Chess Game Project Report

Team Members: Yousaf Khawar Raja (24L-0552)
Date: May 14, 2025

Abstract

This project presents a fully functional chess game implemented in C++ using the SFML library for graphics and audio. The game allows two players to play chess with a graphical interface, supporting piece movements, pawn promotion, check/checkmate detection, and a timer for each player. The purpose is to provide an interactive and user-friendly chess experience while demonstrating object-oriented programming (OOP) principles. Key OOP concepts applied include inheritance for defining chess pieces, polymorphism for handling piece movements, composition for managing the game and board, and aggregation for associating pieces with the board. The project showcases how these concepts enhance code modularity, reusability, and maintainability.

Table of Contents

Introduction

Problem Statement

Traditional chess games often lack engaging digital interfaces for casual players, and many existing implementations do not fully leverage OOP principles to ensure maintainable and scalable code. This project aims to address this by developing a digital chess game that combines an intuitive graphical interface with a robust, OOP-based codebase.

Objectives

- Develop a fully functional chess game with a graphical user interface using SFML.
- Implement core chess rules, including piece movements, pawn promotion, and check/checkmate detection.
- Incorporate timers and sound effects to enhance user experience.
- Apply OOP concepts to ensure the code is modular, reusable, and easy to maintain.

Motivations

Chess is a timeless strategy game that benefits from digital adaptations for accessibility and engagement. The motivation behind this project was to create an interactive chess experience while exploring the practical application of OOP principles in game development. Using C++ and SFML allowed for efficient handling of graphics and audio, making the game both functional and enjoyable.

OOP Concepts Used

Inheritance

Inheritance was used to create a hierarchy of chess pieces. The ChessPiece base class defines a virtual move() method, which is overridden by derived classes like ChessPawn, ChessRook, ChessBishop, ChessQueen, and ChessKing. This allows each piece to implement its specific movement rules while sharing a common interface, reducing code duplication and improving maintainability.

Polymorphism

Polymorphism is implemented through the virtual move() method in the ChessPiece class. The demonstratePieceMoves() function uses base class pointers to call the appropriate move() method of derived classes, enabling dynamic behavior based on the piece type. This makes the code flexible and easier to extend with new piece types.

Composition

Composition is evident in the ChessGame class, which contains a ChessBoard object. The ChessGame class manages the game logic and relies on the ChessBoard to handle the state of the board. This strong ownership ensures that the board is tightly integrated into the game, and its lifecycle is tied to the game instance.

Aggregation

Aggregation is used in the ChessBoard class, which holds an array of pointers to ChessPiece objects. The board does not own these pieces (they can exist independently), but it references them to represent their positions. This loose coupling allows pieces to be manipulated independently of the board while still being part of the game structure.

Class Diagrams

Chess Game Class Diagram - 04:44 PM PKT on Wednesday, May 14, 2025

The UML class diagram illustrates the architecture of the chess game:

- **ChessPiece (Abstract Class)**:
    - **Public Methods**: virtual void move(), virtual ~ChessPiece()
- **ChessPawn, ChessRook, ChessBishop, ChessQueen, ChessKing**: Inherit from ChessPiece.
    - **Public Methods**: void move() (overridden for each piece).
- **ChessBoard**:
    - **Private Members**: ChessPiece* grid[8][8]
    - **Public Methods**: ChessBoard(), void placePiece(int x, int y, ChessPiece* piece), void displayBoard() const
    - **Relationship**: Aggregation with ChessPiece (contains pointers to ChessPiece objects).
- **ChessGame**:
    - **Private Members**: ChessBoard chessBoard
    - **Public Methods**: void startGame()
    - **Relationship**: Composition with ChessBoard (owns a ChessBoard instance).
- **Button**:

- **Private Members**: RectangleShape m_shape, Text m_text, Vector2f m_position
- **Public Methods**: Button(const Vector2f& position, const string& text, Font& font), void draw(RenderWindow& window), bool isMouseOver(const RenderWindow& window) const

This structure highlights inheritance for piece types, aggregation for the board-piece relationship, and composition for the game-board relationship.

Test Cases

- **Test Case 1: Pawn Movement (White Pawn)**
  - **Input**: Select white pawn at (6, 0), move to (5, 0).
  - **Expected Output**: Pawn moves to (5, 0), board updates, and it's black's turn.
  - **Result**: Pass (pawn moves forward one square as expected).
- **Test Case 2: Pawn Promotion (White Pawn)**
  - **Input**: Move white pawn from (1, 0) to (0, 0), simulate reaching the last rank.
  - **Expected Output**: Promotion window opens, user selects "Queen," pawn at (0, 0) becomes a white queen.
  - **Result**: Pass (promotion window appears, and pawn is replaced with the chosen piece).
- **Test Case 3: Check Detection (White King)**
  - **Input**: Move black queen to (4, 4), white king at (5, 4).
  - **Expected Output**: Game detects check on white king, restricts moves to only the king.
  - **Result**: Pass (check is detected, and piece movement is restricted).
- **Test Case 4: Checkmate Detection (Black King)**
  - **Input**: Simulate a checkmate position (e.g., white queen at (6, 6), black king at (7, 7) with no escape).
  - **Expected Output**: "White Won" window appears, game ends.
  - **Result**: Pass (checkmate is detected, and victory screen is shown).

- **Test Case 5: Timer Expiration (White Player)**
    - **Input**: Let white player's timer run out (300 seconds).
    - **Expected Output**: "Black Won" window appears, game ends.
    - **Result**: Pass (timer expiration triggers the correct victory condition).

Future Directions

- **AI Opponent**: Implement an AI opponent using algorithms like Minimax with alpha-beta pruning to allow single-player mode.
- **Online Multiplayer**: Add network functionality to enable online play between two players over the internet.
- **Undo/Redo Moves**: Introduce a move history feature to allow players to undo or redo moves during the game.
- **Customizable Themes**: Enable players to customize the board and piece themes for a personalized experience.
- **Move Validation Enhancements**: Add support for special moves like castling and en passant to fully comply with chess rules.