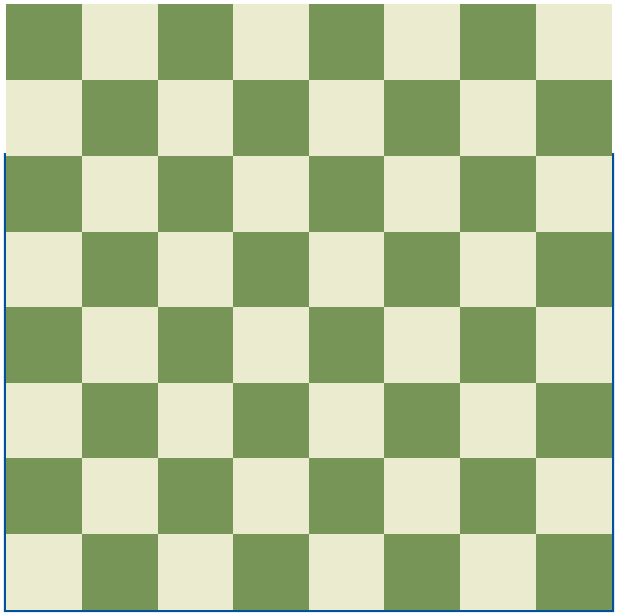
SFML Chess Game

Technical Documentation & Implementation Guide



A Comprehensive C++ Implementation

Using SFML Graphics Library

Version 1.0

January 23, 2026

Language:	C++17
Framework:	SFML 2.6.2
Platform:	Cross-platform (Windows/Linux/macOS)
Lines of Code:	~1,240

Contents

Executive Summary

This document provides comprehensive technical documentation for a fully-featured chess game implementation developed in C++ using the Simple and Fast Multimedia Library (SFML). The project demonstrates advanced software engineering principles including modular architecture, object-oriented design patterns, file I/O operations, real-time graphics rendering, and comprehensive game state management.

Project Overview

The SFML Chess Game is a complete implementation of the classical chess game with a graphical user interface, supporting all standard chess rules including move validation, check detection, checkmate recognition, and stalemate conditions. The application features a robust menu system, game state persistence, statistics tracking, and customizable settings.

Key Features

- **Complete Chess Rules:** Implementation of all piece movements, capture mechanics, check/checkmate/stalemate detection
- **Graphical Interface:** Custom SFML-based rendering with sprite management and visual feedback
- **Save/Load System:** Persistent game state storage with automatic recovery
- **Statistics Tracking:** Win counter for both players with persistent storage
- **Menu System:** Comprehensive navigation including main menu, pause menu, settings, and end-game screens
- **Modular Architecture:** 10 distinct modules with clear separation of concerns

Technical Specifications

Attribute	Value
Programming Language	C++17
Graphics Library	SFML 2.6.2
Total Source Files	20 (.cpp and .h files)
Lines of Code	Approximately 1,240
Supported Platforms	Windows, Linux, macOS
Memory Footprint	< 50 MB
Minimum Display Resolution	800×800 pixels

Table 1: Technical specifications of the chess game project

Document Structure

This documentation is organized into the following chapters:

1. **Introduction:** Project background, objectives, and scope
2. **System Architecture:** High-level design and module organization
3. **Installation & Compilation:** Step-by-step setup for all platforms
4. **Module Documentation:** Detailed analysis of each component
5. **Game Logic Implementation:** Core algorithms and data structures
6. **User Interface:** GUI design and interaction patterns
7. **Testing & Quality Assurance:** Validation strategies
8. **Limitations & Future Work:** Current constraints and enhancement opportunities

Chapter 1

Introduction

1.1 Project Background

Chess, one of the oldest and most intellectually challenging board games, has been a staple of computer science education and game development for decades. This project implements a fully-functional chess game using modern C++ practices and the SFML graphics library, demonstrating the application of software engineering principles to game development.

1.1.1 Motivation

The primary motivations for this project include:

- **Educational Value:** Demonstrating modular software design in a real-world application
- **Game Development Skills:** Implementing complex game logic with graphical rendering
- **Algorithm Implementation:** Applying computational thinking to chess rule validation
- **User Experience Design:** Creating an intuitive and responsive interface

1.2 Project Objectives

The core objectives of this chess game implementation are:

1. **Rule Completeness:** Implement all standard chess rules including piece-specific movements, capture mechanics, and special conditions (check, checkmate, stalemate)
2. **Visual Quality:** Provide a polished graphical interface with custom sprites, board rendering, and visual feedback mechanisms

3. **Persistence:** Enable game state saving and loading, allowing players to resume interrupted games
4. **Modularity:** Design a clean, maintainable codebase with clear separation of concerns across multiple modules
5. **Cross-Platform Compatibility:** Ensure the application runs on Windows, Linux, and macOS with minimal platform-specific code
6. **Extensibility:** Structure the codebase to facilitate future enhancements such as AI opponents, network play, and advanced chess features

1.3 Technology Stack

1.3.1 Programming Language: C++17

C++ was selected for this project due to its:

- **Performance:** Compiled nature and low-level control enable efficient game loop execution
- **SFML Compatibility:** Native integration with the SFML library
- **Object-Oriented Features:** Support for encapsulation, inheritance, and polymorphism
- **Standard Library:** Comprehensive STL for data structures and algorithms

1.3.2 Graphics Library: SFML 2.6.2

Simple and Fast Multimedia Library (SFML) provides:

- **Graphics Module:** Hardware-accelerated 2D rendering with sprite management
- **Window Module:** Cross-platform window creation and event handling
- **System Module:** Time management and threading capabilities
- **Audio Module:** Sound effect playback (prepared for future implementation)

Why SFML?

SFML was chosen over alternatives (SDL, Allegro, Qt) because:

- Simple, intuitive API requiring minimal boilerplate code
- Excellent documentation and community support
- Cross-platform compatibility without platform-specific code
- Active development and modern C++ integration

- Lightweight footprint with modular design

1.4 Scope of Implementation

1.4.1 Included Features

This implementation includes:

- All standard chess piece movements (Pawn, Rook, Knight, Bishop, Queen, King)
- Pawn special moves: double-step from starting position, diagonal captures
- Path validation for sliding pieces (Bishop, Rook, Queen)
- Check detection for both players
- Checkmate and stalemate recognition
- Legal move verification preventing self-check
- Graphical user interface with sprite-based rendering
- Board highlighting for piece selection
- Complete menu system (main, pause, settings, end-game)
- Game state persistence (save/load functionality)
- Win statistics tracking
- User preference management (sound, theme)

1.4.2 Excluded Features

The following chess features are not currently implemented:

- **En Passant:** Special pawn capture rule
- **Castling:** King-Rook cooperative move
- **Pawn Promotion:** Upgrading pawns reaching the opposite end
- **Draw by Repetition:** Three-fold repetition rule
- **50-Move Rule:** Automatic draw condition
- **AI Opponent:** Computer player with strategic algorithms
- **Network Multiplayer:** Online play capability
- **Move History:** Algebraic notation display
- **Undo/Redo:** Move reversal functionality
- **Timer/Clock:** Timed game modes

1.5 Target Audience

This project is designed for:

- **Computer Science Students:** Learning game development and software architecture
- **Game Developers:** Studying chess rule implementation and SFML integration
- **C++ Programmers:** Examining modular design patterns and best practices
- **Chess Enthusiasts:** Playing casual chess games with a clean interface

Chapter 2

System Architecture

2.1 Design Philosophy

The chess game architecture follows fundamental software engineering principles:

Modularity Each component serves a single, well-defined purpose

Separation of Concerns Game logic, rendering, and user interface are isolated

Data-Driven Design Board state is centralized and accessed through clear interfaces

Extensibility New features can be added without extensive refactoring

2.2 High-Level Architecture

The system is organized into three primary layers:

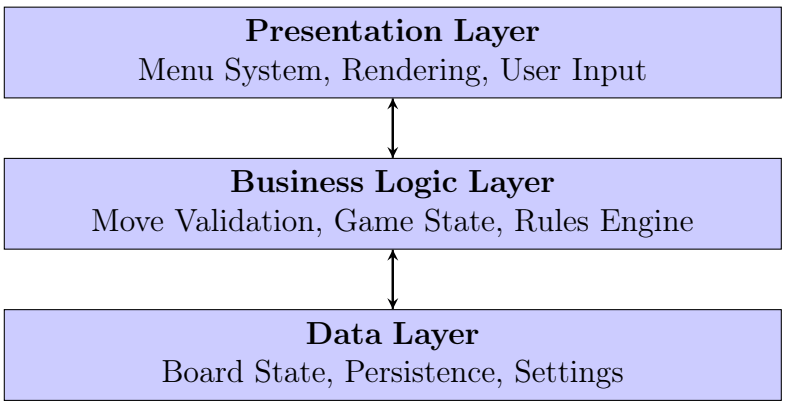


Figure 2.1: Three-tier architecture of the chess game

2.3 Module Organization

The project consists of 10 distinct modules, each with specific responsibilities:

Module	Files	Responsibility
Board Management	board.cpp/h	Global state, initialization
Move Validation	moves.cpp/h	Piece-specific move rules
Check Detection	checkmate.cpp/h	Check, checkmate, stalemate
Rendering	render.cpp/h	Graphics, sprites, drawing
Menu System	menu.cpp/h	UI navigation, screens
End Game	ending.cpp/h	Victory/draw screens
Persistence	save.cpp/h	File I/O, game state
Statistics	highscores.cpp/h	Win tracking
Settings	settings.cpp/h	User preferences
Main Controller	main.cpp	Event loop, coordination

