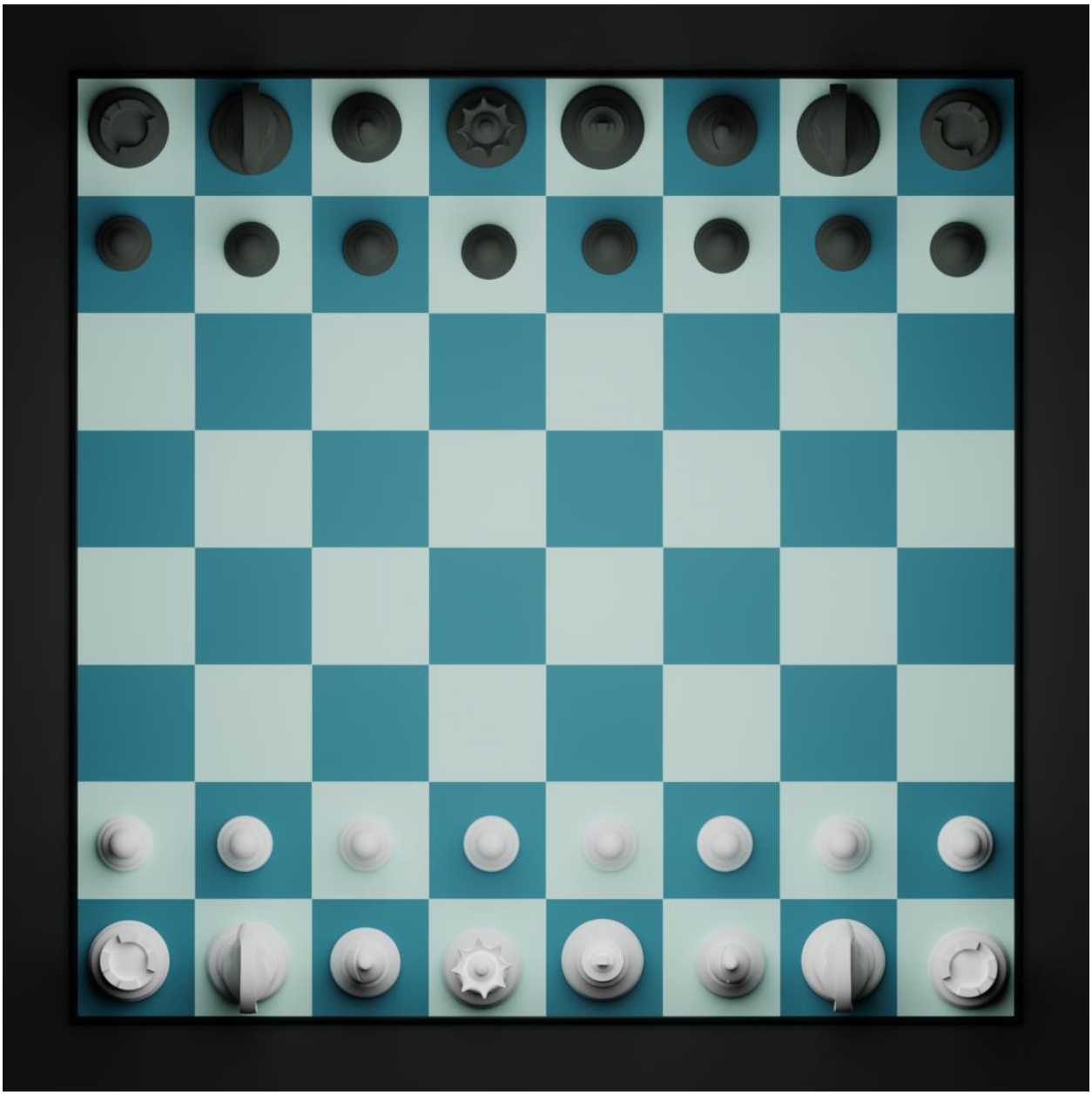
Project Title: Chess Engine in C++

Submitted by: Syed Taha

ERP: 29208

Date of Submission: May 27, 2024



Contents

Introduction

The Chess Engine (Horizon) project aims to develop a sophisticated software system capable of simulating chess games, providing players with an interactive and challenging gaming experience. This document serves as comprehensive documentation for the Chess Engine, detailing its design, implementation, and usage.

Purpose and Scope

The primary objective of the Chess Engine is to create a platform where players can engage in chess matches against the computer or other players. The project encompasses the development of various components, including the game logic, user interface, artificial intelligence algorithms, and supporting utilities.

Technologies Used

The Chess Engine is implemented in C++, leveraging its powerful features for object-oriented programming and performance optimization. Additionally, the project utilizes standard libraries and frameworks for graphical user interface development and system-level operations.

Design Overview

The Design Overview section provides an insight into the architecture and organization of the Chess Engine project. It outlines the key components and their interactions within the system.

Architecture

The Chess Engine follows a modular architecture, dividing the system into several distinct components that manage specific functionalities. These components include the Board, Engine, Functors, Namespaces, and Others.

Component Descriptions

Board

Contains classes responsible for managing the chessboard, game state, and graphical representation.

Includes the ChessBoard class for game logic, BoardState class for storing game states, and GraphicalBoard class for graphical rendering.

Engine

Consists of classes related to the chess engine's core functionality, such as move generation, evaluation, and search algorithms.

Includes the ChessEngine class for controlling the game flow, EngineState class for managing engine states, and TranspositionTables class for storing transposition data.

Exceptions

Consist of custom Exception classes to handle various Exceptions.

Includes KingNotFound Exception class.

Functors

Houses functor classes that perform specific tasks or operations within the engine.

Includes CalculateELO, CheckAfterMove, ConvertNotation, ChessDebugger, isCaptureMove, and SortMoves functors.

GameFLow

Contains class that manage the GameFlow.

Includes GameModes, Menu, Settings, Flags

Namespaces

Contains utility namespaces that organize related functions and classes.

Includes the Utility namespace for general utility functions.

Other

Encompasses various additional classes, files, and resources used throughout the project.

Includes classes such as Timer and User, along with global variables and constants.

Statistics

Contains classes and constants for representing the Game statistics during the game, on the GUI.

Includes BoardStats, StatisticalConstants.

Overall Interaction

The components of the Chess Engine interact through well-defined interfaces and communication channels, facilitating modularity, extensibility, and maintainability.

Board Directory

The "Board" directory contains classes and structures related to managing the chessboard and its state.

Classes:

ChessBoard Class: Represents the chessboard itself, including the arrangement of pieces.

Provides methods for initializing the board, making moves, and checking game state.

GraphicalBoard Class:

Represents the graphical interface for displaying the chessboard.

Provides methods for rendering the board and pieces on the screen.

Structures:

BoardState struct:

Represents the state of the chessboard, including information about the current position, player turn, and move history.

