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

December 26, 2025

Alexandria University

Faculty of Engineering

Computer and Systems Engineering Department

Course: CS221: Computer Programming 1

Project: Chess Engine Implementation in C

Submission Date: December 26, 2025

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

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

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.

Module	File(s)	Responsibility
IO	`save.c`, `save.h`,`load.c`, `load.h`	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.
Colors	`colors.c`, `colors.h`	basic color data for all the game in one place for ease of change.
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
|-- REPORT.md            # The markdown to generate this report.
|-- .gitignore           # Necessary for useable VCS.
|-- tasks.todo           # Helps the team members know what have and haven't been done.
|-- assets/              # PNG images and mp3 sounds
|   |-- pieces/
|   |   |-- pawnW.png
|   |   |-- kingB.png
|   |   |-- ..
|   |-- sound/
|   |   |-- Capture.mp3
|   |   |-- Move.mp3
|   |   |-- ..
|   |
|   |-- icon.png
|   `-- ...
|-- includes/            # External library headers
|   |-- raygui.h
|   `-- raylib.h
|   `-- style_amber.h
|-- saves/
|   |-- youChooseTheName.fen
```

```
`-- 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`:

```
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

The user interface was designed with modern UX principles in mind, prioritizing clarity and ease of use. Unlike traditional console chess engines, our graphical interface provides a rich,

interactive experience. The board and pieces are rendered with high-quality assets, and the layout is designed to be responsive and centered, regardless of window size.

We focused on minimizing cognitive load by using visual cues for valid moves and game states. The color palette is chosen to be easy on the eyes during long play sessions, utilizing the "Amber" theme for a consistent look.

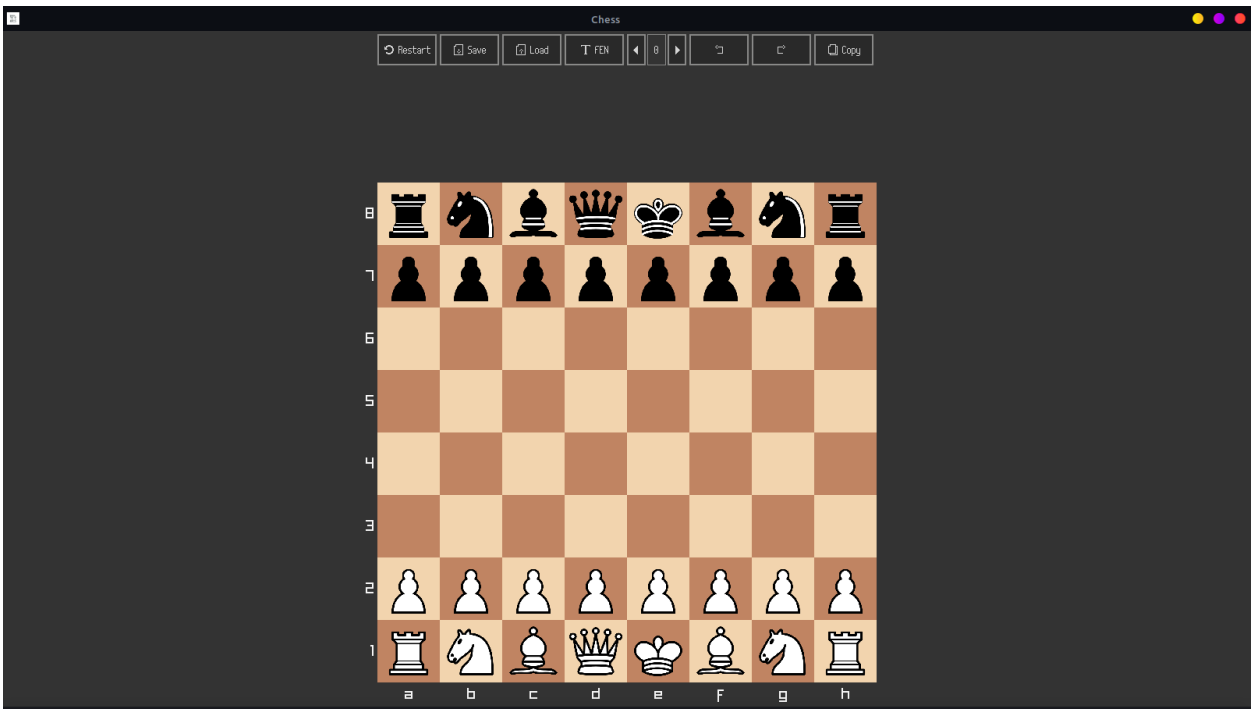