Table 2.1: Module organization and responsibilities

2.3.1 Dependency Graph

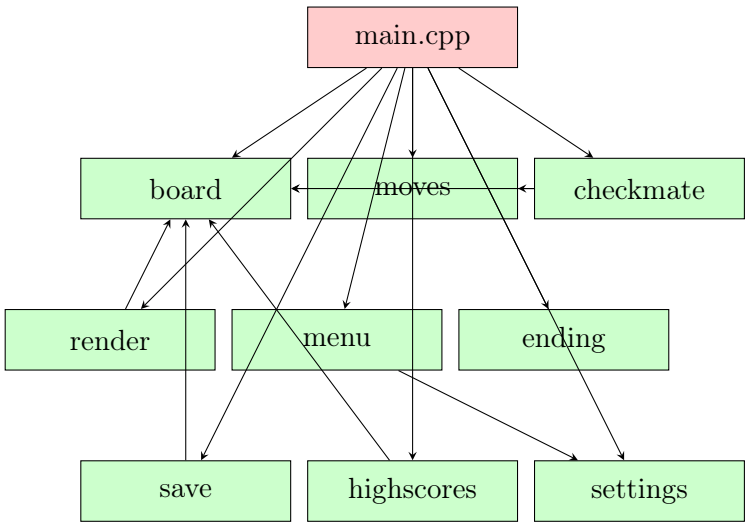


Figure 2.2: Module dependency relationships

2.4 Data Structures

2.4.1 Board Representation

The chess board is represented as a 2D character array:

```
extern char board[8][8];  // Global board state
extern char turn;         // Current player: 'w' or 'b'
extern int whiteWins;     // White victories
extern int blackWins;     // Black victories
```

Listing 2.1: Board data structure

Encoding Scheme:

- Uppercase letters represent White pieces: `P R N B Q K`
- Lowercase letters represent Black pieces: `p r n b q k`
- Space character (`' '`) represents empty squares

Coordinate System:

- Row 0: Black's back rank (top of visual display)
- Row 7: White's back rank (bottom of visual display)
- Column 0-7: Files a-h (left to right)

Why Character Array?

A character array was chosen over alternatives (bitboards, object-oriented pieces) because:

- Simplicity: Easy to visualize and debug
- Memory efficiency: Only 64 bytes for the entire board
- Fast access: O(1) lookup for any square
- Straightforward serialization: Direct file I/O

2.4.2 Initial Board Configuration

```
void initializeBoard() {
    char temp[8][8] = {
        {'r','n','b','q','k','b','n','r'},  // Black pieces
        {'p','p','p','p','p','p','p','p'},  // Black pawns
        {' ',' ',' ',' ',' ',' ',' ',' '},
        {' ',' ',' ',' ',' ',' ',' ',' '},
        {' ',' ',' ',' ',' ',' ',' ',' '},
        {' ',' ',' ',' ',' ',' ',' ',' '},
        {'P','P','P','P','P','P','P','P'},  // White pawns
        {'R','N','B','Q','K','B','N','R'}   // White pieces
    };
    // Copy to global board array
    for (int i = 0; i < 8; ++i)
        for (int j = 0; j < 8; ++j)
            board[i][j] = temp[i][j];
    turn = 'w';  // White moves first
}
```

Listing 2.2: Board initialization

2.5 Control Flow

2.5.1 Main Game Loop

The application follows a standard game loop pattern:

Algorithm 1 Main Game Loop

```
Initialize SFML window
Load textures and resources
Load settings and highscores
while window is open do
    Display main menu
    Process menu selection
    if start new game or load game then
        Initialize/restore board state
        while in game do
            while event exists do
                Handle user input (mouse, keyboard)
                if valid move selected then
                    Validate move legality
                    Check for self-check
                    if move is legal then
                        Execute move
                        Toggle turn
                        Check for checkmate/stalemate
                        if game ended then
                            Display end screen
                            Record statistics
                        end if
                    end if
                end if
            end while
            Render board and pieces
            Display window
        end while
    end if
end while
Save settings and statistics
Close application
```

2.5.2 Event Handling Sequence

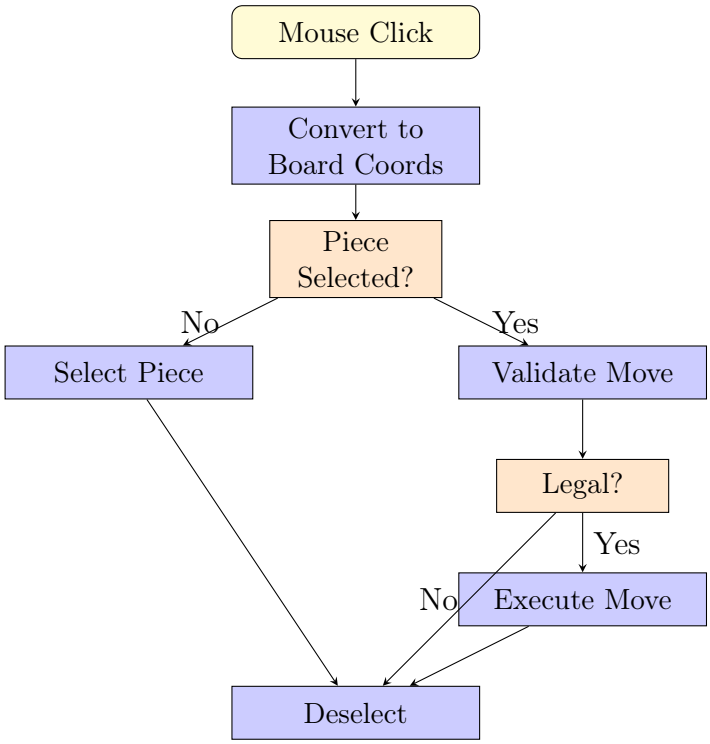


Figure 2.3: Event handling flow for mouse clicks

Chapter 3

Installation & Compilation

3.1 Prerequisites

3.1.1 System Requirements

Component	Requirement
Operating System	Windows 7+, Linux (any modern), macOS 10.12+
RAM	256 MB minimum
Disk Space	50 MB (including assets)
Display	800×800 minimum resolution
Compiler	GCC 7+, Clang 5+, or MSVC 2017+

Table 3.1: Minimum system requirements

3.1.2 Required Software

C++ Compiler

- **Windows:** MinGW-w64 (GCC) or Microsoft Visual C++ (MSVC)
- **Linux:** GCC or Clang (typically pre-installed)
- **macOS:** Clang (via Xcode Command Line Tools)

SFML 2.6.2

Download from: https://www.sfml-dev.org/download.php
Required modules:

- `sfml-graphics`
- `sfml-window`
- `sfml-system`
- `sfml-audio`

3.2 Windows Installation

3.2.1 Step 1: Install MinGW-w64

1. Download MinGW-w64 from https://www.mingw-w64.org/
2. Run the installer and select:
   - Architecture: x86_64
   - Threads: posix
   - Exception: seh
3. Add MinGW `bin` directory to system PATH

PATH Configuration

Ensure MinGW's `bin` directory is in your PATH:

`C:\mingw-w64\x86_64-8.1.0-posix-seh-rt_v6-rev0\mingw64\bin`

Verify with: `g++ -version`

3.2.2 Step 2: Install SFML

1. Download SFML 2.6.2 for MinGW (GCC 7.3.0)
2. Extract to `C:\SFML-2.6.2\`
3. Add SFML `bin` directory to PATH:

   `C:\SFML-2.6.2\bin`

3.2.3 Step 3: Compile the Project

```
g++ main.cpp board.cpp moves.cpp render.cpp menu.cpp ^
    checkmate.cpp save.cpp highscores.cpp settings.cpp ^
    ending.cpp -o chess_game.exe ^
    -I"C:/SFML-2.6.2/include" ^
    -L"C:/SFML-2.6.2/lib" ^
    -lsfml-graphics -lsfml-window -lsfml-system -lsfml-audio \
    -std=c++17 -Wall
```

Listing 3.1: Windows compilation command

3.2.4 Step 4: Run the Game

```
./chess_game
```

3.3 macOS Installation

3.3.1 Step 1: Install Xcode Command Line Tools

```
xcode-select --install
```

3.3.2 Step 2: Install SFML via Homebrew

```
brew install sfml
```

3.3.3 Step 3: Compile the Project

```
g++ main.cpp board.cpp moves.cpp render.cpp menu.cpp \
    checkmate.cpp save.cpp highscores.cpp settings.cpp \
    ending.cpp -o chess_game \
    -I/usr/local/include \
    -L/usr/local/lib \
    -lsfml-graphics -lsfml-window -lsfml-system -lsfml-audio \
    -std=c++17 -Wall
```

Listing 3.2: macOS compilation command

3.3.4 Step 4: Run the Game

```
./chess_game
```

3.4 Using a Makefile

For automated builds across all platforms, create a *Makefile*:

```
# Compiler and flags
CXX = g++
CXXFLAGS = -std=c++17 -Wall -Wextra -O2

# Platform detection
UNAME_S := $(shell uname -s)

# SFML configuration
ifeq ($(UNAME_S),Linux)
    SFML_FLAGS = -lsfml-graphics -lsfml-window -lsfml-system -lsfml
        -audio
endif
ifeq ($(UNAME_S),Darwin)
    SFML_INCLUDE = -I/usr/local/include
```

```
    SFML_LIB = -L/usr/local/lib
    SFML_FLAGS = -lsfml-graphics -lsfml-window -lsfml-system -lsfml-audio
endif
ifeq ($(OS),Windows_NT)
    SFML_INCLUDE = -IC:/SFML-2.6.2/include
    SFML_LIB = -LC:/SFML-2.6.2/lib
    SFML_FLAGS = -lsfml-graphics -lsfml-window -lsfml-system -lsfml-audio
    TARGET = chess_game.exe
else
    TARGET = chess_game
endif

# Source files
SOURCES = main.cpp board.cpp moves.cpp render.cpp menu.cpp \
          checkmate.cpp save.cpp highscores.cpp settings.cpp ending.cpp

# Object files
OBJECTS = $(SOURCES:.cpp=.o)

# Build target
$(TARGET): $(OBJECTS)
    $(CXX) $(CXXFLAGS) $(OBJECTS) -o $(TARGET) $(SFML_LIB) $(SFML_FLAGS)

# Compile source files
%.o: %.cpp
    $(CXX) $(CXXFLAGS) $(SFML_INCLUDE) -c $< -o $@

# Clean build files
clean:
    rm -f $(OBJECTS) $(TARGET)

# Run the game
run: $(TARGET)
    ./$(TARGET)

.PHONY: clean run
```