ChessBoard (Class)

The ChessBoard class represents the chessboard and manages the state of the chess game. It contains methods for initializing the board, computing valid moves for pieces, making moves, checking for check and checkmate, and other operations related to the game state.

Attributes:

state: An instance of the BoardState class that stores the current state of the chess game.

board: A pointer array representing the chessboard, where each element points to a ChessPiece object.

Constructor:

ChessBoard(): Initializes a new instance of the ChessBoard class.

Methods:

GetPieceAtPosition(int index) const: Returns the ChessPiece object at the specified index on the board.

calculatePlayerScore(int playerColor) const: Calculates the score of the specified player based on the current board state.

getAttacksOnSquare(int squareIndex, int opponentColor) const: Computes the number of attacks on a given square by pieces of the specified opponent color.

GetKingIndex(const int& playercolor) const: Returns the index of the king belonging to the specified player color.

AddMoveToHistory(std::string move): Adds a move to the move history.

ComputeKingMoves(int KingIndex) const: Computes valid moves for the king at the specified index.

ComputeKnightMoves(int pieceIndex) const: Computes valid moves for a knight at the specified index.

ComputePawnMoves(int pieceIndex) const: Computes valid moves for a pawn at the specified index.

ComputeSlidingPieceMoves(int pieceIndex) const: Computes valid moves for sliding pieces (queen, rook, bishop) at the specified index.

ComputeOpponentMoves(): Calculates opponent moves and stores them.

DestroyBoard(): Destroys the current board.

DisplayScores() const: Displays player scores.

initializeBoard(): Initializes the board with hardcoded values.

initializeBoardFromFEN(const std::string& fen, bool loadTextures): Initializes the board from a given FEN string.

InitializeDefaultBoard(): Initializes the board using the default FEN string.

MakeMove(int fromTile, int toTile): Makes a move on the board.

MakeCompleteMove(int fromTile, int toTile, std::string move): Makes a complete move on the board.

PlayChessSound() const: Plays a chess sound.

promotePawn(int toTile): Promotes a pawn.

ReverseBoard(): Reverses the board in Multiplayer

saveCurrentFENtofile(std::string file) const: Saves the current FEN position to a file.

SetPiecePositions(): Sets the position of chess pieces on the GUI.

UpdateChessPiece(ChessPiece* piece, int InitialIndex): Updates a chess piece.

canCastleKingSide(int KingIndex) const: Checks if castling kingside is possible.

canCastleQueenide(int KingIndex) const: Checks if castling queenside is possible.

isCheck(const ChessBoard& chessboard, const int playerColor): Checks if a player is in check.

isCheckmate(int player = EMPTY) const: Checks if the game is in checkmate for the given player. If no player is given then checks if the Current player(player whose turn it is to move) is checkmated

isCurrentPlayerWhite() const: Checks if the current player is white.

IsTileUnderAttack(int squareIndex) const: Checks if a tile is under attack.

isValidCaptureMove(int fromTile, int toTile) const: Checks if a capture move is valid.

isValidMove(int index): Checks if a move is valid.

IsEnPassantCapture(int fromTile, int toTile) const: Checks if an en passant capture is valid.

IsEnPassantLegal(int pawnIndex, int targetIndex) const: Checks if en passant is legal.

FilterValidMoves(int fromIndex, std::vector<int> possibleMoves) const: Filters valid moves from a list of moves.

GetAllPossibleMoves(int playerColor) const: Gets all moves for a player.

GetAllPossibleMovesForPiece(int type, int index, bool FilterInvalidMoves) const: Gets all moves for a specific piece.

GetAllPossibleMovesInChessNotation(int playerColor) const: Gets all moves in chess notation.

GetCurrentFEN() const: Gets the current FEN position of the board.

LoadTextures(): Loads textures for each piece on the board.

getCurrentPlayer() const: Gets the current player.

getCheckedPlayer() const: Gets the checked player.

operator<<(std::ostream& os, const ChessBoard& chessboard): overloaded the “<<” operator to display the board on terminal when used with std::cout. Displays the current state of the chessboard on the terminal, with each tile represented by the piece type (e.g., "0" for an empty tile, "1" for a king, "2" for a pawn, etc.). The chessboard is displayed on the terminal with ranks and files labeled, allowing for easy visualization of the board state.

Destructor:

~ChessBoard(): Destructs the ChessBoard object, deallocating memory for pieces.

GraphicalBoard (Class)

The GraphicalBoard class is responsible for rendering the graphical interface of the chessboard.

Attributes:

CoordinateIndexPairs: contains Precalculated Indices and the coordinates that need to be drawn, in the form <index, <coordinate, isRow>>;

Methods:

DrawBoard(const ChessBoard& chessboard) const: Draws the tiles of the chessboard.

DrawCoordinates(const ChessBoard& chessboard) const: Draws the coordinates on the chessboard (e.g., A1, B2, etc.).

DrawSquareIndices(const ChessBoard& chessboard) const: Displays the indices of the squares on the chessboard (e.g., 0 to 63).

DrawChessPiece(const ChessBoard& chessboard) const: Draws the chess pieces on the chessboard.

initCoordinatesIndexPairs(): Initializes and Pre-stores Tiles where Coordinates need to be drawn

getPosition(bool isRow, int LocX, int LocY) const: Returns the position to draw a coordinate.

Note:

The GraphicalBoard class encapsulates methods for drawing various elements of the chessboard, including tiles, coordinates, square indices, and chess pieces.

BoardState (Struct)

The BoardState struct represents the state of the chessboard during gameplay.

Attributes:

PreComputed Moves: SlidingPieceMoveData, PawnMoveData, KnightMoveData, KingMoveData: Precomputed moves for different types of pieces.

OpponentMoves: Precomputed moves for the opponent's pieces.

MovesForSelectedPiece: Precomputed moves for the selected piece by the user.

Move History:

moveHistory: Contains a history of all moves made during the game.

MoveIndices: Stores the indices of the last move made in the format <Initial Index, Final Index>.

Player and Turn Information:

currentPlayerIsWhite: Indicates whether it is currently the white player's turn to play.

checkedPlayer: Indicates the player (white or black) who is in check.

Game State Flags:

DoCastle: Determines whether the move made is a castling move.

isBoardReversed: Indicates whether the board is reversed (for multiplayer mode).

PieceIsCaptured: Indicates whether a piece was captured in the last move.

enPassantTarget: Stores the index of the pawn that is capturable by en passant.

Constructor:

Initializes default values for the attributes of the BoardState struct.

Methods:

operator=: Overloaded assignment operator to assign the state of one BoardState object to another.

getCurrentPlayer(): Returns the color of the current player.

reset(): Resets the state of the BoardState object.

Scores:

Scores assigned to each type of chess piece (pawn, rook, knight, bishop, queen, king).

Note:

The BoardState struct maintains valuable information about the game's state, including precomputed moves, move history, player turn, and game state flags.