Figure 1: The main game interface showing the board and control panel.

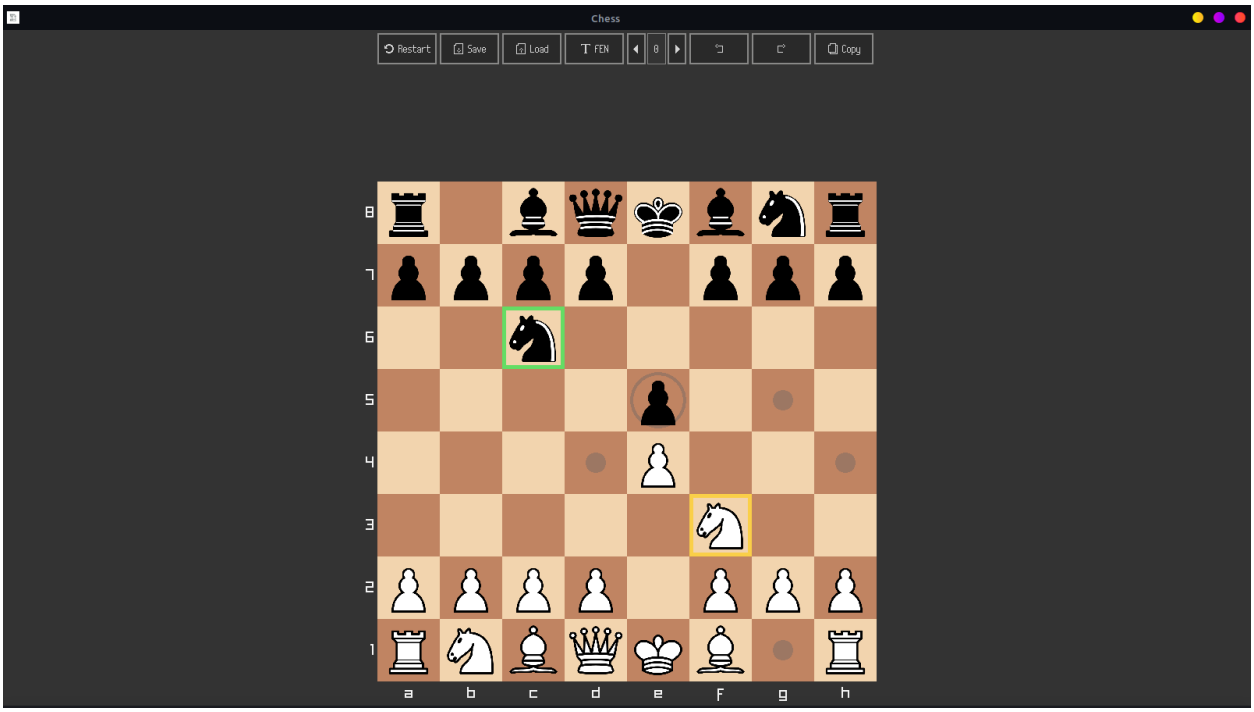

Figure 2: Visual feedback showing valid moves for a selected piece.

Figure 3: A small overlay that tells the player important Game changes.

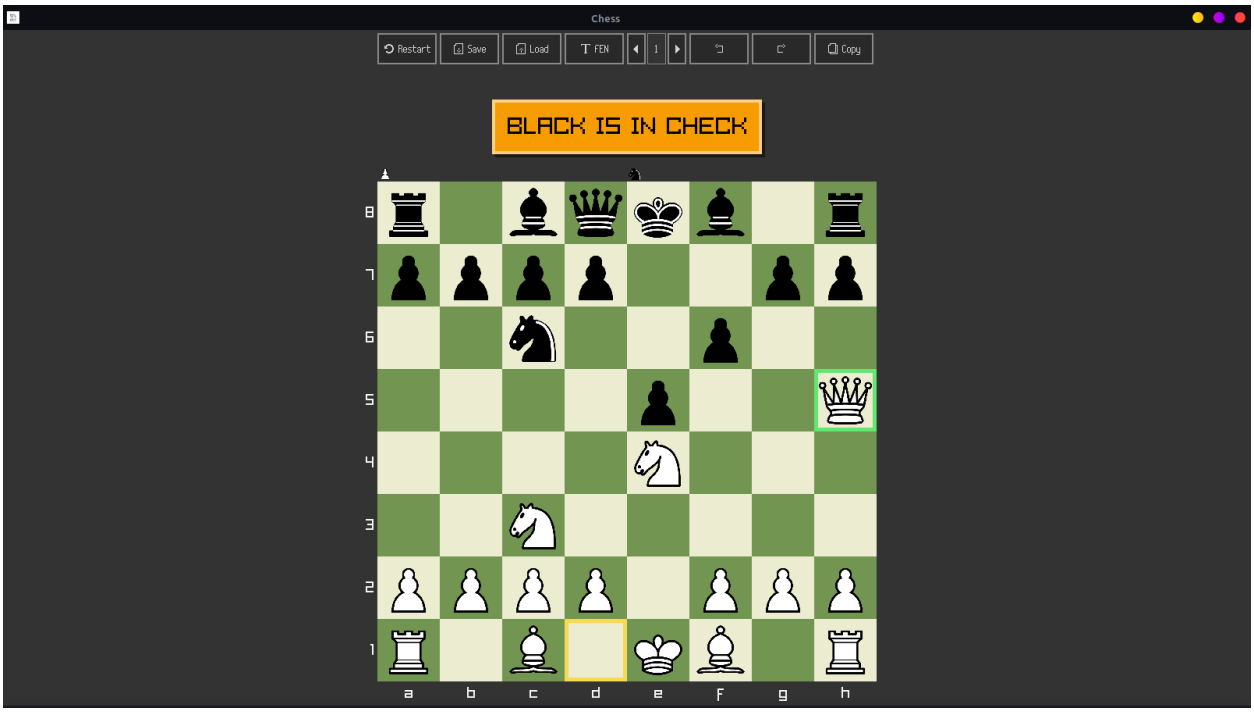

Figure 4: A variety of themes are available, allowing users to customize the visual experience to their preference.

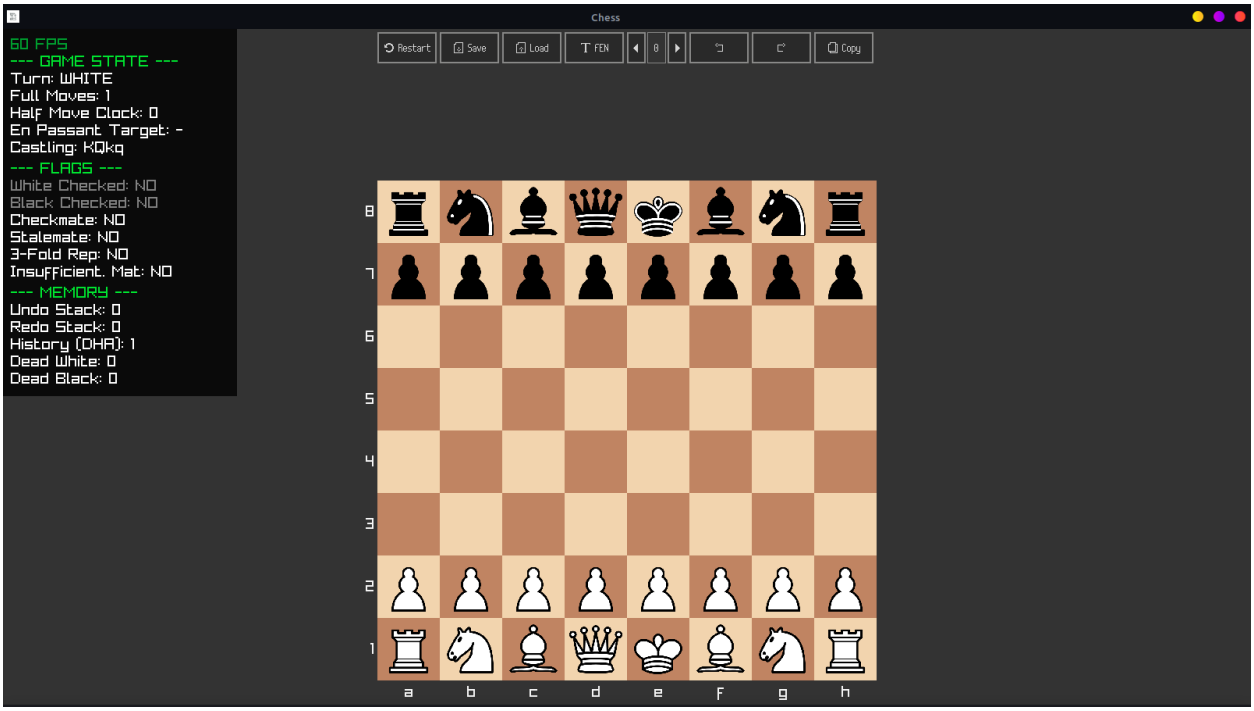

Figure 5: The debug menu.

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:** A voiceover announces "Checkmate" upon game completion.

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 is as simple as swapping the style header.

6.7 Modal Dialogs & Popups

To handle complex interactions without cluttering the main game screen, we implemented a custom modal dialog system. These popups overlay the game board, focusing user attention on specific tasks:

- **Save/Load Game:** A file dialog allows users to type filenames for saving or loading game states.
