CS221: Computer Programming 1 - Final Project Report: Raylib Chess Engine

true true

December 24, 2025

Alexandria University

Faculty of Engineering

Computer and Systems Engineering Department

Course: CS221: Computer Programming 1
Project: Chess Engine Implementation in C
Instructor: [Prof.Dr. Marwan Torki]
Submission Date: December 24, 2025

Abd-El-Rhaman Mohamed Mohamed Seif (ID: 24010397)

Omar Mohamed Abd-El-Mohsen (ID: 24010467)

Table of Contents

---

1. Executive Summary

The objective of this project was to design and implement a fully functional Chess game using the C programming language. While the initial academic requirements specified a console-based application using ASCII characters, our team identified an opportunity to apply advanced software engineering principles by developing a graphical application.

We utilized the **Raylib** game programming library to create a hardware-accelerated 2D interface. The resulting application is not merely a visual upgrade but a complete rewrite of the standard chess engine logic to support **event-driven programming**.

Key achievements include: * **Robust Engine:** A move validation system that strictly adheres to FIDE laws of chess, including complex edge cases like En Passant, Castling, and Promotion. * **Persistence:** Implementation of the Forsyth–Edwards Notation (FEN) standard, allowing game states to be saved to disk and loaded seamlessly. * **Dynamic Memory:** Usage of dynamic arrays (stacks) for unlimited Undo/Redo history and unlimited **Dynamic Hash Array**, demonstrating proficiency in manual memory management (`malloc/free`). * **Cross-Platform Build:** A CMake-based build system that automatically fetches dependencies, ensuring the project compiles on Linux, Windows, and macOS without manual library configuration (**important note** we provide a Makefile that isn't made by Cmake or any other build system but it is made only for linux and you may need to install some dependencies but we tried to make the project as self contained as possible and included the library files in the project).

This report details the design decisions, algorithmic challenges, and implementation details of the Raylib Chess Engine.

---

2. Project Requirements & Scope

2.1 Core Requirements (Academic)

The original assignment mandated the following: 1. **Game Loop:** A continuous loop alternating between Player 1 (White) and Player 2 (Black). 2. **Move Input:** Parsing coordinate-based input (e.g., "E2E4"). 3. **Validation:** Ensuring moves follow piece-specific rules and do not endanger the King. 4. **Special States:** Detection of Check, Checkmate, and Stalemate. 5. **Save/Load:** Ability to serialize the game state to a file. 6. **Undo/Redo:** Ability to revert moves.

2.2 Extended Scope (Team Goals)

To exceed expectations and demonstrate mastery of C, we expanded the scope: 1. **GUI Implementation:** Replacing `printf/scanf` with a graphical window, mouse input, and texture rendering. 2. **Standard Compliance:** Using FEN strings instead of custom binary formats for saving. 3. **Visual Feedback:** Highlighting valid moves, the last move made,

and the King in check. 4. **Audio:** Integrating sound effects for tactile feedback. 5. **Safety:** Ensuring zero crashes via rigorous pointer checking and boundary validation.

---

3. System Architecture

3.1 High-Level Design

The application follows a variation of the **Model-View-Controller (MVC)** pattern, adapted for a game loop architecture.

- **Model (State):** The `GameState` struct and `GameBoard` array hold the absolute truth of the game (piece positions, turn, flags).
- **View (Draw):** The `draw.c` module reads the Model and renders it to the screen using Raylib. It is stateless; it simply draws what exists.
- **Controller (Main/Move):** The `main.c` loop captures input events. These events are passed to `move.c`, which validates them and updates the Model.