Engine Directory

The "Engine" directory contains classes and structures related to the chess engine.

Classes:

ChessEngine Class: Represents the chess engine itself, responsible for making moves and evaluating positions.

Provides methods for searching for the best move, analyzing positions, and making moves for the Engine

EngineState Class: Represents the state of the chess engine during gameplay.

Stores information such as the current search depth and other engine parameters.

TranspositionTables Class: Represents transposition tables used for storing and retrieving previously computed positions during search. Improves search efficiency by avoiding redundant computations and storing valuable information.

Structures:

ChessEngineStatistics Struct: Stores statistics and information about the performance of the chess engine. Includes data such as the number of nodes searched, the time taken for search, and other relevant metrics.

PieceSquareTables Struct: Stores piece-square tables used for evaluating the positional value of pieces on the board. Contains predefined tables for different types of pieces and board positions.

ChessEngine (Class)

The ChessEngine class represents the chess engine responsible for making moves and evaluating positions during gameplay.

Attributes:

state: An instance of the EngineState class, representing the state of the chess engine during gameplay.

Constructor:

ChessEngine(int Color = EMPTY): Initializes the chess engine with an optional parameter specifying the color of the player controlled by the engine.

Move Searching Functions:

shuffleMoves(std::vector<std::string>& possibleMoves): Randomly shuffles the order of moves to introduce variety in move selection.

GenerateMove(const ChessBoard& board): Generates the best move for the current position using the minimax algorithm with alpha-beta pruning.

Minimax Search:

Minimax(ChessBoard& board, int depth, int alpha, int beta, auto time, int currentPlayer): Implements the minimax algorithm with alpha-beta pruning to search for the best move up to a specified depth.

Evaluation Functions:

Evaluate(const ChessBoard& chessboard, int currentPlayerColor) const: Evaluates the current position on the chessboard to determine its desirability for the given player.

Move Execution:

PlayMove(const std::string& move, ChessBoard& board) const: Executes the specified move on the given chessboard.

Note:

The ChessEngine class encapsulates methods for move generation, move searching using minimax, position evaluation, and move execution. It relies on the PieceSquareTables class for positional evaluation and the EngineState class for maintaining the engine's state during gameplay.

ChessEngineStatistics (Struct)

The ChessEngineStatistics struct contains data related to the performance and statistics of the chess engine during gameplay.

Attributes:

NumberofMovesLookedAhead: The number of moves looked ahead during the search process.

BranchesPruned: The number of branches pruned during search, contributing to search efficiency.

TranspositionsFound: The number of transpositions found in the transposition table.

TimeTaken: The total time taken for the search process in seconds.

SizeOfTable: The size of the transposition table in memory.

Speed: The speed of the engine's search process, measured in nodes per second (n/s).

maxDepth: The maximum depth reached during the search process.

totalMoves: The total number of moves considered during the search.

movesEvaluated: The number of moves evaluated during the search.

totalMovesToEvaluate: The total number of moves to be evaluated during the search.

Methods:

reset(): Resets all statistics to their initial values.

External Data:

Heuristics: An instance of the ChessEngineStatistics struct, defined externally in the EngineState.cpp file.

Note:

The ChessEngineStatistics struct provides valuable information about the performance and behavior of the chess engine during gameplay, including search depth, pruning statistics, time taken, and speed.

EngineState (Class)

The EngineState class represents the state of the chess engine during gameplay and search.

Attributes:

EngineColor: Represents the color the engine plays as (default is Black).

MAX_DEPTH: The maximum depth for the minimax algorithm.

transpositionTable: Transposition tables used to store positional data.

terminateSearch: Flag to terminate the search process.

startSearch: Flag to indicate the start of the search process.

timeLimit: Time limit for the search process (0 means no time limit).

useTranspositions: Flag defining whether to use transpositions in search.

useAlphaBetaPruning: Flag defining whether to use alpha-beta pruning in search.

pieceValues: A mapping of piece types to their corresponding values.

engineEloRating: The Elo rating of the chess engine.

Constructor:

EngineState(): Initializes the engine state with default values.

Methods:

setEngineColor(int color): Sets the color the engine plays as.

LoadSavedSettings(): Loads the various saved settings

TerminateSearch(): Sets the terminate search flag to stop the search process.

isSearchStarted() const: Checks if the search process has started.

StartSearch(): Sets the start search flag to indicate the beginning of the search.

StopSearching(): Sets the stop flag to stop the search process.

getELOFromSetting(): Loads the Elo from the saved settings

getELO(): Returns the engines ELO rating

setELO(): Sets the engines Elo rating

getDepth() const: Returns the current search depth.

setDepth(int NewDepth): Sets the search depth to a new value.

SaveTranspositionTable(): Saves the transposition table to a file.

getSizeOfTranspositionTable() const: Returns the size of the transposition table.

getPieceValue(int index) const: Returns the value of a piece based on its type.

reset(): Resets the engine state to its initial values.

Note:

The EngineState class manages the state of the chess engine, including search parameters, transposition tables, and engine Elo rating.

TranspositionTables (Class)

The TranspositionTables class represents the transposition tables used for storing positions encountered during the search process in the chess engine.

Attributes:

zobristKeys: A 2D array representing the Zobrist keys used for hashing positions.

transpositionTable: An unordered map storing the transposition table entries, where the key is the hash value of the position, and the value is a pair containing the score and depth information.

SizeOfTranspostionTable: The size of the transposition table.

Constructor:

TranspositionTables(): Initializes the transposition tables.

Methods:

initZobristKeys(): Initializes the Zobrist keys used for hashing positions.

storeTranspositionTable(uint64_t hash, int score, int depth): Stores a position in the transposition table along with its score and depth information.

ComputeSizeOfTranspositionTable(): Computes the size of the transposition table.

saveTranspositionTableToFile(std::string filename): Saves the transposition table entries to a file for debugging and testing purposes.

computeHash(const ChessBoard& board) const: Computes the hash value of a given chessboard position.

lookupTranspositionTable(uint64_t hash) const: Looks up a position in the transposition table based on its hash value and returns its score and depth information.

isValuePresent(uint64_t hash): Checks if a given hash value is present in the transposition table.

GetSizeOfTable() const: Returns the size of transposition table.

Note:

Transposition tables are used to store previously evaluated positions during the search process to avoid redundant evaluations and improve search efficiency.

Exceptions Directory

Contains Various Exception Classes.

1) KingNotFound

KingNotFound

The KingNotFound exception class is derived from the standard std::exception class and is used to indicate the absence of a king at a specified position on the chessboard.

Private Members

int position: Stores the position on the chessboard where the king was not found.

std::string message: Stores the message describing the exception.

Constructor

explicit KingNotFound (int position): Constructs a KingNotFound exception object with the specified position where the king was not found.