- **FEN Input:** A dedicated text input box lets users paste FEN strings to set up custom board positions instantly.
- **Promotion Selection:** When a pawn reaches the opposite end, a modal appears forcing the player to choose a promotion piece (Queen, Rook, Bishop, Knight) before the game proceeds.
- **Game Over:** A summary screen displays the result (Checkmate, Stalemate, Draw) and offers options to restart or exit.

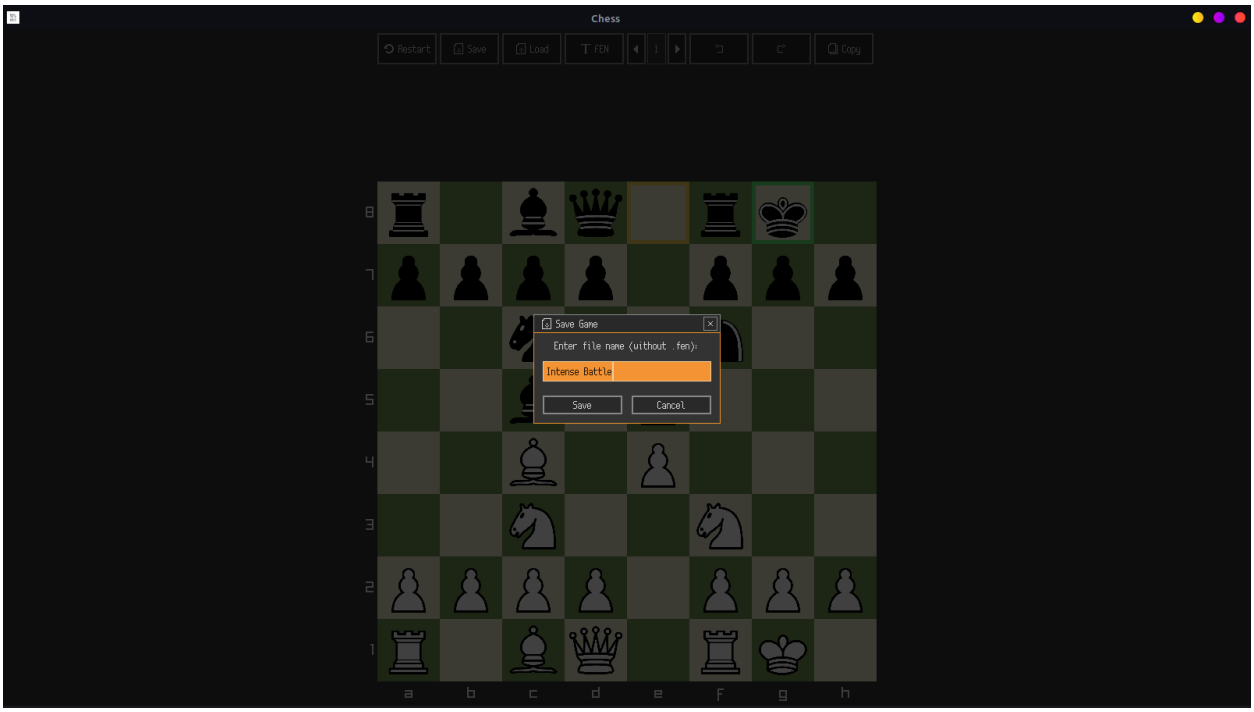

Figure 6: Save Game PopUp menu.

Figure 7: Overwrite Confirmation Dialog.

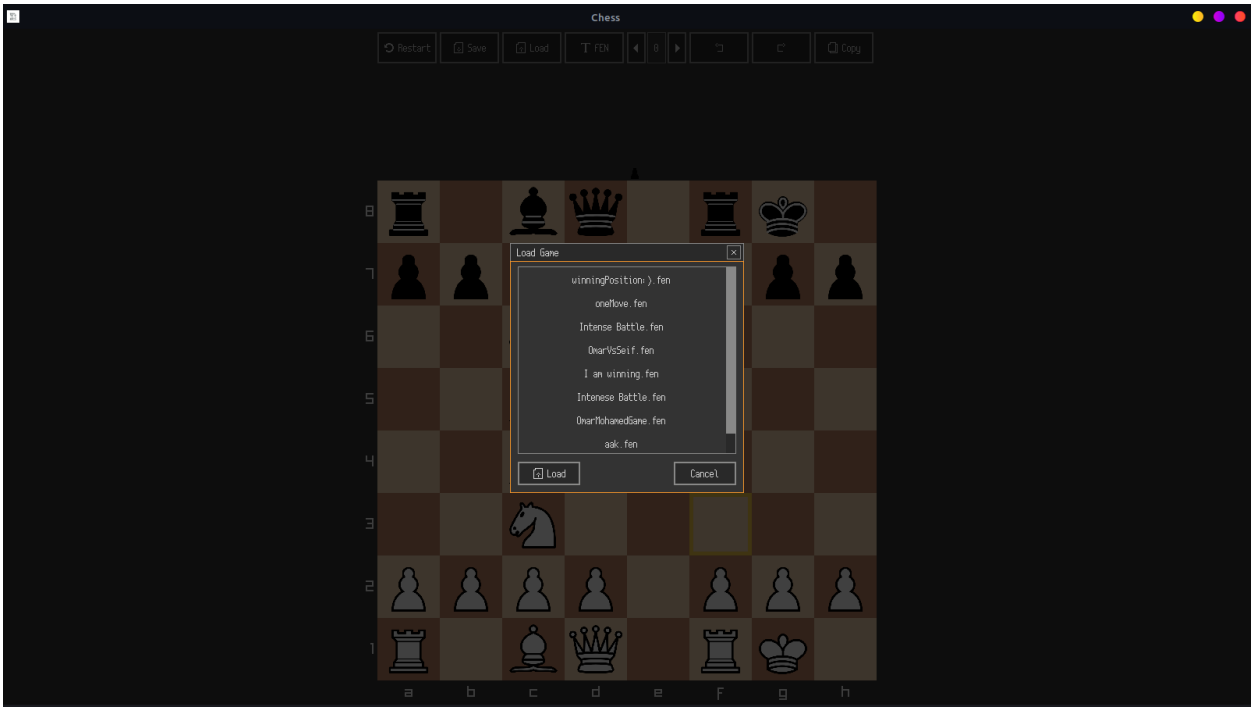

Figure 8: Load Game PopUp menu.

These dialogs block interaction with the underlying board until dismissed, ensuring state consistency.

(We will skip other dialogs because We don't want to take too long.)

7. Build System & Portability

The project uses **Make** as the primary build system and at the end we added **CMake** for other people to be able to test our work.

Makefile

- **Advanced Makefile:** it handles building automatically with different build modes and builds only what's necessary.

