room

  WHILE (Window is open)
    Get DeltaTime
```

```
    IF Game is Playing
      Check Input
      Update Zombies & Bats
      Check Collisions
      Check Room Change
    ENDIF

    Start Drawing
      Clear Screen
      Draw Map
      Draw Entities
      Draw UI
    Stop Drawing
  ENDWHILE

  Free Memory
  Close Window
END
```

3.2.2 Algorithm 2: Treasure Tracking

Simple distance check to give the player feedback.

```
FUNCTION GetTreasureStatus(playerPos, treasurePos):
  diff_x = playerPos.x - treasurePos.x
  diff_y = playerPos.y - treasurePos.y
  dist = Sqrt(diff_x*diff_x + diff_y*diff_y)

  IF dist < 2: RETURN "BURNING"
  IF dist < 5: RETURN "HOT"
  IF dist < 10: RETURN "WARM"
  RETURN "COLD"
```

3.2.3 Algorithm 3: Zombie Logic

Zombies move on a timer (every 0.75s) to make them feel ”retro”.

```
FUNCTION UpdateZombie(zombie, player):
  zombie.timer += DeltaTime
  IF zombie.timer >= 0.75:
    zombie.timer = 0
    diff_x = player.x - zombie.x
    diff_y = player.y - zombie.y

    // Move towards player on the longest axis
    IF Abs(diff_x) > Abs(diff_y):
      MoveZombie(Sign(diff_x), 0)
    ELSE:
      MoveZombie(0, Sign(diff_y))
```

4 IMPLEMENTATION DETAILS

4.1 Data Structures

We used standard structs to define game objects:

```
typedef struct Vector2Int {
    int x, y;
} Vector2Int;

typedef struct Player {
    Vector2Int position;
    int direction; // 0:N, 1:E, 2:S, 3:W
    int hp;
    bool hasAttacked;
} Player;

typedef struct Room {
    int tiles[20][15];
    int width, height;
} Room;
```

Listing 1: Entity Struct Definitions

4.2 Core Logic Snippets

4.2.1 Player Input

We separated rotation from movement. If you press a new direction, the player just turns. If you press it again, they move.

```
void UpdatePlayerInput(Player* p) {
    if (IsKeyPressed(KEY_UP)) {
        if (p->direction == DIR_NORTH) TryMove(p, 0, -1);
        else p->direction = DIR_NORTH;
    }
    else if (IsKeyPressed(KEY_DOWN)) {
        if (p->direction == DIR_SOUTH) TryMove(p, 0, 1);
        else p->direction = DIR_SOUTH;
    }

    if (IsKeyPressed(KEY_D)) PerformDig(p);
    if (IsKeyPressed(KEY_SPACE)) PerformAttack(p);
}
```

Listing 2: src/player.c Snippet

4.2.2 Room Transitions

When the player hits the edge, we generate a fresh room and spawn new enemies.

```
void CheckRoomTransition(Player* p, Room* r) {
    if (p->position.x < 0 || p->position.x >= r->width ||
        p->position.y < 0 || p->position.y >= r->height) {

        GenerateNewRoom(r);

        // Wrap player to other side
```

```
        if (p->position.x < 0) p->position.x = r->width - 1;
        else if (p->position.x >= r->width) p->position.x = 0;

        SpawnEnemies(r);
    }
}
```

Listing 3: src/room.c Snippet

5 TESTING & RESULTS

5.1 Test Cases

We ran these tests manually to check the mechanics.

ID	Scenario	Result
TC-01	Move to empty tile	PASS
TC-02	Move into wall	PASS (Blocked)
TC-03	Dig normal floor	PASS (Nothing happens)
TC-04	Dig treasure tile	PASS (Chest appears)
TC-05	Zombie hits player	PASS (Lose 1 HP)
TC-06	Bat hits player	PASS (Instant damage)
TC-07	HP drops to 0	PASS (Game Resets)

Table 1: Mechanics Test Suite

5.2 Performance

- **Frame Rate**: Runs at a solid 60 FPS on Linux.
- **Memory**: Very low heap usage since we used static arrays for the grid.
- **Latency**: Controls respond instantly.

6 CONCLUSION & FUTURE WORK

6.1 Conclusion

The project works as intended. We successfully implemented random map generation, enemy pathfinding, and the treasure mechanic. Using C forced us to be careful with memory, but the game runs smoothly without leaks.

6.2 Future Work

- **Graphics**: Replace color blocks with actual sprites.
- **Audio**: Add sound effects for actions like digging.
- **Content**: Add more enemy types and varied levels.
- **Save System**: Implement a way to save progress.

7 REFERENCES

1. Raylib Technologies. *Raylib Cheatsheet*. https://www.raylib.com/cheatsheet/cheatsheet.html
2. Kernighan, B. W., & Ritchie, D. M. (1988). *The C Programming Language* (2nd ed.). Prentice Hall.
3. GitHub Repository: raysan5/raylib. https://github.com/raysan5/raylib
4. BMo. ”Making A Game in C With Raylib”. YouTube, uploaded by BMo, 25 May 2025, https://www.youtub Bktq791Qo
5. Project Repository (Varchas Jakkaraju): https://github.com/VarchasJ/SPELUNKING
6. Project Repository (Shashwat Kashyap Tiwary): https://github.com/Basix-95/spelunking/