Member Functions

const char* what () const noexcept override: Overrides the what() method from std::exception to provide a description of the exception. Returns a C-style string containing a message indicating the position where the king was not found.

Functors Directory

This directory contains various functor classes utilized within the chess engine for different purposes. Functors are function objects that can be customized through operator overloading, making them flexible for a range of operations.

Contents:

CalculateELO(functor class): Calculates the Elo rating based on game results and player ratings.

CheckAfterMove(functor class): Checks the game state after a move is made, including checking for checkmate, stalemate, or other conditions.

ConvertNotation(functor class): Converts between different chess notations, such as algebraic notation and coordinate notation.

ChessDebugger(functor class): Provides debugging information during runtime, aiding in development and troubleshooting.

isCaptureMove(functor class): Determines if a move results in capturing an opponent's piece.

SortMoves(functor class): Sorts available moves based on certain criteria, enhancing move selection strategies.

CalculateELO (Functor)

The CalculateELO class provides functionality to calculate the Elo rating of players based on game outcomes and their current ratings.

Methods:

calculateExpectedScore(int PlayerARating, int PlayerBRating): Calculates the expected score for Player A based on their rating and the rating of their opponent, Player B.

operator()(int PlayerARating, int PlayerBRating, bool PlayerAWon): Overloaded function call operator that takes the ratings of both players and a boolean indicating whether Player A won the game. It returns the updated Elo rating for Player A after the game.

Note:

The Elo rating system is commonly used in chess and other competitive games to estimate the skill level of players based on their performance against opponents of varying ratings. The CalculateELO class encapsulates the logic for updating player ratings according to the outcome of games.

ConvertNotation (Functor)

The ConvertNotation functor provides functionality for converting between different chess notations and indices on the chessboard.

Methods:

operator()(const std::string& move) const: Takes a move string in algebraic notation format (e.g., "e2e4"). Returns a pair of integers representing the indices on the chessboard corresponding to the given move. For example, "e2e4" would be converted to the pair <60, 44>, where 60 represents the index of the starting tile and 44 represents the index of the destination tile.

operator()(int fromTile, int toTile) const: Takes two integers representing the indices of the starting and destination tiles on the chessboard. Returns a string representing the move in algebraic notation format. For example, given fromTile = 12 and toTile = 28, it would return "c2c4".

operator()(int pieceIndex) const: Takes an integer representing the index of a tile on the chessboard. Returns a pair of integers representing the coordinates (file and rank) of the tile on the chessboard. For example, given pieceIndex = 27, it would return the pair <4, 3>, representing the tile at row 4 and column 3 on the chessboard.

Note:

The ConvertNotation functor is used to facilitate conversions between different representations of moves and positions on the chessboard, allowing for easier interaction with the chess engine and user interface.

ChessDebugger (Functor)

The ChessDebugger functor provides functionality for debugging and displaying information on the terminal.

Methods:

static void moves()(const std::vectorstd::string& moves): Takes a vector of strings representing moves. Prints each move in the vector to the terminal, separated by spaces, followed by a newline.

print_args(std::ostream& os): Base case function to print the last argument. This function is used in the recursive variadic template function to terminate the recursion.

print_args(std::ostream& os, const T& firstArg, const Args&... args): Recursive variadic template function to print arguments. This function prints the first argument to the output stream, then recursively calls itself with the remaining arguments until all arguments are printed.

operator()(Args&&... args): Wrapper function to allow usage with std::cout. This function forwards the arguments to the print_args function, enabling convenient usage with std::cout for displaying multiple arguments on the terminal.

Note:

The ChessDebugger functor is a versatile tool for debugging and displaying several types of information on the terminal, including move sequences and chessboard states. It aids in understanding the internal state of the chess engine during development and testing phases.

IsCaptureMove (Functor)

The IsCaptureMove functor determines whether a given move results in a capture of an opponent's piece on the chessboard.

Methods:

operator()(const std::string& move, const ChessBoard board) const:

Parameters: Takes a string representing the move in algebraic notation and a ChessBoard object representing the current state of the chessboard.

Output: Returns a boolean value indicating whether the move results in a capture. If the move involves capturing an opponent's piece, the function returns true; otherwise, it returns false.

Note:

The IsCaptureMove functor is useful for detecting capture moves during the move generation process in the chess engine. It helps in identifying moves that lead to material gain or loss on the chessboard, which is crucial for evaluating the desirability of different moves during the search process.

GameFlow Directory

The GameFlow directory contains classes and components related to managing the flow and control of the chess game, including handling game modes, settings, flags, and the main menu.

Classes:

Flags: The Flags class manages various flags and states related to the game flow, such as indicating whether a game mode has started or if certain settings are enabled.

GameModes: The GameModes class manages different game modes available in the chess game, such as single-player and multiplayer modes. It orchestrates the logic and flow of each game mode, including setting up the game, managing player moves, and controlling game state transitions.

Menu Class: The Menu class is responsible for displaying and managing the main menu of the chess game. It manages user interactions with menu options and provides functionality for starting the game, accessing settings, and other menu-related tasks.

Settings Class: The Settings class manages game settings and configurations, allowing players to customize various aspects of the game, such as difficulty level, time limits, and display preferences.

Note:

These components work together to ensure smooth game flow and provide players with options to customize their gaming experience.

Flags (Class)

The Flags class manages various flags and settings related to the game state and user interface in the Chess Engine project.

Attributes:

isMoveMade: Indicates whether a move has been made.

SinglePlayer: Indicates if the game is in single-player mode.

MultiplayerGame: Indicates if the game is in multiplayer mode.

gameStarted: Indicates if the game has started.

showSettings: Indicates if settings are being displayed.

SetFEN: Indicates if the FEN settings are being displayed.

Constructor:

Flags(): Initializes an instance of the Flags class.

Methods:

CheckGameState(): Checks the game state.

GameStateChecked(): Indicates that the game state has been checked.

MoveIsMade(): Checks if a move has been made.

StartGame(): Starts the game.

isMultiplayerGame(): Checks if the game is in multiplayer mode.

isSinglePlayer(): Checks if the game is in single-player mode.

isGameStarted(): Checks if the game has started.

EndGame(): Ends the game.

SinglePlayerMode(): Sets the game mode to single player.

MultiplayerMode(): Sets the game mode to multiplayer.

DisableSinglePlayer(): Disables single-player mode.

OpenSettings(): Opens the settings.

SettingsOpened(): Checks if settings are open.

closeSettings(): Closes the settings.

OpenFENSettings(): Opens the FEN settings.

isFENSettingsOpened(): Checks if the FEN settings are open.

closeFENSettings(): Closes the FEN settings.

Menu (Class)

The Menu class manages the user interface elements and options for the game menu.

Attributes:

InfoBox: Rectangle representing the information box.

StartButton: Rectangle representing the start game button.

MultiplayerButton: Rectangle representing the multiplayer mode button.