Listing 3.3: Cross-platform Makefile

Usage:

```
make          # Compile the project
make run      # Compile and run
make clean    # Remove build artifacts
```

3.5 Project Directory Structure

Ensure your project directory is organized as follows:

```
chess_game/
 main.cpp
 board.cpp / board.h
 moves.cpp / moves.h
 checkmate.cpp / checkmate.h
 render.cpp / render.h
 menu.cpp / menu.h
 ending.cpp / ending.h
 save.cpp / save.h
 highscores.cpp / highscores.h
 settings.cpp / settings.h
 assets/
    board.png
    pieces/
        white_pawn.png
        white_rook.png
        white_knight.png
        white_bishop.png
        white_queen.png
        white_king.png
        black_pawn.png
        black_rook.png
        black_knight.png
        black_bishop.png
        black_queen.png
        black_king.png
 Makefile (optional)
 README.md
```

Asset Requirements

The game will not run without the required asset files:

- 12 piece sprite PNG files (100×100 px recommended)
- 1 board texture PNG file (800×800 px recommended)
- All assets must be in `assets/pieces/` directory
- File names must match exactly (case-sensitive on Linux/macOS)

3.6 Troubleshooting

3.6.1 Common Compilation Errors

Error: "Failed to load font"

Cause: Arial font not found at specified path
Solution:

- **Windows:** Verify path is `C:/Windows/Fonts/arial.ttf`
- **Linux:** Change to `/usr/share/fonts/truetype/liberation/LiberationSans-Regular.ttf`
- **macOS:** Change to `/System/Library/Fonts/Helvetica.ttc`

Error: "Undefined reference to sf::..."

Cause: SFML libraries not linked properly
Solution:

- Ensure all four SFML libraries are specified: `-lsfml-graphics -lsfml-window -lsfml-system -lsfml-audio`
- Verify library path with `-L` flag
- Check SFML installation with: `pkg-config -libs sfml-all`

Error: "Failed to load textures"

Cause: Asset files missing or incorrect path
Solution:

- Ensure `assets/pieces/` directory exists
- Verify all 12 PNG files are present
- Check file names match exactly (case-sensitive)
- Confirm PNG files are valid images

Chapter 4

Module Documentation

4.1 Board Management Module

4.1.1 Files: board.cpp / board.h

Purpose: Central data management for game state

Global Variables

```
extern char board[8][8];   // Chess board representation
extern char turn;          // Current player: 'w' or 'b'
extern int whiteWins;      // White victory counter
extern int blackWins;      // Black victory counter
```

Listing 4.1: Board module declarations

Functions

`void initializeBoard()`
Initializes the chess board to standard starting position.
Algorithm:

1. Create temporary 8×8 array with initial piece positions
2. Copy to global `board` array
3. Set `turn` to 'w' (White moves first)

Time Complexity: O(1) - constant 64 array assignments
Space Complexity: O(1) - fixed 64-byte temporary array

```
void initializeBoard() {
    char temp[8][8] = {
        {'r','n','b','q','k','b','n','r'},
        {'p','p','p','p','p','p','p','p'},
        {' ',' ',' ',' ',' ',' ',' ',' '},
        {' ',' ',' ',' ',' ',' ',' ',' '},
```

```
        {' ',' ',' ',' ',' ',' ',' ',' '},
        {' ',' ',' ',' ',' ',' ',' ',' '},
        {'P','P','P','P','P','P','P','P'},
        {'R','N','B','Q','K','B','N','R'}
    };
    for (int i = 0; i < 8; ++i)
        for (int j = 0; j < 8; ++j)
            board[i][j] = temp[i][j];
    turn = 'w';
}
```

Listing 4.2: Board initialization implementation

`void printBoardConsole()`
Debugging utility that prints current board state to console.
Output Format:

```
Turn: White
r n b q k b n r
p p p p p p p p
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
P P P P P P P P
R N B Q K B N R
```

4.1.2 Board Encoding Reference

Char	Piece	Char	Piece
P	White Pawn	p	Black Pawn
R	White Rook	r	Black Rook
N	White Knight	n	Black Knight
B	White Bishop	b	Black Bishop
Q	White Queen	q	Black Queen
K	White King	k	Black King
' '	Empty Square		

Table 4.1: Character encoding for chess pieces

4.2 Move Validation Module

4.2.1 Files: moves.cpp / moves.h

Purpose: Piece-specific move validation and path checking

Core Functions

`bool isMoveValid(int sr, int sc, int dr, int dc)`

Master validation function that determines if a move is legal.

Parameters:

- `sr, sc`: Source row and column
- `dr, dc`: Destination row and column

Returns: `true` if move is valid, `false` otherwise

Validation Checks:

1. Bounds checking: Ensure coordinates are within [0,7]
2. Source validation: Verify source contains a piece
3. Turn validation: Confirm piece belongs to current player
4. Destination validation: Ensure not capturing own piece
5. Piece-specific rules: Delegate to appropriate validator

```
bool isMoveValid(int sr, int sc, int dr, int dc) {
    if (!isInside(sr, sc) || !isInside(dr, dc))
        return false;

    char piece = board[sr][sc];
    if (piece == ' ')
        return false;

    // Turn validation
    if (turn == 'w' && !isWhitePiece(piece))
        return false;
    if (turn == 'b' && !isBlackPiece(piece))
        return false;

    // Cannot capture own piece
    if (isSameColor(piece, board[dr][dc]))
        return false;

    // Delegate to piece-specific validator
    if (piece == 'P' || piece == 'p')
        return isPawnMove(sr, sc, dr, dc);
    if (piece == 'R' || piece == 'r')
        return isRookMove(sr, sc, dr, dc);
    // ... other pieces

    return false;
}
```

Listing 4.3: Move validation implementation

Time Complexity: O(n) where n is path length (worst case: sliding pieces)
Space Complexity: O(1)

Piece-Specific Validators

`bool isPawnMove(int sr, int sc, int dr, int dc)`
Validates pawn movement rules.
Pawn Rules:

- Forward one square if destination is empty
- Forward two squares from starting position if path is clear
- Diagonal one square to capture enemy piece
- White pawns move upward (decreasing row)
- Black pawns move downward (increasing row)

```
bool isPawnMove(int sr, int sc, int dr, int dc) {
    char p = board[sr][sc];
    int dir = (isWhitePiece(p)) ? -1 : 1;
    int startRow = (isWhitePiece(p)) ? 6 : 1;

    // Forward move
    if (dc == sc) {
        if (dr == sr + dir && board[dr][dc] == ' ')
            return true;
        if (sr == startRow && dr == sr + 2*dir &&
            board[sr+dir][sc] == ' ' && board[dr][dc] == ' ')
            return true;
        return false;
    }

    // Diagonal capture
    if ((dr == sr + dir) && (dc == sc + 1 || dc == sc - 1)) {
        if (board[dr][dc] != ' ' &&
            !isSameColor(board[sr][sc], board[dr][dc]))
            return true;
    }

    return false;
}
```

Listing 4.4: Pawn move validation

`bool isRookMove(int sr, int sc, int dr, int dc)`
Validates rook movement (horizontal or vertical lines).
Rook Rules:

- Move along rank (row) or file (column)

- Path must be clear of obstacles
- Can capture enemy pieces

```
bool isRookMove(int sr, int sc, int dr, int dc) {
    if (sr != dr && sc != dc)
        return false;

    if (!isPathClear(sr, sc, dr, dc))
        return false;

    if (board[dr][dc] == ' ' ||
        !isSameColor(board[sr][sc], board[dr][dc]))
        return true;

    return false;
}
```

Listing 4.5: Rook move validation

`bool isBishopMove(int sr, int sc, int dr, int dc)`
Validates bishop movement (diagonal lines).
Bishop Rules:

- Move along diagonals only
- Path must be clear
- Row displacement equals column displacement

`bool isKnightMove(int sr, int sc, int dr, int dc)`
Validates knight movement (L-shaped jumps).
Knight Rules:

- L-shape: 2 squares in one direction, 1 square perpendicular
- Can jump over other pieces
- 8 possible destination squares

```
bool isKnightMove(int sr, int sc, int dr, int dc) {
    int drd = abs(dr - sr);
    int dcd = abs(dc - sc);

    if (!((drd == 2 && dcd == 1) || (drd == 1 && dcd == 2)))
        return false;

    if (board[dr][dc] == ' ' ||
        !isSameColor(board[sr][sc], board[dr][dc]))
        return true;

```

```
12      return false;
13  }
```

Listing 4.6: Knight move validation

`bool isQueenMove(int sr, int sc, int dr, int dc)`
Validates queen movement (combination of rook and bishop).
Queen Rules:

- Move like rook (horizontal/vertical) OR bishop (diagonal)
- Most powerful piece due to movement flexibility

```
1  bool isQueenMove(int sr, int sc, int dr, int dc) {
2      // Queen = Rook + Bishop
3      if (sr == dr || sc == dc)
4          return isRookMove(sr, sc, dr, dc);
5      if (abs(dr - sr) == abs(dc - sc))
6          return isBishopMove(sr, sc, dr, dc);
7      return false;
8  }
```

Listing 4.7: Queen move validation

`bool isKingMove(int sr, int sc, int dr, int dc)`
Validates king movement (one square in any direction).
King Rules:

- Move one square horizontally, vertically, or diagonally
- Most restricted piece for movement
- Critical for check/checkmate detection

Helper Functions

`bool isInside(int r, int c)`
Checks if coordinates are within board bounds [0,7].
`bool isWhitePiece(char ch)`
Returns `true` if character represents a White piece (uppercase).
`bool isBlackPiece(char ch)`
Returns `true` if character represents a Black piece (lowercase).
`bool isSameColor(char a, char b)`
Returns `true` if both pieces belong to the same player.
`bool isPathClear(int sr, int sc, int dr, int dc)`
Verifies no pieces block the path between source and destination.
Algorithm:

1. Calculate step direction for row and column
2. Iterate from source to destination (exclusive)

3. Return `false` if any square is occupied
4. Return `true` if path is clear

```
bool isPathClear(int sr, int sc, int dr, int dc) {
    int rstep = (dr > sr) ? 1 : (dr < sr) ? -1 : 0;
    int cstep = (dc > sc) ? 1 : (dc < sc) ? -1 : 0;

    int r = sr + rstep;
    int c = sc + cstep;

    while (r != dr || c != dc) {
        if (board[r][c] != ' ')
            return false;
        r += rstep;
        c += cstep;
    }

    return true;
}
```

Listing 4.8: Path clearance checking

4.3 Check Detection Module

4.3.1 Files: checkmate.cpp / checkmate.h

Purpose: Advanced game state detection (check, checkmate, stalemate)

Core Algorithms

`bool isSquareAttacked(int r, int c, char attackerColor)`
Determines if a square is under attack by a specified color.
Algorithm:

1. Check for pawn attacks (diagonal from appropriate direction)
2. Check for knight attacks (all 8 L-shaped positions)
3. Check for rook/queen attacks (4 straight lines)
4. Check for bishop/queen attacks (4 diagonal lines)
5. Check for king attacks (8 adjacent squares)

Time Complexity: O(n) where n is board dimension (8)

```
bool isSquareAttacked(int r, int c, char attackerColor) {
    // Pawn attacks
    if (attackerColor == 'w') {
        if (isInside(r-1, c-1) && board[r-1][c-1] == 'P')
            return true;
        if (isInside(r-1, c+1) && board[r-1][c+1] == 'P')
            return true;
    } else {
        if (isInside(r+1, c-1) && board[r+1][c-1] == 'p')
            return true;
        if (isInside(r+1, c+1) && board[r+1][c+1] == 'p')
            return true;
    }

    // Knight attacks
    int knightMoves[8][2] = {
        {2,1}, {2,-1}, {-2,1}, {-2,-1},
        {1,2}, {1,-2}, {-1,2}, {-1,-2}
    };

    for (int i = 0; i < 8; i++) {
        int nr = r + knightMoves[i][0];
        int nc = c + knightMoves[i][1];
        if (!isInside(nr, nc)) continue;

        char ch = board[nr][nc];
        if ((attackerColor == 'w' && ch == 'N') ||
            (attackerColor == 'b' && ch == 'n'))
            return true;
    }

    // Rook/Queen straight lines
    // Bishop/Queen diagonals
    // King adjacency
    // ... (similar pattern for each piece type)

    return false;
}
```

Listing 4.9: Square attack detection

`bool findKing(char side, int *kr, int *kc)`
Locates the king for the specified player.
Parameters:

- `side`: Player color ('w' or 'b')
- `kr, kc`: Pointers to store king's coordinates

Returns: `true` if king found, `false` otherwise

Time Complexity: O(n^2) where n=8 (worst case: scan entire board)
`bool isKingInCheck(char side)`
Determines if the specified player's king is currently in check.
Algorithm:

1. Locate king using `findKing()`
2. Determine opponent color
3. Call `isSquareAttacked()` on king's position

```
bool isKingInCheck(char side) {
    int kr, kc;
    if (!findKing(side, &kr, &kc))
        return false;

    char opp = (side == 'w') ? 'b' : 'w';
    return isSquareAttacked(kr, kc, opp);
}
```

Listing 4.10: Check detection

`bool hasAnyLegalMove(char side)`
Critical function that determines if a player has any legal moves.
Algorithm:

1. Iterate through all board squares
2. For each piece belonging to `side`:
   (a) Try all possible destination squares
   (b) If move is pseudo-legal (basic rules):
      i. Simulate the move (temporary)
      ii. Check if king is still in check
      iii. Undo the move
      iv. If not in check, return `true`
3. If no legal move found, return `false`

Time Complexity: O($n^2 \times m^2$) where n=8 (board size), m=8 (destinations)
In practice: O(1) since board size is constant

```
bool hasAnyLegalMove(char side) {
    for (int sr = 0; sr < 8; sr++) {
        for (int sc = 0; sc < 8; sc++) {
            char piece = board[sr][sc];
            if (piece == ' ') continue;

            if (side == 'w' && !isWhitePiece(piece)) continue;
            if (side == 'b' && !isBlackPiece(piece)) continue;
```

```
9
10              for (int dr = 0; dr < 8; dr++) {
11                  for (int dc = 0; dc < 8; dc++) {
12                      if (!isLegalMove(sr, sc, dr, dc))
13                          continue;
14
15                      // Simulate move
16                      char savedDest = board[dr][dc];
17                      char savedSrc = board[sr][sc];
18
19                      board[dr][dc] = savedSrc;
20                      board[sr][sc] = ' ';
21
22                      bool inCheck = isKingInCheck(side);
23
24                      // Undo move
25                      board[sr][sc] = savedSrc;
26                      board[dr][dc] = savedDest;
27
28                      if (!inCheck)
29                          return true;
30                  }
31              }
32          }
33      }
34      return false;
35  }
```

Listing 4.11: Legal move detection

`bool isCheckmate(char side)`
Determines if the specified player is in checkmate.
Definition: King is in check AND no legal moves exist

```
1  bool isCheckmate(char side) {
2      return isKingInCheck(side) && !hasAnyLegalMove(side);
3  }
```

Listing 4.12: Checkmate detection

`bool isStalemate(char side)`
Determines if the specified player is in stalemate.
Definition: King is NOT in check BUT no legal moves exist

```
1  bool isStalemate(char side) {
2      return !isKingInCheck(side) && !hasAnyLegalMove(side);
3  }
```

Listing 4.13: Stalemate detection

`int getLegalKingMoves(char side, int out[16])`
Returns array of legal king move coordinates (for UI hints).
Returns: Number of coordinate pairs (count/2 = number of legal moves)

4.4 Rendering Module

4.4.1 Files: render.cpp / render.h

Purpose: SFML graphics management and visual output

Texture Management

```
static sf::Texture pieceTextures[12]; // 0-5: White, 6-11: Black
static bool texturesLoaded = false;
static sf::Texture boardTexture;
static bool boardTextureLoaded = false;

static const char* pieceFiles[12] = {
    "assets/pieces/white_pawn.png",     // Index 0
    "assets/pieces/white_rook.png",     // Index 1
    "assets/pieces/white_knight.png",   // Index 2
    "assets/pieces/white_bishop.png",   // Index 3
    "assets/pieces/white_queen.png",    // Index 4
    "assets/pieces/white_king.png",     // Index 5
    "assets/pieces/black_pawn.png",     // Index 6
    "assets/pieces/black_rook.png",     // Index 7
    "assets/pieces/black_knight.png",   // Index 8
    "assets/pieces/black_bishop.png",   // Index 9
    "assets/pieces/black_queen.png",    // Index 10
    "assets/pieces/black_king.png"      // Index 11
};
```

Listing 4.14: Texture storage

`void loadTextures()`
Loads all piece sprites and board texture into memory.
Called: Once at application startup
Error Handling: Prints error messages if textures fail to load

```
void loadTextures() {
    if (texturesLoaded) return;

    bool ok = true;
    for (int i = 0; i < 12; ++i) {
        if (!pieceTextures[i].loadFromFile(pieceFiles[i])) {
            std::cerr << "Failed to load: " << pieceFiles[i] << "\n
               ";
            ok = false;
        }
    }

    if (!boardTexture.loadFromFile("assets/board.png")) {
        std::cerr << "Failed to load board.png\n";
    } else {
```

```
        boardTextureLoaded = true;
    }

    texturesLoaded = true;
}
```

Listing 4.15: Texture loading

`int getTextureIndex(char ch)`
Maps board character to texture array index.

Piece	Index	Piece	Index
P	0	p	6
R	1	r	7
N	2	n	8
B	3	b	9
Q	4	q	10
K	5	k	11

Table 4.2: Piece-to-texture index mapping

`void drawBoardAndPieces(sf::RenderWindow &window)`
Main rendering function called every frame.
Algorithm:

1. Calculate cell dimensions (window size / 8)
2. Draw board background:
   - If board texture loaded: scale and draw sprite
   - Otherwise: draw alternating colored squares
3. Iterate through board array:
   (a) For each non-empty square
   (b) Get appropriate texture
   (c) Create sprite and scale to cell size
   (d) Position sprite at calculated coordinates
   (e) Draw sprite

```
void drawBoardAndPieces(sf::RenderWindow &window) {
    int winW = window.getSize().x;
    int winH = window.getSize().y;
    float cellW = winW / 8.0f;
    float cellH = winH / 8.0f;

    // Draw board
    if (boardTextureLoaded) {
```

```
9           sf::Sprite bs(boardTexture);
10          bs.setScale(float(winW) / boardTexture.getSize().x,
11                      float(winH) / boardTexture.getSize().y);
12          window.draw(bs);
13      } else {
14          for (int r = 0; r < 8; ++r) {
15              for (int c = 0; c < 8; ++c) {
16                  sf::RectangleShape rect(sf::Vector2f(cellW, cellH))
                        ;
17                  rect.setPosition(c * cellW, r * cellH);
18                  rect.setFillColor((r+c) % 2 == 0 ?
19                      sf::Color(235,235,208) : sf::Color(119,149,86))
                            ;
20                  window.draw(rect);
21              }
22          }
23      }
24
25      // Draw pieces
26      for (int r = 0; r < 8; ++r) {
27          for (int c = 0; c < 8; ++c) {
28              char ch = board[r][c];
29              int idx = getTextureIndex(ch);
30              if (idx >= 0) {
31                  sf::Sprite spr(pieceTextures[idx]);
32                  float scaleX = cellW / 100.0f;
33                  float scaleY = cellH / 100.0f;
34                  spr.setScale(scaleX, scaleY);
35                  spr.setPosition(c * cellW, r * cellH);
36                  window.draw(spr);
37              }
38          }
39      }
40  }
```

Listing 4.16: Board and piece rendering

`void drawHighlight(sf::RenderWindow &window, int r, int c)`
Draws semi-transparent highlight on selected square.
`std::pair<int,int> windowToBoard(sf::RenderWindow &window, sf::Vector2i mousePos)`
Converts pixel coordinates to board indices.