3.2 Module Breakdown

Module	File(s)	Responsibility
Core	`main.c`, `main.h`	Entry point, window initialization, main loop, event polling.
Logic	`move.c`, `move.h`	Move validation, execution, checkmate detection, special rules.
Render	`draw.c`, `draw.h`	Drawing board, pieces, highlights, and UI overlays.
IO	`save.c`, `load.c`	FEN string generation and parsing.
Data	`stack.c`, `stack.h`	Dynamic stack implementation for history.
Utils	`utils.c`, `utils.h`	High-level helpers (Restart, Load Game).
Hash	`hash.c`, `hash.h`	MD5 hashing for board state repetition detection.
Config	`settings.h`	Compile-time constants (screen size, colors, rules).

3.3 File Structure

The project is organized to separate source code, headers, and assets.

```
/chess
|-- CMakeLists.txt       # Build configuration
|-- Makefile             # Legacy build script
|-- README.md            # Documentation
|-- assets/              # PNG images and WAV sounds
|   |-- pawnW.png
|   |-- kingB.png
|   `-- ...
|-- includes/            # External library headers
|   |-- raygui.h
|   `-- style_amber.h
`-- src/                 # Source code
    |-- main.c
    |-- move.c
    |-- draw.c
    |-- save.c
    |-- load.c
    |-- stack.c
    |-- utils.c
    |-- hash.c
    `-- ... (corresponding .h files)
```

4. Data Structures & Memory Management

4.1 The Board Representation

Defined in `main.h`:

```c
typedef struct Cell
{
    Piece piece;                    /* piece occupying the cell (PIECE_NONE if empty) */
    Vector2 pos;                    /* pixel position for rendering (top-left) */
    unsigned int row : 5, col : 5; /* board coordinates (0..7) */
    bool primaryValid : 1;          // This is a primary validation geometrically
    bool isvalid : 1;               // Final validation of moves FINAL_VALIDATION
    bool selected : 1;              // will also need this
    bool vulnerable : 1;            // what pieces are under attack on my team this will
    bool hasMoved : 1;
    bool PawnMovedTwo : 1;
    bool JustMoved : 1;
    // Saved 8 bytes with these bitfields and it will also help us debug errors
} Cell;

// Global Board Instance
Cell GameBoard[8][8];
```

The board is represented as an 8x8 array of `Cell` structs. Each `Cell` contains:

This representation simplifies rendering and input handling, as each cell knows its position and state.

4.2 Dynamic History Stacks

The Undo/Redo functionality is powered by dynamic stacks, allowing for an arbitrary number of moves to be reverted or reapplied.

- **Push Operation:** Adds a new entry to the top of the stack.
- **Pop Operation:** Removes the top entry, returning the game state to the previous move.
- **Duplicate Prevention:** The stack implementation ensures no duplicate states are pushed, conserving memory.

The building blocks of the stack

```
typedef struct MoveStack
{
    Move *data; /*We will discuss the Move struct later because it's one of the best e
    size_t size;
    size_t capacity;

} MoveStack;
```

4.3 Global Game State

The `GameState` struct maintains all necessary information for the game logic:

```
typedef struct GameState
{
    //NOTE: this is a shortened version of the GameState but it's better for our repor

    Cell board[8][8];    // The chess board
    int turn;            // 0: White's turn, 1: Black's turn
    bool enPassant;      // En Passant flag
    int castling;        // Castling rights
    int halfMove;        // Half-move clock for draw conditions
    int fullMove;        // Full move counter
} GameState;
```

4.4 Memory Safety Strategy

To prevent memory leaks and ensure safety:

- **Manual Memory Management:** All dynamic allocations use `malloc`/`free`, with careful tracking of allocated pointers.

- **Pointer Validation:** Before dereferencing, all pointers are checked against `NULL`.
- **Buffer Overrun Prevention:** Array accesses are bounds-checked, and string operations use safe variants (`strncpy`).

4.5 Dynamic Hash Array

To enforce the **Threefold Repetition Rule** (a draw occurs if the exact same board position appears three times), the engine must store a history of all previous board states. Storing full copies of the `GameState` struct for every turn would be memory-inefficient and slow to compare.