FENButton: Rectangle representing the load game from FEN button.

SettingsMenu: Rectangle representing the settings menu button.

height: Height of the menu buttons.

width: Width of the menu buttons.

buttonX: X-coordinate for button placement.

StartButtonY: Y-coordinate for the start game button.

MultiPlayerbuttonY: Y-coordinate for the multiplayer mode button.

FenButtonY: Y-coordinate for the load game from FEN button.

SettingMenuY: Y-coordinate for the settings menu button.

Constructor:

Menu(): Initializes the Menu object.

Methods:

ShowMenu(): Displays the game menu.

isOptionPressed() const: Checks if any menu option is pressed.

DrawMenuBox() const: Draws the menu box.

Note:

The Menu class manages the display and interaction of menu options in the game interface.

GameModes (Class)

The GameModes class manages different game modes and their associated logic within the chess engine.

Attributes:

DoOnce: Flag to execute certain tasks once per match.

FENString: Stores the FEN string representing the current game state.

graphics: Instance of the GraphicalBoard class for graphical board rendering.

engine_depth: Depth setting for the chess engine.

Game Related Objects:

Player: User object representing the player.

GameStats: BoardStats object for managing game statistics.

chessboard: ChessBoard object representing the game board.

Horizon: ChessEngine object for game logic and AI opponent.

GameMenu: Menu object for the game's menu

Constructor:

GameModes(): Initializes the GameModes object.

Methods:

Options(): Displays game options.

InitialiseMultiplayerMode(): Sets up multiplayer mode.

MultiplayerMode(): Executes logic for multiplayer mode.

InitialiseSinglePlayerMode(): Sets up single-player mode.

SinglePlayerMode(): Executes logic for single-player mode.

HandleMoves(int): Handles player moves based on the player's color.

DisplayBoard() const: Displays the game board on the GUI.

ResetBoard(bool): Resets or restarts the game.

RestartGame(): Restarts the game.

BackToMenu(): Returns to the main menu.

UpdateElos(): Calculates player Elo ratings.

setFENstring(std::string): Sets the FEN string representing the game state.

FENSettings(): Displays FEN settings.

SaveTranspositions(): Saves transposition tables.

BoardSetUp(): Sets up the game board.

Settings(): Displays game settings.

Destroy(): Destroys the GameModes object.

GameLoop(): Logic for the game loop

Note:

The GameModes class manages game modes, including single-player and multiplayer modes, along with game setup, logic, and settings.

Settings (Class)

The Settings class manages various settings and configurations for the Chess Engine project.

Attributes:

filename: A constant string representing the filename for storing settings.

Static Attributes:

depth: A string representing the search depth for the engine.

engineElo: A string representing the Elo rating of the engine.

userElo: A string representing the Elo rating of the user.

Constructor:

The Settings class does not have a constructor as all attributes and methods are static.

Methods:

save(int depth, int userElo, int engineElo): Saves the settings to a file.

get(const std::string& element): Retrieves a specific setting value.

saveElement(const std::string& element, int value): Saves a specific setting element with a value.

Namespaces Directory

The "Namespaces" directory contains namespaces used to organize and encapsulate related functions and classes.

Contents:

Utility Namespace: Contains utility functions and classes for common tasks such as string manipulation, file I/O, and other general-purpose utilities.

Purpose:

The namespaces within the "Namespaces" directory aim to promote code organization, encapsulation, and reusability by grouping related functionalities.

Utility (Namespace)

The Utility namespace provides utility functions for common tasks.

Components:

AddCommas Function: Adds commas to an integer number for better readability.

CloseInBrackets Function: Adds enclosing brackets around a string.

DrawTextWithCustomFont Function: Draws text with a custom font at a specified position and size on the screen.

SetPrecision Function: Sets the floating-point precision of a number.

TextCenter Function: Calculates the coordinates of the center of a text for proper positioning.

UnloadFontsAndSounds Function: Unloads all fonts and sounds.

LoadFontsAndSounds Function: Loads fonts and sounds

SuppressRaylibLog Function: Suppress Raylib Log messages.

Engine_Calculations Function: Logic for Engine thread move calculations.

Note:

The Utility namespace provides essential utility functions for various tasks such as text manipulation, formatting, and drawing.

Other Directory

The "Other" directory contains miscellaneous classes, files, and constants used in the chess engine.

Classes:

Timer Class: Implements a timer for tracking elapsed time during gameplay.

User Class: Represents the user and manages user-related operations in the chess engine.

GlobalVariables File: Contains global variables used across the chess engine.

Resources File: Contains constants for file names and paths used in the chess engine.

Note:

The "Other" directory contains various classes, files, and constants essential for different functionalities in the chess engine.

Timer (Class)

The Timer class provides functionality to measure elapsed time during the execution of code segments.

Attributes:

start_time: A std::chrono::time_point object representing the start time of the timer.

Constructor:

Timer(): Initializes the timer by capturing the current time using std::chrono::steady_clock::now().

Destructor:

~Timer(): Destructs the timer and calculates the elapsed time since its creation. It then outputs the elapsed time in seconds to the console.

Methods:

The Timer class does not have any additional methods beyond its constructor and destructor.

User (Class)

The User class represents a player in the chess game.

Attributes:

Username: The name of the user.

ELO: The Elo rating of the user.

filename: The filename for storing the user's Elo rating.

Constructor:

User(std::string name = "", int elo = EMPTY): Initializes a user object with the specified name and Elo rating.

Methods:

setUserName(std::string name): Sets the username of the user.

Note:

The User class stores information about a player, including their username and Elo rating. ***This is still a Basic Implementation and has limited applications in the current version.***

Global Variables

Sounds:

ChessPiecePlaced: Represents the sound played when a chess piece is placed on the board.

ChessPieceCaptured: Represents the sound played when a chess piece is captured.

KingChecked: Represents the sound played when a king is checked.

GameStarts: Represents the sound played when the game starts.

GameEnds: Represents the sound played when the game ends.

Fonts:

myFont: Represents the font used for text rendering.

fontSize: Default font size

Window:

screenWidth: Width of the game window.

screenHeight: Height of the game window.

Board:

boardSize: Size of the chessboard (number of squares per side).

tileSize: Size of each square on the chessboard.

Total_tiles: Total number of tiles on the chessboard.

MaxHistorySize: Maximum size of the move history.

ReverseOffset: Offset for reversing the board.

BoardOffsetX: Horizontal offset for positioning the board.

BoardOffsetY: Vertical offset for positioning the board.

Game State:

infinity: Represents the maximum value for certain calculations.

InfoBox:

Segments: Number of segments in the info box.

Roundedness: Roundedness of the info box.

InfoBoxX, InfoBoxY: Position of the info box.

InfoBoxWidth, InfoBoxHeight: Dimensions of the info box.

Colors:

lightSquare, darkSquare: Colors representing light and dark squares on the chessboard.