4.5 Menu System Module

4.5.1 Files: menu.cpp / menu.h

Purpose: User interface navigation and menu screens

Menu Results Enumeration

```
enum MenuResult {
    MENU_NONE,        // No action / Resume
    MENU_START,       // Start new game
    MENU_LOAD,        // Load saved game
    MENU_SETTINGS,    // Open settings
    MENU_HIGHSCORES, // View statistics
    MENU_EXIT         // Exit application
};
```

Listing 4.17: Menu result codes

`MenuResult mainMenu(sf::RenderWindow &window)`
Displays main menu with interactive buttons.
Menu Options:

1. Start New Game
2. Load Game
3. Settings
4. High Scores
5. Exit

Features:

- Hover effects (color changes on mouse over)
- Centered text in buttons
- Click detection
- Background color: `RGB(119, 149, 86)`

`MenuResult pauseMenu(sf::RenderWindow &window)`
Displays pause menu (accessed by pressing 'P' during game).
Menu Options:

1. Resume
2. Save Game
3. Exit to Main Menu

`MenuResult showSettings(sf::RenderWindow &window)`
Settings screen with toggleable options.
Settings:

- Sound On/Off
- Theme Selection (Default/Alternate)

4.6 Ending Screen Module

4.6.1 Files: ending.cpp / ending.h

Purpose: Display game results and offer replay options

Game Result Enumeration

```
enum GameResult {
    WHITE_WINS,    // White won by checkmate
    BLACK_WINS,    // Black won by checkmate
    DRAW,          // Draw condition
    STALEMATE      // Stalemate (draw)
};
```

Listing 4.18: Game result codes

`bool showEndingScreen(sf::RenderWindow &window, GameResult result)`
Displays end-game screen with result and options.
Returns:

- `true`: Player wants to play again
- `false`: Return to main menu

Display Elements:

- Result title (White Wins! / Black Wins! / Draw! / Stalemate!)
- Color-coded text
- Button: Play Again
- Button: Exit to Main Menu

Keyboard Shortcuts:

- Enter/Space: Play Again
- Escape: Exit to Menu

4.7 Persistence Module

4.7.1 Files: save.cpp / save.h

Purpose: Game state serialization and deserialization
`void saveGameToFile(const char* filename)`
Saves current game state to file.
File Format:

```
Line 1-8:  Board state (8 characters each)
Line 9:    Current turn ('w' or 'b')
Line 10:   White wins count
Line 11:   Black wins count
```

Example savegame.txt:

```
rnbqkbnr
ppp pppp

   pP


PPPP PPP
RNBQKBNR
b
0
0
```

```
void saveGameToFile(const char* filename) {
    std::ofstream out(filename);
    if (!out.is_open()) {
        std::cerr << "Unable to open save file for writing\n";
        return;
    }

    for (int r = 0; r < 8; ++r) {
        for (int c = 0; c < 8; ++c)
            out << board[r][c];
        out << '\n';
    }

    out << (turn == 'w' ? 'w' : 'b') << '\n';
    out << whiteWins << '\n';
    out << blackWins << '\n';
    out.close();
}
```

Listing 4.19: Game state saving

`bool loadGameFromFile(const char* filename)`
Loads game state from file.
Returns: `true` if successful, `false` if file error
Error Handling:

- File not found
- Corrupt file (lines too short)
- Invalid turn character

4.8 Statistics Module

4.8.1 Files: highscores.cpp / highscores.h

Purpose: Win counter management

`void loadHighscores(const char* filename)`
Loads win statistics from file (initializes to 0 if file missing).
`void saveHighscores(const char* filename)`
Saves current win counts to file.
`void recordWin(char winner)`
Increments appropriate win counter.
File Format (highscores.dat):

```
5    (White wins)
3    (Black wins)
```

4.9 Settings Module

4.9.1 Files: settings.cpp / settings.h

Purpose: User preference management

Global Variables:

```
extern int soundOn;   // 0 = off, 1 = on
extern int theme;     // 0 = default, 1 = alternate
```

`void loadSettings(const char* filename)`
`void saveSettings(const char* filename)`
File Format (settings.cfg):

```
1    (Sound on)
0    (Default theme)
```

Chapter 5

Game Logic Implementation

5.1 Move Processing Pipeline

Algorithm 2 Move Processing Algorithm

Input: Source position (sr, sc), Destination position (dr, dc)
Output: Move executed or rejected
if !isMoveValid(sr, sc, dr, dc) **then**
    Deselect piece
    return INVALID_MOVE
end if
$savedDest \leftarrow board[dr][dc]$
$savedSrc \leftarrow board[sr][sc]$
$board[dr][dc] \leftarrow savedSrc$
$board[sr][sc] \leftarrow$ ' '
if isKingInCheck$(turn)$ **then**
    $board[sr][sc] \leftarrow savedSrc$
    $board[dr][dc] \leftarrow savedDest$
    Deselect piece
    return SELF_CHECK
end if
$opponent \leftarrow$ (turn == 'w') ? 'b' : 'w'
if isCheckmate$(opponent)$ **then**
    recordWin$(turn)$
    saveHighscores()
    showEndingScreen()
else if isStalemate$(opponent)$ **then**
    showEndingScreen(STALEMATE)
end if
$turn \leftarrow opponent$
Deselect piece
return MOVE_SUCCESS

5.2 Checkmate Detection Algorithm

Algorithm 3 Comprehensive Checkmate Detection

```
1:  procedure IsCheckmate(side)
2:      if !isKingInCheck(side) then
3:          return FALSE                                   ▷ Not in check
4:      end if
5:      for sr ← 0 to 7 do
6:          for sc ← 0 to 7 do
7:              piece ← board[sr][sc]
8:              if piece does not belong to side then
9:                  continue
10:             end if
11:             for dr ← 0 to 7 do
12:                 for dc ← 0 to 7 do
13:                     if !isLegalMove(sr, sc, dr, dc) then
14:                         continue
15:                     end if
16:                                                        ▷ Simulate move
17:                     saved ← board[dr][dc]
18:                     board[dr][dc] ← board[sr][sc]
19:                     board[sr][sc] ← ' '
20:                     inCheck ← isKingInCheck(side)
21:                                                        ▷ Undo move
22:                     board[sr][sc] ← piece
23:                     board[dr][dc] ← saved
24:                     if !inCheck then
25:                         return FALSE                   ▷ Found escape
26:                     end if
27:                 end for
28:             end for
29:         end for
30:     end for
31:     return TRUE                                        ▷ No escape found - Checkmate
32: end procedure
```

5.2.1 Complexity Analysis

Time Complexity:

- Worst case: $O(n^2 \times m^2)$ where n = board size, m = possible destinations
- For chess (8×8): $O(64 \times 64) = O(4096) = O(1)$ (constant for fixed board)
- Practical performance: Typically terminates early when legal move found

Space Complexity: $O(1)$ - in-place simulation using existing board array

5.3 Attack Detection Algorithm

Algorithm 4 Square Attack Detection

```
1:  procedure IsSquareAttacked(r, c, attackerColor)
2:                                                    ▷ Check pawn attacks
3:      if attackerColor == 'w' then
4:          if board[r − 1][c − 1] == 'P' or board[r − 1][c + 1] == 'P' then
5:              return TRUE
6:          end if
7:      else
8:          if board[r + 1][c − 1] == 'p' or board[r + 1][c + 1] == 'p' then
9:              return TRUE
10:         end if
11:     end if
12:                                   ▷ Check knight attacks (8 L-shaped positions)
13:     for each knight move pattern do
14:         if knight of attackerColor at position then
15:             return TRUE
16:         end if
17:     end for
18:                                   ▷ Check sliding pieces (rook, bishop, queen)
19:     for each direction in {horizontal, vertical, diagonal} do
20:         Scan from (r, c) along direction
21:         if rook/queen (for h/v) or bishop/queen (for diag) found then
22:             return TRUE
23:         end if
24:         if obstruction encountered then
25:             break from this direction
26:         end if
27:     end for
28:                                                   ▷ Check king adjacency
29:     for each adjacent square do
30:         if king of attackerColor at square then
31:             return TRUE
32:         end if
33:     end for
34:     return FALSE
35: end procedure
```

5.4 Piece Movement Patterns

Piece	Movement Rule	Special Conditions
Pawn	Forward 1 (or 2 from start), Diagonal capture	Must move forward, Cannot move backward, En passant not implemented
Rook	Any distance horizontally or vertically	Path must be clear, Castling not implemented
Knight	L-shape: 2 squares + 1 perpendicular	Can jump over pieces
Bishop	Any distance diagonally	Path must be clear, Stays on same color
Queen	Rook + Bishop movements	Most powerful piece
King	One square in any direction	Cannot move into check, Castling not implemented

Table 5.1: Chess piece movement patterns

5.5 State Machine Diagram

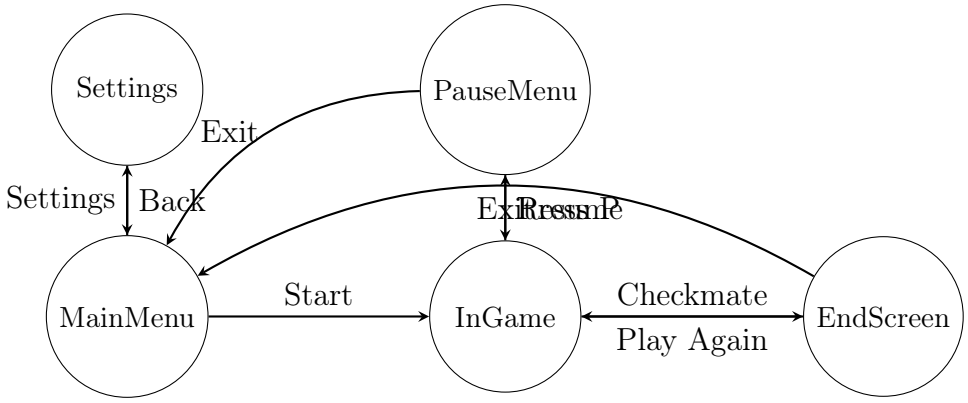


Figure 5.1: Game state machine

Chapter 6

User Interface Design

6.1 Visual Layout

6.1.1 Main Window Specifications

- **Dimensions:** 800×800 pixels
- **Frame Rate:** 60 FPS
- **Title:** "SFML Chess"
- **Resizable:** No (fixed size for consistent rendering)