```
omar@Omar-PC:~/Project/chess$ make clean
Building in RELEASE mode (-O2, C17)...
Cleaning build directory...
rm -rf build

omar@Omar-PC:~/Project/chess$ make
Building in RELEASE mode (-O2, C17)...
gcc -O3 -s -Wall -Wextra -std=c17 -MMD -Isrc -Iincludes -c src/main.c -o build/Release/main.o
gcc -O3 -s -Wall -Wextra -std=c17 -MMD -Isrc -Iincludes -c src/draw.c -o build/Release/draw.o
gcc -O3 -s -Wall -Wextra -std=c17 -MMD -Isrc -Iincludes -c src/load.c -o build/Release/load.o
gcc -O3 -s -Wall -Wextra -std=c17 -MMD -Isrc -Iincludes -c src/save.c -o build/Release/save.o
gcc -O3 -s -Wall -Wextra -std=c17 -MMD -Isrc -Iincludes -c src/move.c -o build/Release/move.o
gcc -O3 -s -Wall -Wextra -std=c17 -MMD -Isrc -Iincludes -c src/colors.c -o build/Release/colors.o
gcc -O3 -s -Wall -Wextra -std=c17 -MMD -Isrc -Iincludes -c src/hash.c -o build/Release/hash.o
gcc -O3 -s -Wall -Wextra -std=c17 -MMD -Isrc -Iincludes -c src/stack.c -o build/Release/stack.o
gcc -O3 -s -Wall -Wextra -std=c17 -MMD -Isrc -Iincludes -c src/utils.c -o build/Release/utils.o
Linking build/chess...
gcc build/Release/main.o build/Release/draw.o build/Release/load.o build/Release/save.o build/Release/move.o build/
Release/colors.o build/Release/hash.o build/Release/stack.o build/Release/utils.o -o build/chess -Llibs -lraylib -lGL
-lm -lpthread -ldl -lrt -lX11 -Wl,-rpath=libs