Transparent, Background, Translucent: Colors for transparency and background.

MoveHighlight, MovesForPieceHighLight, MoveHighlightRed, NextMoveHighlight: Colors used for highlighting moves.

Resources

Audio Files:

ChessPiecePlacedFile: Path to the audio file played when a chess piece is placed on the board.

ChessPieceCapturedFile: Path to the audio file played when a chess piece is captured.

KingCheckedFile: Path to the audio file played when a king is checked.

GameStartsFile: Path to the audio file played when the game starts.

GameEndsFile: Path to the audio file played when the game ends.

Font File:

fontFile: Path to the TrueType font file used for text rendering in the program.

Program Icon:

program_icon: Path to the icon file used for the program's icon.

Note:

These resource paths specify the location of audio files, font files, and program icons used in the Chess Engine project. These resources are typically loaded into the program during runtime for use in various functionalities such as audio playback, text rendering, and graphical user interface.

Pieces Directory

The "Pieces" directory contains classes related to different types of chess pieces.

Classes:

ChessPiece Class: Represents a generic chess piece. Provides methods and attributes common to all chess pieces. Subclasses are inherited from this class to represent specific types of pieces.

ChessPiece (Class)

The ChessPiece class represents a generic chess piece in the game.

Attributes:

pawnScore, rookScore, knightScore, bishopScore, queenScore, kingScore: Scores assigned to each type of chess piece.

type: An integer representing the type of the piece.

color: An integer representing the color of the piece.

firstMove: A boolean indicating whether the piece has made its first move.

isSlidingPiece: A boolean indicating whether the piece is a sliding piece (rook, bishop, or queen).

texture: A Texture2D object representing the texture of the piece.

rectangle: A Rectangle object representing the position and size of the piece on the game board.

isDragged: A boolean indicating whether the piece is currently being dragged by the player.

canCastleQueenSide, canCastleKingSide: Booleans indicating whether the piece can castle queenside or kingside.

PieceCode: An integer representing the piece code (used for identification).

EnpassantTarget: An integer representing the target square for en passant capture.

Constructor:

ChessPiece(int pieceType = EMPTY, int pieceColor = EMPTY, bool isFirstMove = true): Initializes a chess piece with the specified type, color, and first move status.

Methods:

AssignTextures(): Loads textures based on the piece code.

DestroyTextures(): Unloads textures associated with the piece.

getScore(): Returns the score assigned to the piece. (This method is virtual and overridden in subclasses.)

clone(): returns a new ChessPiece copy of *this object

Piece Type Codes:

Piece type codes are used to identify the type and color of a chess piece. Each piece is assigned a unique piece code. See the commented section in the class for the mapping of piece type codes to piece types.

Note: The ChessPiece class serves as a base class for specific types of chess pieces. It provides common attributes and methods shared by all pieces.

Pieces (Classes)

The piece classes (Pawn, Rook, Queen, Bishop, Knight, and King) inherit from the ChessPiece class, which represents a generic chess piece.

Inheritance:

Each piece class inherits common attributes and methods from the ChessPiece class.

Constructor:

The constructor of each piece class initializes the piece with the specified type, color, and first move status.

Overloaded methods:

Clone(): returns the new object of the *this object

External:

PST: An instance of the PieceSquareTables struct, storing piece-square tables used for positional evaluation. Defined in Pieces.cpp

Methods:

The getScore() method returns the score assigned to the specific piece type.

PieceSquareTables (Struct)

The PieceSquareTables struct represents the piece-square tables used for evaluating the positional advantage of pieces on the chessboard.

Attributes:

Table: A 2D array representing the piece-square tables for each piece type and each square on the board.

InvertedTable: A 2D array representing the inverted piece-square tables for each piece type and each square on the board.

Constructor:

PieceSquareTables(): Initializes the piece-square tables and their inverted versions.

Methods:

getPSTValue(ChessPiece* piece, int squareIndex, char currentPlayerColor) const: Retrieves the value of the piece at a given square index considering the current player's color.

Note:

Piece-square tables are used by the engine to evaluate the positional strength of pieces on the board. The values in these tables represent the positional advantage or disadvantage of placing a particular piece on a specific square on the board.

A Global Instance on PieceSquareTables is defined in Pieces.h and Pieces.cpp

Statistics Directory

The Statistics directory encompasses classes and structures dedicated to tracking and displaying statistical information during gameplay.

BoardStats Class

The BoardStats class is responsible for collecting and presenting statistics related to the current state of the game. It calculates and displays information such as move history, player Elo ratings, and game outcomes. Additionally, it manages the visualization of game statistics on the graphical user interface (GUI), providing players with valuable insights into their gameplay.

StatisticalConstants Struct

The StatisticalConstants struct holds constant values and messages used throughout the statistics-related functionalities. It defines parameters such as text size, color schemes, and predefined messages for displaying game statistics. These constants ensure consistency and facilitate the presentation of statistical information in a uniform manner across the game interface.

Together, the BoardStats class and StatisticalConstants struct contribute to enhancing the player experience by offering informative and visually appealing displays of game statistics during gameplay.

BoardStats (Class)

The BoardStats class manages game statistics and information related to the chessboard.

Attributes:

whiteProportion: Proportion of Game in favor of white

blackProportion: Proportion of Game in favor of Black

TimerStart: Time point representing the start of the game timer.

constants: Instance of the StatisticalConstants struct containing statistical constants.

ShowMoveHistory: Determines whether to show move history.

SaveData: Determines whether to save game data.

winner: Indicates the winner of the game.

EndMessage: Message indicating the game outcome.

DisplayMoveHistory: Determined whether to how move history or not.

Constructor:

BoardStats(): Initializes the BoardStats object.

Methods:

DisplayEndMessage(): Displays the end message indicating the game outcome.

DisplayStats(ChessBoard&, ChessEngine&, User&): Displays game statistics in single-player mode.

DisplayStats(ChessBoard&): Displays game statistics in multiplayer mode.

MovesAndHistory(std::string, const std::vector<std::string>&): Displays the last move or move history based on the flag.

getData(const ChessEngine&, const User&, const ChessBoard&, std::vector<std::string>&): Retrieves game data.

DrawPlayerElos(int, int): Draws player Elo ratings.

DrawStatistics(const std::vector<std::string>&): Draws game statistics.

DisplayPlayerTitles(const ChessBoard&): Displays player titles.

Reset(): Resets game statistics.

DrawMoveHistory(const std::vector<std::string>&): Draws move history on GUDrawEvaluationColumn(ChessBoard&): Draws the evaluation column.

Evaluate(const ChessBoard&, int): Evaluates the game state, for the given player

GameIsEnded(ChessBoard&): Checks if the game has ended.

DisplayNewDepthMessage(const int&): Displays a message for a new search depth.

DisplayNewFENMessage(const std::string&): Displays a message for a new FEN.

toggleHistory(): Turns history on or off.

getWinner() const: Returns the winner.