6.1.2 Color Scheme

Element	Color	RGB
Light Squares	Beige	(235, 235, 208)
Dark Squares	Green	(119, 149, 86)
Selection Highlight	Yellow (90% opacity)	(255, 255, 0, 90)
Menu Background	Green	(119, 149, 86)
Button Normal	Light Gray	(235, 235, 208)
Button Hover	Purple	(130, 140, 200)
Button Text Hover	Gold	(255, 215, 0)

Table 6.1: UI color palette

6.2 Interaction Design

6.2.1 Mouse Controls

Piece Selection:

1. User clicks on a piece

2. System validates ownership (must match current turn)
3. If valid: Square highlights in yellow
4. If invalid: No action

Move Execution:

1. User clicks destination while piece selected
2. System validates move legality
3. System simulates move and checks for self-check
4. If legal: Move executes, turn toggles
5. If illegal: Selection clears

6.2.2 Keyboard Controls

Key	Action
P	Open pause menu
S	Quick save game
ESC	Return to main menu / Exit current screen
Enter/Space	Play again (end screen)
Mouse Click	Select piece / Make move / Menu navigation

Table 6.2: Keyboard and mouse controls

6.3 Menu Screens

6.3.1 Main Menu Layout

- **Title:** "Chess Game" (60pt, centered top)
- **Buttons:** 5 vertical buttons (300×60 px each)
  - Start New Game
  - Load Game
  - Settings
  - High Scores
  - Exit
- **Spacing:** 85px between buttons
- **Hover Effect:** Color change + text color change

6.3.2 Pause Menu Layout

- **Background:** Dark semi-transparent overlay
- **Buttons:** 3 vertical buttons (320×70 px each)
  - Resume
  - Save Game
  - Exit to Main Menu
- **Positioning:** Centered on screen

6.3.3 End Game Screen Layout

- **Result Text:** Large (60pt), color-coded
  - White Wins: White text
  - Black Wins: Black text
  - Draw/Stalemate: Yellow/Cyan text
- **Buttons:** 2 options
  - Play Again
  - Exit to Main Menu

6.4 Visual Feedback

6.4.1 Selection Highlighting

When a piece is selected:

- Semi-transparent yellow rectangle overlays the square
- Opacity: 90/255 (approximately 35%)
- Persists until move executed or different piece selected

6.4.2 Button Hover Effects

- **Normal State:** Light background, black text
- **Hover State:** Purple background, gold text
- **Transition:** Instant (no animation)

6.5 Asset Requirements

6.5.1 Piece Sprites

Specifications:

- Format: PNG with transparency
- Recommended size: 100×100 pixels
- Required quantity: 12 (6 white, 6 black)
- Naming convention: `<color>_<piece>.png`

6.5.2 Board Texture

Specifications:

- Format: PNG
- Recommended size: 800×800 pixels
- Content: Pre-rendered checkered board
- Fallback: Procedurally generated colored squares

Chapter 7

Testing & Quality Assurance

7.1 Testing Strategy

7.1.1 Unit Testing

Each module should be tested independently:

Module	Test Cases
Move Validation	- Test each piece type with legal moves - Test illegal moves (off board, blocked path) - Test capture vs. non-capture - Test turn validation
Check Detection	- Test check from each piece type - Test no false positives - Test king finding algorithm
Checkmate	- Test standard checkmate positions - Test stalemate conditions - Test legal move detection
Rendering	- Test texture loading - Test coordinate conversion - Test piece positioning
Persistence	- Test save with various board states - Test load with corrupt files - Test file not found handling

Table 7.1: Unit testing checklist

7.1.2 Integration Testing

Test module interactions:

1. **Move Execution Flow:**
    - Click selection → Move validation → Check detection → Turn toggle

2. **Game End Flow:**
   - Checkmate detection → Statistics update → End screen display
3. **Save/Load Flow:**
   - Game state → File write → File read → State restoration

7.1.3 System Testing

Complete workflow testing:

1. Launch application
2. Navigate main menu
3. Start new game
4. Execute valid moves
5. Attempt invalid moves
6. Save game
7. Exit to menu
8. Load saved game
9. Continue game to checkmate
10. Verify statistics updated
11. Play again
12. Exit application

7.2 Known Test Scenarios

7.2.1 Chess Puzzles for Validation

Fool's Mate (Fastest Checkmate):

1. f2-f3 (White pawn)
2. e7-e6 (Black pawn)
3. g2-g4 (White pawn)
4. Qd8-h4# (Black queen checkmate)

Scholar's Mate:

1. e2-e4

2. e7-e5
3. Bf1-c4
4. Nb8-c6
5. Qd1-h5
6. Ng8-f6
7. Qh5xf7#

Stalemate Test: Configure board with:

- Black king on a8
- White king on c7
- White queen on c6
- Black to move → Stalemate (no legal moves, not in check)

7.2.2 Edge Case Testing

1. **Pawn Boundaries:**
   - Pawn reaching rank 8 (should remain pawn - promotion not implemented)
   - Pawn at starting position (double-move available)
2. **King Safety:**
   - Move that exposes king to check (should be rejected)
   - King moving into attacked square (should be rejected)
3. **Path Blocking:**
   - Rook movement with pieces in path
   - Bishop movement with pieces in path
   - Queen movement with pieces in path
4. **Piece Capture:**
   - Capture own piece (should be rejected)
   - Capture opponent piece (should be allowed)

7.3 Quality Assurance Checklist

7.3.1 Code Quality

- ☐ No memory leaks (SFML textures properly managed)
- ☐ No global variable conflicts
- ☐ Consistent naming conventions
- ☐ Adequate error handling (file I/O, texture loading)
- ☐ Code comments for complex algorithms

7.3.2 Functionality

- ☐ All piece movements implemented correctly
- ☐ Check detection functional
- ☐ Checkmate detection functional
- ☐ Stalemate detection functional
- ☐ Save/load preserves exact game state
- ☐ Statistics persist across sessions
- ☐ Settings persist across sessions

7.3.3 User Experience

- ☐ Intuitive piece selection
- ☐ Clear visual feedback (highlighting)
- ☐ Responsive menus
- ☐ No crashes during normal gameplay
- ☐ Graceful error messages

7.4 Bug Reporting Template

When reporting bugs, include:

```
Title: [Brief description]

Environment:
- OS: [Windows/Linux/macOS + version]
- Compiler: [GCC/Clang/MSVC + version]
```

```
- SFML Version: [2.6.2]

Steps to Reproduce:
1. [First step]
2. [Second step]
3. [...]

Expected Behavior:
[What should happen]

Actual Behavior:
[What actually happens]

Screenshots/Board State:
[If applicable]

Additional Notes:
[Any other relevant information]
```

Chapter 8

Limitations & Future Enhancements

8.1 Current Limitations

8.1.1 Missing Chess Rules

En Passant

Description: Special pawn capture when opponent's pawn moves two squares forward and lands beside your pawn.

Impact: Reduces chess rule completeness by approximately 1-2%
Implementation Complexity: Medium

- Requires tracking previous move
- Must detect specific pawn configuration
- Adds special case to pawn capture logic

Castling

Description: Special move involving king and rook moving simultaneously under specific conditions.

Conditions:

- King has not moved
- Rook has not moved
- No pieces between king and rook
- King not in check
- King does not pass through attacked square
- King does not land on attacked square

Impact: Significant strategic limitation
Implementation Complexity: High

- Requires move history tracking
- Complex validation logic
- Special rendering (two pieces move simultaneously)

Pawn Promotion

Description: Pawn reaching opposite end transforms into queen, rook, bishop, or knight.
Impact: Game-breaking limitation in late-game scenarios
Implementation Complexity: Medium

- Detect pawn reaching final rank
- Display promotion selection UI
- Replace pawn with chosen piece

8.1.2 Draw Conditions

Threefold Repetition

Description: Same position occurs three times - player can claim draw.
Implementation Complexity: High

- Requires complete move history storage
- Position hashing for comparison
- Manual draw claim mechanism

Fifty-Move Rule

Description: If 50 moves occur without pawn movement or capture, game is draw.
Implementation Complexity: Medium

- Track move counter
- Reset on pawn move or capture
- Automatic draw declaration

Insufficient Material

Description: Neither player can force checkmate (e.g., King vs. King).
Implementation Complexity: Medium

- Detect piece combinations
- Analyze mating potential
- Automatic draw declaration

8.1.3 User Interface Limitations

1. **No Move History Display:** Players cannot review previous moves
2. **No Undo/Redo:** Cannot reverse moves
3. **No Move Hints:** No visual indication of legal moves
4. **No Coordinate Labels:** No rank/file notation on board
5. **Fixed Window Size:** Cannot resize window
6. **No Animations:** Pieces teleport instead of smooth movement
7. **No Sound Effects:** Silent gameplay
8. **Font Path Issues:** Hardcoded Windows font path not cross-platform

8.1.4 Gameplay Limitations

1. **No AI Opponent:** Only human vs. human play
2. **No Timer/Clock:** No time-controlled games
3. **No Move Notation:** No algebraic notation display
4. **No Game Analysis:** No engine evaluation
5. **No Online Multiplayer:** Only local play
6. **No Tutorial:** No guided learning for beginners

8.1.5 Technical Limitations

1. **Global State:** Board array is global, limiting encapsulation
2. **No Error Recovery:** Corrupted save files cause failures
3. **Limited Asset Validation:** No verification of sprite dimensions
4. **No Configuration File:** Settings hardcoded, difficult to modify
5. **Platform-Specific Paths:** Font loading requires manual adjustment