Instead, we implemented a **Dynamic Hash Array** that stores 128-bit MD5 hashes of the board state.

Structure

Similar to the move stack, this array grows dynamically as the game progresses, ensuring we never run out of space for long games.

```
typedef struct Hash
{
    uint32_t data[4];
} Hash;

typedef struct HashArray
{
    Hash *data;
    size_t size;                // Current number of hashes stored
    size_t capacity;           // Current allocated capacity
} HashArray;
```

---

5. Algorithmic Implementation

5.1 Move Validation Pipeline

The move validation process is divided into several stages:

1. **Basic Checks:** Ensure the move is within the board limits and the source cell contains a piece of the correct color.
2. **Piece-Specific Validation:** Each piece type has its own validation function (e.g., `validatePawnMove`, `validateRookMove`).
3. **King Safety Check:** Ensure the move does not place the player's king in check.
4. **Update Simulation:** Temporarily apply the move to check for check/checkmate conditions.

5.2 Check & Checkmate Detection

The engine detects check and checkmate using a two-pronged approach:

1. **Attacking Pieces Calculation:**
   - For each enemy piece, calculate all possible attack vectors.
   - Store these vectors in a list for quick lookup.
2. **King Safety Verification:**
   - Before confirming any move, verify that the player's king is not within the attack vectors of any enemy pieces.
   - If the king is attacked, the move is invalid.

Checkmate is a subset of this logic:

- If the player is in check and has no valid moves to escape, it's checkmate.
- The engine exhaustively searches all possible moves using the current player's pieces. If no move results in the king being safe, checkmate is declared.

5.3 Special Moves Logic

Special moves like castling and en passant are handled as follows:

- **Castling:**
  - If the king and rook involved have not moved, and there are no pieces between them, castling is allowed.
  - The move is executed in one atomic operation, updating both the king and rook positions.
- **En Passant:**
  - If a pawn moves two squares forward from its starting position and lands beside an opponent's pawn, en passant is possible.
  - The opponent's pawn is captured as if it had moved one square forward.

5.4 FEN Serialization (Save/Load)

The Forsyth–Edwards Notation (FEN) is used for saving and loading game states:

- **Save Operation:**
  - The current game state is converted into a FEN string, representing piece positions, turn, castling rights, and en passant status.
  - The FEN string is then saved to a file.
- **Load Operation:**
  - A FEN string is read from a file and parsed.
  - The game state is updated to reflect the positions and status encoded in the FEN string.

5.5 State Hashing (MD5)

MD5 hashing is used to detect repeated states (threefold repetition rule):

- After every move, the current state is hashed, and the hash is compared against a history of previous hashes (that may be the same).
- If a match is found, the engine checks if the conditions for draw by repetition are met.

---

6. User Interface Design

6.1 Raylib Integration

Raylib is integrated as the primary graphics and input handling library:

- **Window Management:** Raylib handles the creation and management of the game window.
- **2D Rendering:** All game objects (board, pieces, UI elements) are rendered using Raylib's 2D drawing functions.
- **Input Handling:** Mouse and keyboard inputs are captured and processed by Raylib.

6.2 Input Handling

Input handling is centralized in the `main.c` file:

- **Mouse Input:** Raylib's `GetMouseButtonPressed` and `GetMousePosition` functions are used to detect mouse clicks and get the cursor position.
- **Keyboard Input:** Keyboard events are captured for shortcuts to improve usability:
  - `Ctrl + Z`: Undo last move.
  - `Ctrl + Shift + Z`: Redo last move.
  - `Ctrl + S`: Save the game.
  - `Ctrl + R`: Hide ranks and files.
  - `ESC`: Exit the game and close open popup windows.
  - `F5`: Show debug menu. ## 6.3 Visual Feedback Systems

To ensure a smooth user experience, the engine provides immediate visual feedback for every interaction. This is handled in the `draw.c` module, which overlays graphical elements on top of the board state.