ShowMoveHistory(): returns dispayMoveHistory

Note:

The BoardStats class manages game statistics, move history, and end-game messages for both single-player and multiplayer modes.

StatisticalConstants (Struct)

The StatisticalConstants struct holds constant values used for game statistics and messages.

Attributes:

TextSize: Constant text size for display.

textX: X-coordinate reference point based on the dimensions of the information box on the GUI.

textY: Y-coordinate reference point for text display.

BoardDimensions: Calculation of the height and width of the game board.

messageColor: Color constant for message display.

AlertColor: Color constant for alert messages.

BlackInCheck: Constant message indicating Black's King is in check.

WhiteInCheck: Constant message indicating White's King is in check.

ELO_text: Constant message prefix for Elo rating.

Black_: Constant message for “Black.”

White_: Constant message for “White.”

MinScore: Constant minimum score value after which the game draws

get_white_pos(): Function to calculate the position of "White" text on the GUI.

get_black_pos(): Function to calculate the position of "Black" text on the GUI.

Note:

The StatisticalConstants struct provides constant values and messages used for game statistics and display.

Piece Codes

Piece Type Codes:

EMPTY: Represents an empty square on the chessboard.

KING: Represents a king piece.

PAWN: Represents a pawn piece.

KNIGHT: Represents a knight piece.

BISHOP: Represents a bishop piece.

ROOK: Represents a rook piece.

QUEEN: Represents a queen piece.

Color Codes:

White: Represents pieces belonging to the white player.

Black: Represents pieces belonging to the Black player.

Note:

Piece codes are typically used to identify the type and color of a chess piece on the board. The combination of a piece type code and a color code uniquely identifies a specific piece. For example, a white pawn can be represented by combining the PAWN code with the White code.

User Guide

Features

The Current Release v5.2.0 Includes the Following Features

- Singleplayer Mode(Against AI)
- Multiplayer Mode
- Loading Game From FEN
- Settings to adjust Depth

Installation

To run the chess engine, follow these steps:

Method 1

Dowload The .msi file Provided in the realeases section and follow the setup. (This will Create a shortcut on your desktop and Startmenu)

Method 2

If you recieve a security warning, then you can download the zip file, and run the .msi file provided there under Installer/Chess/Debug.

Method 3

Alternatively, you can also open the porject on Visual Studio, and build the project from there.

Usage

Click on the shortcut on your Desktop. Choose a game mode.

- Start (Single Player)
- Multiplayer
- Load from FEN
- Settings

In Current Version, "Load from FEN" will work Only for Multiplayer Game

Navigating

- Press "r" to Restart.
- Press "m" to return to Menu.
- Press “c” to clear terminal.
- Press "esc” to close.

Configuration

The engine's behavior and settings can be configured in the following ways:

The Search depth Can be set inside settings. For more Control you can navigate over to ChessEngine.h header file and change the following Property MAX_DEPTH. To avoid long search times This is set to 3 by Default. You can Change this to any value, according to your device.

Changing MAX_DEPTH will increase Search times and might even cause the program to crash completely.

In case of Unexpected Exits, Weird Behaviors

In case of any weird behavior, like message not showing on Win, Engine calculation not stopping on game end etc., Kindly do the following.

1. Take a Screenshot
2. Close the program.
3. Navigate over to AppData\Roaming\ChessData
4. Email the UnexpectedExits.txt file to me at syetaha@gmail.com along with a screenshot/explanation of the error.

Alternatively report an issue on GitHub.

Concepts Used in the Project

Functional Programming

v5.2.1 introduces Functional Programming concept such as Functors.

Templates

V5.2.1 Also uses templates (used in the ChessDebugger class) for cleaner code.

Error Handling

V5.2.1 Further incorporates a better Exception Handling. Although there is still room for improvement.

Polymorphism

v5.2.0 uses polymorphism to Assign Chess Pieces to the Board. Instead of Using a single Class for Chess Pieces, I have derived Six Classes from the ChessPiece class, and then Dynamically Assigned Chess Pieces on the Board. (This is still a Very basic Implementation of Polymorphism)

Testing

Overview

The test_engine.cpp file is designed to evaluate the functionality of the chess engine by analyzing various board positions represented in Forsyth-Edwards Notation (FEN). It utilizes command-line arguments to process FEN strings and provide feedback on the engine's performance, including the best move found, time taken for analysis, and search depth.

Building the Test File

To build the test_engine.cpp file, follow these steps:

1. Open Terminal: Launch the terminal or command prompt on your system.
2. Navigate to Directory: Use the cd command to navigate to the directory containing the test_engine.cpp file.
3. Compile the File: Compile the test_engine.cpp file using:

```
g++ -o test_engine.exe test_engine.cpp ../src/Engine/EngineState.cpp
../headers/Engine/ChessEngineStatistics.h ../src/Engine/ChessEngine.cpp
../src/Board/ChessBoard.cpp ../src/GameFlow/Flags.cpp
../src/Engine/TranspositionTables.cpp ../src/Other/GlobalVariables.cpp
../src/GameFlow/Settings.cpp ../src/Pieces/ChessPiece.cpp ../src/Pieces/Pieces.cpp
../src/Functors/ConvertNotation.cpp ../src/Exceptions/KingNotFound.cpp -std=c++20 -
ljsoncpp -lraylib -lGL -lm -lpthread -ldl -lrt -lX11
```

Build Success: Upon successful compilation, an executable file named test_engine.exe will be generated in the same directory.

Testing Procedure

Prepare FEN Database: A file name ExtractedFENs.txt should be present in the directory. If not run FenExtractor.sh. using ./FenExtractor.sh

Execute Test Engine:

After the executable and FEN database are ready. Use start_test.sh to tun the test.

"Usage: $0 <FEN database file> [loop_limit] [Depth]"

By Default, Depth is two.

For Example:

1) ./start_test.sh ExtractedFENs.txt
2) ./start_test.sh ExtractedFENs.txt 10
3) ./start_test.sh ExtractedFENs.txt 10 3
4) ./start_test.sh ExtractedFENs.txt -1 3

Explanation:

1) Tests engine on All the FENs in the database
2) Tests engine on the first 10 FENS in the database
3) Tests engine on the first 10 FENS in the database at depth three
4) Tests engine on All the FENS in the database at depth three

Analysis and Feedback:

The test engine will process each FEN string from the database, analyzing the position and providing feedback on the engine's performance. Feedback may include:

- Best move found.
- Time taken for analysis.
- Search depth at which analyzed.
- Other relevant statistics

Save Data:

Data generated during testing, including FEN strings, analysis results, and performance metrics, will be saved to a .json file for further analysis.

Conclusion

The test_engine.cpp file serves as a valuable tool for evaluating the chess engine's performance under various conditions and board positions. By analyzing feedback and data generated during testing, we can identify areas for improvement and optimize the engine's algorithms for enhanced gameplay and efficiency.