omar@Omar-PC:~/Project/chess$ make debug
Building in DEBUG mode (-g -O0, C17, ASan, UBSan)...
gcc -g -O0 -DDEBUG -fsanitize=address,undefined -Wall -Wextra -std=c17 -MMD -Isrc -Iincludes -c src/main.c -o build/Debug/main.o
gcc -g -O0 -DDEBUG -fsanitize=address,undefined -Wall -Wextra -std=c17 -MMD -Isrc -Iincludes -c src/draw.c -o build/Debug/draw.o
gcc -g -O0 -DDEBUG -fsanitize=address,undefined -Wall -Wextra -std=c17 -MMD -Isrc -Iincludes -c src/load.c -o build/Debug/load.o
gcc -g -O0 -DDEBUG -fsanitize=address,undefined -Wall -Wextra -std=c17 -MMD -Isrc -Iincludes -c src/save.c -o build/Debug/save.o
gcc -g -O0 -DDEBUG -fsanitize=address,undefined -Wall -Wextra -std=c17 -MMD -Isrc -Iincludes -c src/move.c -o build/Debug/move.o
gcc -g -O0 -DDEBUG -fsanitize=address,undefined -Wall -Wextra -std=c17 -MMD -Isrc -Iincludes -c src/colors.c -o build/Debug/colors.o
gcc -g -O0 -DDEBUG -fsanitize=address,undefined -Wall -Wextra -std=c17 -MMD -Isrc -Iincludes -c src/hash.c -o build/Debug/hash.o
gcc -g -O0 -DDEBUG -fsanitize=address,undefined -Wall -Wextra -std=c17 -MMD -Isrc -Iincludes -c src/stack.c -o build/Debug/stack.o
gcc -g -O0 -DDEBUG -fsanitize=address,undefined -Wall -Wextra -std=c17 -MMD -Isrc -Iincludes -c src/utils.c -o build/Debug/utils.o
Linking build/debugChess...
gcc build/Debug/main.o build/Debug/draw.o build/Debug/load.o build/Debug/save.o build/Debug/move.o build/Debug/colors.o
build/Debug/hash.o build/Debug/stack.o build/Debug/utils.o -o build/debugChess -fsanitize=address,undefined -Llibs
-lraylib -lGL -lm -lpthread -ldl -lrt -lX11 -Wl,-rpath=libs
omar@Omar-PC:~/Project/chess$ make run
omar@Omar-PC:~/Project/chess$ make run-debug
```

Figure 9: Sophisticated build system with make to have different types of build and build the files that you need only.

CMake

- **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.

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.

Figure 10: Omar having fun time debugging the `RecordMove` *function.*

9. User Manual

9.1 Installation

To install the Raylib Chess Engine:

1. Clone the repository:

```
git clone https://github.com/OmarMohamed26/ChessProject chess

cd chess
```

2. Build using CMake:

   (If you want the debug build use cmake .. -DCMAKE_BUILD_TYPE=Debug)

   ```
   mkdir CMakeBuild

   cd CMakeBuild

   cmake .. -DCMAKE_BUILD_TYPE=Release

   make -j{number}
   ```

3. Run the game: *(or ./debugChess If you built in debug mode.)*

   ```
   ./chess
   ```

9.2 Controls

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

---

11. References

1. Raylib Documentation: **Raylib**
2. FEN Specification: Wikipedia Forsyth–Edwards_Notation
3. MD5 Algorithm: Wikipedia MD5
4. Chess Programming Wiki: chessprogramming
5. chess.com chess.com
6. **FIDE Laws of Chess: FIDE** rules

7. **Valgrind User Manual:** valgrind docs

---

12. Appendices

(Note: We don't list all of the functions here because that would be too much but only the most important ones)

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;
```

```c
/* 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)

```c
/* 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)

```
/**
 * move.h
 *
 * Responsibilities:
 * - Export functions for moving pieces, validating moves, and managing game state.
 * - Includes prototypes for move execution, validation, simulation, and undo/redo.
 */

/* Executes a move on the board, handling captures and special rules */
void MovePiece(int initialRow, int initialCol, int finalRow, int finalCol);

/* Clears a cell and unloads its texture */
void SetEmptyCell(Cell *cell);

/* Computes raw geometric moves for a piece (primary validation) */
void PrimaryValidation(PieceType Piece, int CellX, int CellY, bool selected);

/* Helper for geometric validation of specific piece types */
void MoveValidation(int CellX, int CellY, PieceType type, Team team, bool moved);

/* Filters primary moves to ensure they don't leave the King in check */
void FinalValidation(int CellX, int CellY, bool selected);

/* Scans all enemy pieces to mark squares they attack (vulnerable) */
void ScanEnemyMoves();

/* Checks if the current player's King is under attack */
void CheckValidation();

/* Checks if a simulated move results in the King being under attack */
void SimCheckValidation();

/* Helper for sliding pieces (Rook, Bishop, Queen) */
bool HandleLinearSquare(int row, int col, Team team);

/* Helper for Pawn movement logic */
void HandlePawnMove(int CellX, int CellY, Team team, bool moved);
```

```
/* Helper for Knight movement logic (single square) */
void HandleKnightSquare(int row, int col, Team team);

/* Helper for Knight movement logic (all 8 squares) */
void HandleKnightMove(int CellX, int CellY, Team team);

/* Helper for King movement logic */
void HandleKingMove(int CellX, int CellY, Team team);

/* Simulates a move on the board (without graphics/sound) */
void MoveSimulation(int CellX1, int CellY1, int CellX2, int CellY2, PieceType piece);

/* Reverts a simulated move */
void UndoSimulation(int CellX1, int CellY1, int CellX2, int CellY2, PieceType piece1, Pi

/* Checks if a player has any legal moves to escape check */
bool CheckmateFlagCheck(Team playerTeam);

/* Validates if the game is in Checkmate */
void CheckmateValidation();

/* Validates if the game is in Stalemate */
void StalemateValidation();

/* Promotes a pawn to the selected piece type */
void PromotePawn(PieceType selectedType);

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

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

/* Undoes the last move */
void UndoMove(void);

/*Redo the last move*/
void RedoMove(void);
```