- **Valid Move Indicators:** When a user selects a piece, the engine immediately calculates all pseudo-legal moves. Valid destination squares are marked with a semi-transparent gray circle. This helps beginners understand piece movement and prevents illegal move attempts.
- **Last Move Highlighting:** To help players track the opponent's action, the source and destination squares of the most recent move are highlighted in yellow and green accordingly. This persists until the next move is made.
- **Check Warning:** If a King is in check, a visual small alert will appear at the top of the board. This visual cue is critical, as it explains why certain moves (that don't resolve the check) are disabled.

6.4 Audio Feedback

Beyond visuals, the application integrates audio cues to provide tactile feedback for game events. We utilize Raylib's `Audio` module to load and play `.mp3` files asynchronously.

- **Move Sound:** A subtle "thud" plays on standard moves.
- **Capture Sound:** A distinct, sharper sound plays when a piece is removed from the board.
- **Check Sound:** An alert sound plays when a King is placed in check.
- **Checkmate:** it's said in human voice.

6.5 Adaptive Layout & Responsive Design

A key requirement for modern applications is the ability to handle different screen resolutions. Our engine does not rely on hardcoded pixel coordinates. Instead, it implements a **dynamic layout system**: . * **Board Centering:** The board's position is recalculated every frame based on the current window dimensions. This ensures that whether the window is 800x600 or 1920x1080, the game board remains perfectly centered.

- **Widget Anchoring:** UI elements such as the "Save", "Load", and "Undo" buttons are anchored relative to the board's position rather than the window's top-left corner. This guarantees that the controls always remain visually associated with the game area, preventing UI fragmentation on large monitors.

6.6 UI Theming System

To separate the application's logic from its aesthetic presentation, we utilized **Raygui's** styling system.

- **Style Abstraction:** Instead of hardcoding colors (e.g., `RED`, `BLUE`) into the drawing functions, the UI components reference a style definition file.
- **Amber Theme:** We integrated the `style_amber.h` header, which defines a cohesive color palette (warm oranges and dark grays) and font properties for all interactive elements.
-

  Extensibility: This architecture allows for "skinning" the application. Changing the entire look and feel of the game

7. Build System & Portability

The project uses CMake as the primary build system:

- **CMake Configuration:** A `CMakeLists.txt` file in the root directory configures the build, specifying source files, include directories, and linked libraries.

- **Dependency Management:** External dependencies (e.g., Raylib) are fetched and built automatically.
- **Multi-Platform Support:** The CMake configuration supports building on Windows, macOS, and Linux.

Legacy support for Makefile-based builds is also provided:

- A `Makefile` in the root directory allows building the project using `make` commands.
- This is primarily for environments where CMake is not available.

---

8. Testing & Debugging

Testing and debugging were integral to the development process:

- **Integration Testing:** Ensured that all modules (logic, rendering, input) work together seamlessly.
- **Debugging Tools:** Utilized GDB, Valgrind, ASAN and UBSAN for debugging and memory leak detection.

---

9. User Manual

9.1 Installation

To install the Raylib Chess Engine:

1. Clone the repository: `sh git clone https://github.com/OmarMohamed26/ChessProject chess cd chess`
2. Build using CMake: `sh mkdir build cd build cmake .. -DCMAKE_BUILD_TYPE=Rel make -j{core number}`
3. Run the game: `sh ./chess`

9.2 Controls

- **Mouse:**
  - Left Click: Select piece / Dismiss menu
- **Keyboard:**
  - `Ctrl+Z`: Undo
  - `Ctrl+Y`: Redo

---

10. Conclusion & Future Work

The Raylib Chess Engine project successfully met its objectives, delivering a robust and user-friendly chess application. Future enhancements could include:

- **AI Opponent:** Implementing a computer player using minimax algorithm with alpha-beta pruning.
- **Online Multiplayer:** Enabling play over the internet using a client-server architecture.
- **Advanced Graphics:** Enhancing visuals with shaders and more detailed sprites.