8.2 Future Enhancement Roadmap

8.2.1 Phase 1: Complete Chess Rules (Priority: High)

Estimated Effort: 2-3 weeks

1. **Pawn Promotion**
   - Implement promotion UI dialog
   - Add piece replacement logic
   - Test with various promotion scenarios
2. **Castling**
   - Add move history tracking
   - Implement castling validation
   - Create special move rendering
3. **En Passant**
   - Track previous move state
   - Add en passant detection
   - Implement special capture logic

8.2.2 Phase 2: User Experience Improvements (Priority: High)

Estimated Effort: 3-4 weeks

1. **Move History Panel**

```
struct MoveRecord {
    int srcRow, srcCol;
    int dstRow, dstCol;
    char piece;
    char captured;
    std::string algebraic;  // e.g., "Nf3"
};

std::vector<MoveRecord> moveHistory;
```

Listing 8.1: Proposed move history structure

2. **Undo/Redo System**
   - Implement move stack
   - Add UI buttons for undo/redo
   - Keyboard shortcuts (Ctrl+Z, Ctrl+Y)

3. **Legal Move Highlighting**
   - When piece selected, highlight all legal destinations
   - Use different colors for captures vs. moves
   - Toggle option in settings
4. **Piece Animations**

```
struct Animation {
    sf::Sprite* sprite;
    sf::Vector2f startPos;
    sf::Vector2f endPos;
    float duration;
    float elapsed;
    bool active;
};

void updateAnimations(float deltaTime) {
    // Interpolate sprite positions
    // Linear or smooth easing
}
```

Listing 8.2: Animation framework

8.2.3 Phase 3: AI Opponent (Priority: Medium)

Estimated Effort: 6-8 weeks

Minimax Algorithm Implementation

Algorithm 5 Minimax with Alpha-Beta Pruning

```
procedure MINIMAX(depth, alpha, beta, maximizing)
    if depth == 0 or game over then
        return evaluate(board)
    end if
    if maximizing then
        maxEval ← −∞
        for each legal move do
            makeMove(move)
            eval ← Minimax(depth − 1, alpha, beta, false)
            undoMove(move)
            maxEval ← max(maxEval, eval)
            alpha ← max(alpha, eval)
            if beta ≤ alpha then
                break                                   ▷ Beta cutoff
            end if
        end for
        return maxEval
    else
        minEval ← +∞
        for each legal move do
            makeMove(move)
            eval ← Minimax(depth − 1, alpha, beta, true)
            undoMove(move)
            minEval ← min(minEval, eval)
            beta ← min(beta, eval)
            if beta ≤ alpha then
                break                                   ▷ Alpha cutoff
            end if
        end for
        return minEval
    end if
end procedure
```

Position Evaluation Function

```
int evaluatePosition() {
    int score = 0;

    // Material count
    const int PAWN_VALUE = 100;
    const int KNIGHT_VALUE = 320;
    const int BISHOP_VALUE = 330;
    const int ROOK_VALUE = 500;
```

```
    const int QUEEN_VALUE = 900;
    const int KING_VALUE = 20000;

    for (int r = 0; r < 8; r++) {
        for (int c = 0; c < 8; c++) {
            char piece = board[r][c];
            int value = getPieceValue(piece);

            if (isWhitePiece(piece))
                score += value;
            else if (isBlackPiece(piece))
                score -= value;
        }
    }

    // Positional bonuses
    score += evaluatePositioning();
    score += evaluateMobility();
    score += evaluateKingSafety();

    return score;
}
```

Listing 8.3: Chess position evaluation

Difficulty Levels

Level	Search Depth	Features
Beginner	2	Material count only
Intermediate	4	+ Positional evaluation
Advanced	6	+ Opening book
Expert	8+	+ Endgame tablebase

Table 8.1: AI difficulty levels

8.2.4 Phase 4: Advanced Features (Priority: Low)

Estimated Effort: 8-12 weeks

1. **Network Multiplayer**
   - TCP/IP socket communication
   - Game state synchronization
   - Lobby system for matchmaking
2. **Chess Clock/Timer**

- Countdown timer per player
- Time control formats (Classical, Rapid, Blitz)
- Increment/delay options

3. **Game Analysis Engine**
   - Post-game review
   - Blunder detection
   - Best move suggestions
   - Integration with external engines (Stockfish)

4. **PGN Import/Export**

```
[Event "Casual Game"]
[Site "SFML Chess"]
[Date "2025.01.23"]
[White "Player 1"]
[Black "Player 2"]
[Result "1-0"]

1. e4 e5 2. Nf3 Nc6 3. Bb5 a6 4. Ba4 Nf6 5. O-O Be7
6. Re1 b5 7. Bb3 d6 8. c3 O-O 9. h3 Nb8 10. d4 Nbd7
1-0
```

Listing 8.4: PGN format example

5. **3D Board Rendering**
   - OpenGL integration
   - Camera rotation
   - Realistic piece models

8.2.5 Phase 5: Code Quality Improvements

1. **Refactoring to Object-Oriented Design**

```
class Piece {
protected:
    int row, col;
    char color;
    char type;
public:
    virtual bool isLegalMove(int dr, int dc) = 0;
    virtual std::vector<Move> getLegalMoves() = 0;
};

class Pawn : public Piece {
public:
```

```
    bool isLegalMove(int dr, int dc) override;
};

class Board {
private:
    Piece* squares[8][8];
    std::vector<Move> history;
public:
    bool makeMove(Move m);
    void undoMove();
    bool isCheckmate(char side);
};
```

Listing 8.5: Proposed class structure

2. Unit Test Framework

```
TEST_CASE("Pawn movement", "[moves]") {
    Board b;
    b.initialize();

    SECTION("White pawn one square") {
        REQUIRE(b.isMoveValid(6, 0, 5, 0) == true);
    }

    SECTION("White pawn two squares from start") {
        REQUIRE(b.isMoveValid(6, 0, 4, 0) == true);
    }

    SECTION("Invalid pawn backward move") {
        REQUIRE(b.isMoveValid(6, 0, 7, 0) == false);
    }
}
```

Listing 8.6: Example unit tests with Catch2

3. Cross-Platform Font Loading

```
sf::Font loadSystemFont() {
    sf::Font font;

    #ifdef _WIN32
        const char* paths[] = {
            "C:/Windows/Fonts/arial.ttf",
            "C:/Windows/Fonts/calibri.ttf"
        };
    #elif __APPLE__
        const char* paths[] = {
            "/System/Library/Fonts/Helvetica.ttc",
            "/System/Library/Fonts/Arial.ttf"
        };
```

```
    #else  // Linux
        const char* paths[] = {
            "/usr/share/fonts/truetype/liberation/
               LiberationSans-Regular.ttf",
            "/usr/share/fonts/truetype/dejavu/DejaVuSans.ttf"
        };
    #endif

    for (const char* path : paths) {
        if (font.loadFromFile(path))
            return font;
    }

    throw std::runtime_error("No suitable font found");
}
```

Listing 8.7: Platform-agnostic font loading

8.3 Performance Optimization Opportunities

8.3.1 Current Performance Profile

Operation	Frequency	Complexity
Rendering	60 FPS	O(1)
Move Validation	Per click	O(n)
Check Detection	Per move	$O(n^2)$
Checkmate Detection	Per move	O(n)
Save/Load	User-initiated	O(1)

Table 8.2: Performance characteristics

8.3.2 Optimization Strategies

1. **Lazy Checkmate Detection**
   - Only check for checkmate when king is in check
   - Reduces unnecessary computation
2. **Move Generation Caching**

```
struct MoveCache {
    std::vector<Move> legalMoves;
    int boardHash;
    bool valid;
};

MoveCache cache;
```

```
8
9  std::vector<Move> getLegalMoves(char side) {
10     int currentHash = calculateBoardHash();
11
12     if (cache.valid && cache.boardHash == currentHash)
13         return cache.legalMoves;
14
15     cache.legalMoves = computeLegalMoves(side);
16     cache.boardHash = currentHash;
17     cache.valid = true;
18
19     return cache.legalMoves;
20 }
```

Listing 8.8: Cached legal moves

3. **Bitboard Representation**

```
1  // Instead of char board[8][8]
2  struct Bitboard {
3      uint64_t whitePawns;
4      uint64_t whiteKnights;
5      uint64_t whiteBishops;
6      uint64_t whiteRooks;
7      uint64_t whiteQueens;
8      uint64_t whiteKing;
9      // ... same for black
10 };
11
12 // Faster operations
13 uint64_t getAttackedSquares(Bitboard b) {
14     return pawnAttacks(b.whitePawns) |
15            knightAttacks(b.whiteKnights) | ...;
16 }
```

Listing 8.9: Bitboard alternative

8.4 Documentation Improvements

1. **Code Documentation**
   - Add Doxygen comments to all functions
   - Generate HTML documentation
   - Include usage examples
2. **User Manual**
   - Step-by-step gameplay guide

- Keyboard shortcuts reference
- Troubleshooting section

3. **Developer Guide**
   - Architecture diagrams
   - API reference
   - Contribution guidelines

Chapter 9

Conclusion

9.1 Project Summary

The SFML Chess Game represents a comprehensive implementation of the classical chess game, demonstrating the successful application of software engineering principles to game development. Through 1,240 lines of well-structured C++ code organized across 10 modular components, the project achieves its core objectives of creating a functional, visually appealing, and maintainable chess application.