Appendix D: Rendering Interface (draw.h)

The interface responsible for all graphical output, resource management, and layout calculations.

```
typedef enum LoadPlace
{
    GAME_BOARD = 0,
    DEAD_WHITE_PIECES,
    DEAD_BLACK_PIECES,
} LoadPlace;

/* Render board and pieces for the provided color theme index. */
void DrawBoard(int ColorTheme, bool showFileRank);

/* Load a piece texture for cell (row,col). squareLength selects texture size. */
void LoadPiece(int row, int col, PieceType type, Team team, LoadPlace place);

/* Initialize the chess board to have appropriate starting values */
void InitializeBoard(void);

/* Initialize the DeadPieces to have appropriate starting values */
void InitializeDeadPieces(void);

/* Run after the game finishes or you want a new game to prevent memory leaks and flus
void UnloadBoard(void);

/* Run after the game finishes or you want a new game to prevent memory leaks and flus
void UnloadDeadPieces(void);

/* Highlight a single square */
void HighlightSquare(int row, int col, int ColorTheme);

/* Highlights The piece when hovering over it */
void HighlightHover(int ColorTheme);

/* Compute the pixel size of a single board square using current render dimensions. */
int ComputeSquareLength(void);

/* Highlight valid moves for the selected piece */
void HighlightValidMoves(bool selected);

/* Updates the border highlight for the destination of the last move made. */
void UpdateLastMoveHighlight(int row, int col);

/* Renders an overlay with debug information */
void DrawDebugInfo(void);
```

Appendix E: Hashing capabilities to detect 3-fold repetition (hash.h)

The interface responsible for all graphical output, resource management, and layout calculations.

```c
/**
 * Hash
 *
 * Represents a 128-bit hash value (MD5 result).
 * Used to compare board states efficiently.
 */

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

/**
 * DynamicHashArray
 *
 * A resizable array container for Hash objects.
 * Used to store the history of all board positions in the current game.
 */
typedef struct
{
    Hash *hashArray; // Pointer to the heap-allocated array
    size_t size;     // Current number of elements
    size_t capacity; // Total allocated slots
} DynamicHashArray;

/* Allocates and initializes a new DynamicHashArray */
DynamicHashArray *InitializeDHA(size_t capacity);

/* Frees the array and the structure itself */
void FreeDHA(DynamicHashArray *DHA);

/* Resets the count to 0 (does not free memory) */
void ClearDHA(DynamicHashArray *DHA);

/* Checks if 'currentHash' exists at least twice in the history */
bool IsRepeated3times(DynamicHashArray *DHA, Hash currentHash);

/* Adds a hash to the history, expanding if necessary */
bool PushDHA(DynamicHashArray *DHA, Hash hash);
```

```
/* Removes the last hash (for undo operations) */
Hash PopDHA(DynamicHashArray *DHA);

/* Computes the hash of the current game state */
Hash CurrentGameStateHash(void);
```