---

11. References

1. Raylib Documentation: raylib.com
2. FEN Specification: en.wikipedia.org/wiki/Forsyth–Edwards_Notation
3. MD5 Algorithm: en.wikipedia.org/wiki/MD5
4. Chess Programming Wiki: chessprogramming.org
5. chess.com chess.com

---

12. Appendices

Appendix A: Core Data Structures (`main.h`)

These structures form the backbone of the application state.

```
/* Represents a physical piece on the board */
typedef struct Piece
{
    PieceType type;     // Enum: PAWN, ROOK, KNIGHT, BISHOP, QUEEN, KING
    Team team;          // Enum: TEAM_WHITE, TEAM_BLACK
    Texture2D texture;  // Raylib texture handle
    bool hasMoved;      // Critical for Castling logic
} Piece;

/* Represents a single square on the 8x8 grid */
typedef struct Cell
{
    Piece piece;        // The piece occupying this cell (or PIECE_NONE)
    Rectangle rect;     // Screen coordinates for rendering/collision
    Color color;        // Board square color (Light/Dark)
    bool primaryValid;  // Highlight flag for valid move destinations
    bool secondaryValid;// Highlight flag for secondary interactions
} Cell;

/* Represents a move action */
typedef struct Move
{
    int srcRow, srcCol; // Source coordinates
    int dstRow, dstCol; // Destination coordinates
    Piece captured;     // Piece captured (if any), for Undo logic
    Piece moved;        // The piece that moved
    bool isCastling;    // Flag for special move handling
    bool isEnPassant;   // Flag for special move handling
    bool isPromotion;   // Flag for special move handling
    PieceType promotionType; // The type chosen during promotion
} Move;

/* Global Game State Container */
typedef struct GameState
{
    Team turn;

    // Castling Availability
    bool whiteKingSide;
```

```
    bool whiteQueenSide;
    bool blackKingSide;
    bool blackQueenSide;

    // En Passant Target
    int enPassantCol;   // Column index of pawn moved 2 squares, or -1

    // Game Termination Flags
    bool isCheckmate;
    bool isStalemate;
    bool isRepeated3times;
    bool isInsufficientMaterial;

    // Promotion State
    bool isPromoting;
    int promotionRow;
    int promotionCol;

    // History Stacks
    MoveStack *undoStack;
    MoveStack *redoStack;
} GameState;
```

Appendix B: Dynamic Stack Interface (stack.h)

```
/* Allocates and initializes a new stack with initial capacity */
MoveStack *InitializeStack(size_t initialMaximumCapacity);

/* Frees the stack structure and its internal data array */
void FreeStack(MoveStack *stack);

/* Pushes a move onto the stack, resizing (realloc) if necessary */
bool PushStack(MoveStack *stack, Move move);

/* Pops the top move from the stack into 'out' */
bool PopStack(MoveStack *stack, Move *out);

/* Returns the top move without removing it */
Move PeekStack(MoveStack *stack);

/* Returns true if the stack has no elements */
bool IsStackEmpty(MoveStack *stack);
```

Appendix C: Key Logic Prototypes (move.h)

```
/* Validates if a move from (r1,c1) to (r2,c2) is legal */
/* checkTurn: If true, ensures piece belongs to current player */
bool ValidateMove(int r1, int c1, int r2, int c2, bool checkTurn);

/* Executes a move on the board, updating state and flags */
void MakeMove(Move *move);

/* Checks if the King of the specified team is currently under attack */
bool IsKingInCheck(Team team);

/* Validates Castling moves specifically */
void PrimaryCastlingValidation();

/* Validates En Passant moves specifically */
void PrimaryEnpassantValidation(int row, int col);

/* Central routine to update game state (Checkmate, Stalemate, etc.) */
void ResetsAndValidations();
```