Limitations of the Program

Pawn Promotion Implementation

The program's handling of pawn promotion is not fully implemented. Currently, pawns are automatically promoted to queens without providing options for other piece promotions.

Known Issue with engine’s Pawn promotion: After promoting Pawn, textures are not loaded.

Engine Testing Limitation

The chess engine has not been thoroughly assessed as White. It can only play as Black during gameplay and analysis. This limitation may affect the engine's performance and strategic decisions.

Limited FEN Loading Functionality

The feature to load a game from Forsyth-Edwards Notation (FEN) strings is only functional for multiplayer mode. In single-player mode against the AI, FEN loading is not supported.

Performance and Effectiveness of the Engine

The chess engine included in the program is slow and may not provide optimal gameplay experiences. Additionally, its strategic decision-making and overall effectiveness may be limited compared to more advanced or optimized engines. Further development and optimization efforts are required to enhance the engine's performance.

Future Enhancements

Utilization of Smart Pointers for Board Management:

Implementing smart pointers, such as std::unique_ptr or std::shared_ptr, for managing the board data structure can improve memory management and reduce the risk of memory leaks.

Transition to More Efficient GUI Libraries:

Consider transitioning from Raylib to more efficient and feature-rich GUI libraries like SFML (Simple and Fast Multimedia Library) or SDL (Simple DirectMedia Layer). These libraries offer enhanced graphics rendering capabilities and better cross-platform support.

Integration of Bitboards for Board Representation:

Incorporating bitboards for board representation can significantly improve the efficiency of move generation and evaluation algorithms. Bitboards allow for fast bitwise operations, enabling more efficient board manipulation and evaluation.

Enhancement of Search and Evaluation Functions

1) Improve the chess engine's search algorithms by implementing techniques like iterative deepening, and principal variation search (PVS) to enhance search efficiency and depth.
2) Implement Quiescence Search to address the horizon effect, allowing the engine to explore deeper into critical positions by considering capturing moves and other forcing moves beyond the search horizon.
3) Enhance the evaluation function by considering additional factors such as piece mobility, pawn structure, king safety, and positional advantages. Utilizing more sophisticated evaluation heuristics can lead to better strategic decision-making by the engine.
4) Explore machine learning techniques, such as neural networks, to develop more advanced evaluation functions that can learn from game data and adapt to various playing styles and positions.

Project Timeline:

January 2024:

1) Project Idea selection.
2) Initial Research.
3) Installing Libraries(Raylib)
4) Basic Implementation of Chess Board, Piece representations, move handling.

February 2024:

1) Submitted Milestone 1:
    - Make a basic chess game with all the rules
    - Loading games from FEN strings
    - Make a GUI using Raylib or SFML
    - Implement a Bot that plays randomly
    - Implement an AI that plays strategically
    - Improvements to AI
    - Writing efficient and reusable code, by dividing parts into multiple files
    - Have AI beat stockfish.(If possible)

Challenges faced So far:

- Implementing the Board: 8x8 array or an Array with 64 elements?
- Implementing Piece movements
- Implementing Piece Specific Rules
- Checks and Checkmates.
- Pawn Promotion

Aims achieved:

- Added a GUI
- piece specific rules
- loading games from FEN strings
- Piece movement.
- Sound effects
- Basic AI with Minimax algorithm

March 2024:

1) Further enhancements to the Chess Engine.
2) Experimenting with various Enhancement techniques like Alpha-Beta Pruning, Quiescence Search, iterative Deepening etc.
3) Started using Git and GitHub for version control instead of multiple folders for each version
4) Basic Error Handling
5) Added Testing files and related Bash scripts

6) Refactored code to use Polymorphism for ChessPieces
7) Added Installation files.
8) Implemented Basic file handling to save and read settings.

April 2024:

1) Structured Project files.
2) Refactored classes in to smaller classes
3) Compiled related Variables to separate structs
4) Changed Implementation to incorporate Functional Programming, and lambda functions.
5) Fixed Several issues related to App closing unexpectedly, segmentation faults,
6) Removed a lot of redundant code.
7) Added comments.
8) Started work on this documentation
9) Fixed an issue that arose in the test script, that caused it to not compile
10) Implemented better File handling. This time saving data in the AppData folder. This resolved an issue with the installed version of the game which caused it to crash when trying the save data to the installation directory(due to lack of permissions)

May 2024:

1) Applied Abstraction
2) Significant improvements to the Chess Engines Evaluation function.
3) Completion of Chess Documentation

Conclusion

In the journey of developing the chess engine "Horizon," I have embarked on a quest to explore the intricacies of chess programming, confront challenges, and strive for continuous improvement. Through meticulous design, implementation, and iteration, I have designed a chess engine that encapsulates both the triumphs and the trials of the development process.

Reflection on Accomplishments

Feature-Rich Functionality:

Horizon boasts a range of features, including single-player and multiplayer modes, support for loading games from FEN strings, and customizable settings such as search depth adjustments.

User-Friendly Interface:

With an intuitive menu system, graphical board representation, and informative game statistics display, Horizon offers a user-friendly experience for players of all levels.

Testing and Debugging:

Extensive testing, both manual and automated, have been conducted to ensure the reliability and robustness of the engine. Debugging efforts have addressed numerous issues and improved the overall stability of the program.

Acknowledging Limitations

Despite all these achievements, it is essential to acknowledge the limitations and challenges encountered during the development process:

Incomplete Features:

Certain features, such as pawn promotion and evaluating the engine's performance as White, remain incomplete and require further refinement.

Performance Concerns:

Horizon's performance is currently suboptimal, characterized by slow search times and weak gameplay. Addressing these performance concerns is a priority for future enhancements.

Looking to the Future

As we look ahead, the journey of Horizon is far from over. Future enhancements and refinements, are envisioned to elevate the engine to new heights of sophistication and strength:

Enhanced Engine Strength:

Implementing advanced search algorithms, refining the evaluation function, and leveraging innovative techniques like machine learning hold the promise of significantly enhancing Horizon's playing strength.

Optimization and Efficiency:

Exploring optimization opportunities, such as transitioning to smart pointers and adopting more efficient GUI libraries, will improve the engine's performance and resource utilization.

Continued Learning and Innovation:

Embracing a culture of continuous learning and innovation, I will remain vigilant in my pursuit of excellence, seeking out innovative ideas, technologies, and strategies to propel Horizon forward.

References

League, S. (2021, February 12). *Coding adventure: Chess*. YouTube. https://www.youtube.com/watch?v=U4ogK0MIzqk

League, S. (2023, June 30). *Coding adventure: Making A better chess bot*. YouTube. https://www.youtube.com/watch?v=_vqlIPDR2TU

Main page. Chessprogramming wiki. (n.d.). https://www.chessprogramming.org/Main_Page