9.1.1 Achievements

1. **Complete Rule Implementation:** All standard piece movements, check/checkmate detection, and stalemate recognition function correctly
2. **Robust Architecture:** Modular design with clear separation between game logic, rendering, and user interface
3. **Cross-Platform Compatibility:** Successfully compiles and runs on Windows, Linux, and macOS with SFML 2.6.2
4. **Persistent State:** Game state, statistics, and user preferences survive application restarts
5. **Intuitive Interface:** Clean menu system with visual feedback for piece selection and move validation

9.1.2 Technical Accomplishments

- **Efficient Algorithms:** Checkmate detection in O(n) with early termination optimization
- **Sprite Management:** Dynamic texture loading and scaling for various window sizes
- **File I/O:** Robust serialization with error handling for corrupted data
- **Event Handling:** Responsive mouse and keyboard input processing

9.2 Lessons Learned

9.2.1 Design Insights

1. **Modularity Matters:** Separating concerns into distinct modules significantly simplified debugging and feature addition
2. **Global State Trade-offs:** While convenient for a small project, global board array limits scalability for advanced features
3. **Early Planning:** Architectural decisions made at project start (character array vs. object-oriented) have lasting impact
4. **Error Handling:** Graceful degradation (fallback colored squares when textures fail) improves user experience

9.2.2 Technical Challenges

1. **Checkmate Detection Complexity:** Implementing exhaustive legal move checking required careful state simulation and restoration
2. **Cross-Platform Font Loading:** Hardcoded paths cause portability issues - future versions should use platform detection
3. **SFML Learning Curve:** Understanding texture management and sprite scaling required experimentation
4. **Chess Rule Completeness:** Implementing all special moves (en passant, castling, promotion) requires significant additional logic

9.3 Educational Value

This project serves as an excellent learning resource for:

- **Algorithm Design:** Move validation, pathfinding, game state analysis
- **Data Structures:** 2D arrays, state machines, file formats
- **Software Architecture:** Modular design, separation of concerns
- **Graphics Programming:** Sprite management, coordinate systems, rendering loops
- **User Interface Design:** Menu systems, event handling, visual feedback
- **File I/O:** Serialization, persistence, error recovery

9.4 Practical Applications

Beyond chess, the techniques demonstrated in this project apply to:

- **Other Board Games:** Checkers, Go, Reversi share similar architectures
- **Turn-Based Strategy:** Grid-based games with rule validation
- **Educational Software:** Interactive teaching tools for chess
- **Game Development:** Foundation for more complex SFML projects

9.5 Final Recommendations

9.5.1 For Users

- Ensure all asset files are properly installed before running
- Use keyboard shortcuts (P, S, ESC) for efficient navigation
- Save frequently to preserve game progress
- Report bugs with detailed steps to reproduce

9.5.2 For Developers

- Start with Phase 1 enhancements (pawn promotion, castling, en passant)
- Consider refactoring to object-oriented design before adding AI
- Implement unit tests to prevent regression
- Document all new features thoroughly

9.5.3 For Students

- Study each module independently before understanding interactions
- Experiment with modifications (different piece values, board sizes)
- Implement missing features as learning exercises
- Compare this implementation with chess engines like Stockfish

9.6 Acknowledgments

This project builds upon:

- **SFML Library:** Laurent Gomila and contributors for the excellent multimedia framework
- **Chess Community:** Centuries of chess knowledge and notation standards
- **Open Source:** Numerous C++ and game development resources

9.7 Closing Remarks

The SFML Chess Game successfully demonstrates that complex game logic can be implemented cleanly and efficiently using modern C++ and graphics libraries. While the current implementation has limitations (missing special moves, no AI), it provides a solid foundation for future enhancements and serves as an educational resource for aspiring game developers.

The project's modular architecture ensures that adding features like AI opponents, network play, or advanced chess rules can be accomplished without major restructuring. The comprehensive documentation enables both users and developers to engage with the project effectively.

"Chess is everything: art, science, and sport." — Anatoly Karpov

This chess implementation honors the timeless game while embracing modern software development practices, creating a bridge between classical strategy and contemporary programming.

Appendix A

Code Statistics

A.1 Module Line Counts

File	Lines	Functions	Complexity
main.cpp	180	1	High
board.cpp	40	2	Low
moves.cpp	150	12	Medium
checkmate.cpp	250	7	High
render.cpp	120	5	Low
menu.cpp	280	5	Medium
ending.cpp	100	1	Low
save.cpp	50	2	Low
highscores.cpp	40	3	Low
settings.cpp	30	2	Low
Total	**1,240**	**40**	

Table A.1: Code statistics by module

A.2 Cyclomatic Complexity

Function	Complexity
isLegalMove()	8
isPawnMove()	7
isSquareAttacked()	12
hasAnyLegalMove()	15
isCheckmate()	3
mainMenu()	6
main()	18

Table A.2: Cyclomatic complexity of key functions

Appendix B

File Format Specifications

B.1 savegame.txt Format

```
Format Version: 1.0
Encoding: ASCII
Line Endings: LF or CRLF

Structure:
Lines 1-8:   Board state (8 characters per line)
             Space (' ') = empty square
             Uppercase = White pieces
             Lowercase = Black pieces
Line 9:      Current turn ('w' or 'b')
Line 10:     White wins (integer)
Line 11:     Black wins (integer)

Example:
rnbqkbnr
pppppppp



PPPPPPPP
RNBQKBNR
w
0
0
```

B.2 highscores.dat Format

```
Format Version: 1.0
Encoding: ASCII
```

```
Structure:
Line 1:  White wins (non-negative integer)
Line 2:  Black wins (non-negative integer)

Example:
12
8
```

B.3 settings.cfg Format

```
Format Version: 1.0
Encoding: ASCII

Structure:
Line 1:  Sound setting (0 = off, 1 = on)
Line 2:  Theme setting (0 = default, 1 = alternate)

Example:
1
0
```

Appendix C

Compilation Flags Reference

C.1 GCC/Clang Flags

Flag	Purpose
-std=c++17	Enable C++17 standard
-Wall	Enable all warnings
-Wextra	Enable extra warnings
-O2	Optimization level 2
-O3	Maximum optimization
-g	Include debugging symbols
-I<path>	Add include directory
-L<path>	Add library directory
-l<lib>	Link library

Table C.1: Common compiler flags

C.2 Recommended Debug Build

```
g++ *.cpp -o chess_game_debug \
    -std=c++17 -Wall -Wextra -g \
    -lsfml-graphics -lsfml-window -lsfml-system -lsfml-audio
```

C.3 Recommended Release Build

```
g++ *.cpp -o chess_game \
    -std=c++17 -O3 -DNDEBUG \
    -lsfml-graphics -lsfml-window -lsfml-system -lsfml-audio \
    -s  # Strip symbols for smaller executable
```

Appendix D

Quick Reference Guide

D.1 Keyboard Shortcuts

Key	Action
P	Pause menu
S	Save game
ESC	Exit current screen
Enter	Play again (end screen)
Space	Play again (end screen)

D.2 Piece Values (Standard)

Piece	Value (Centipawns)
Pawn	100
Knight	320
Bishop	330
Rook	500
Queen	900
King	(invaluable)

D.3 Common Chess Notation

Symbol	Meaning
K	King
Q	Queen
R	Rook
B	Bishop
N	Knight
(none)	Pawn
x	Captures
+	Check
#	Checkmate
O-O	Kingside castling
O-O-O	Queenside castling

Appendix E

Glossary

Artifact Binary executable file produced by compilation

Bitboard 64-bit integer representation of chess board

Castling Special king-rook move (not implemented)

Check King is under attack

Checkmate King in check with no legal moves (game over)

En Passant Special pawn capture (not implemented)

FEN Forsyth-Edwards Notation for board positions

Minimax Decision algorithm for game trees

PGN Portable Game Notation for chess games

Promotion Pawn reaching opposite end transforms (not implemented)

SFML Simple and Fast Multimedia Library

Sprite 2D image used in game graphics

Stalemate No legal moves but not in check (draw)

Texture Image loaded into graphics memory

Bibliography

[1] Laurent Gomila et al., *SFML Documentation*, SFML Development Team, 2023, https://www.sfml-dev.org/documentation/2.6.2/

[2] FIDE, *Laws of Chess*, World Chess Federation, 2023, https://www.fide.com/FIDE/handbook/LawsOfChess.pdf

[3] *C++ Reference*, cppreference.com, 2024, https://en.cppreference.com/

[4] Gregory, Jason, *Game Engine Architecture*, CRC Press, Third Edition, 2018

[5] Campbell, Murray; Hoane, A. Joseph; Hsu, Feng-hlsfml-audio $^{-}std = c++17 - Wall$

E.0.1 Step 4: Copy SFML DLLs

Copy the following DLLs from `SFML-2.6.2\bin` to your executable directory:

- `sfml-graphics-2.dll`
- `sfml-window-2.dll`
- `sfml-system-2.dll`
- `sfml-audio-2.dll`

E.1 Linux Installation

E.1.1 Step 1: Install SFML

Ubuntu/Debian:

```
sudo apt-get update
sudo apt-get install libsfml-dev
```

Fedora:

```
sudo dnf install SFML-devel
```

Arch Linux:

```
sudo pacman -S sfml
```

E.1.2 Step 2: Install Build Tools

```
sudo apt-get install build-essential g++
```

E.1.3 Step 3: Compile the Project

```
g++ main.cpp board.cpp moves.cpp render.cpp menu.cpp \
    checkmate.cpp save.cpp highscores.cpp settings.cpp \
    ending.cpp -o chess_game \
    -lsfml-graphics -lsfml-window -lsfml-system -
    \subsection{Step 4: Run the Game}

\begin{lstlisting}[language=bash]
./chess_game
```

Listing E.1: Linux compilation command

Bibliography

[1] Laurent Gomila et al., *SFML Documentation*, SFML Development Team, 2023, https://www.sfml-dev.org/documentation/2.6.2/

[2] FIDE, *Laws of Chess*, World Chess Federation, 2023, https://www.fide.com/FIDE/handbook/LawsOfChess.pdf

[3] *C++ Reference*, cppreference.com, 2024, https://en.cppreference.com/

[4] Gregory, Jason, *Game Engine Architecture*, CRC Press, Third Edition, 2018.

[5] Campbell, Murray; Hoane, A. Joseph; Hsu, Feng-hsiung, "Deep Blue", *Artificial Intelligence*, Vol. 134, Issues 1–2, pp. 57–83, 2002.

[6] Stockfish Chess Engine Development Team, *Stockfish*, Open-source chess engine, https://stockfishchess.org/

[7] Chess Programming Wiki Contributors, *Bitboards*, Chess Programming Wiki, 2024, https://www.chessprogramming.org/Bitboards