Software Specification

Minor Carlson

Version 1.0

Team 21:
Anthony Flores
Chenyu Wang
Davis Nguyen
Nathan Bae
Timothy Chan

Table of Contents

Glossary

Chess Pieces:

Pawn: The Pawn can only move forward. It can only attack diagonally. On its first move it can either move forward once or twice. Once it reaches the completely other side of the board it is able to transform into either a Bishop, Knight, Queen, or Rook.

Rook: The Rook can move vertically or horizontally for as many spaces it wants given there isn't any piece in the way. It can attack any piece horizontal or vertical.

Knight: The Knight can move in L shape so either two moves vertically and one move horizontally or two moves horizontally and one move vertically. It is able to jump over pieces. It can attack any piece it lands on.

Bishop: The Bishop can move diagonally as many spaces as it wants. It can not jump over pieces and it attacks diagonally as well.

Queen: The Queen can move diagonally, vertically, or horizontally as many spaces as it wants. It can not jump over pieces and it attacks diagonally, vertically, or horizontally.

King: The King can move diagonally, vertically, or horizontally but for only one space. It can not jump over pieces and it attacks diagonally, vertically, or horizontally.

Chess Moves and Outcomes:

Castling: Is a special move where the Ring and Rook simultaneously move towards each other. The king moves two squares towards the rook and the rook moves right next to the king on the opposite side. Castling can only be done when the King and Rook have yet to move and there are no pieces between them.

Enpassant: this special move involves the pawns. This move consists of a pawn being able to capture a pawn right after the pawn has taken two moves to avoid capture. The capture must be done right after the pawn's two space evasion. The capturing pawn can diagonally attack the opposing pawn at the square right behind the opposing pawn.

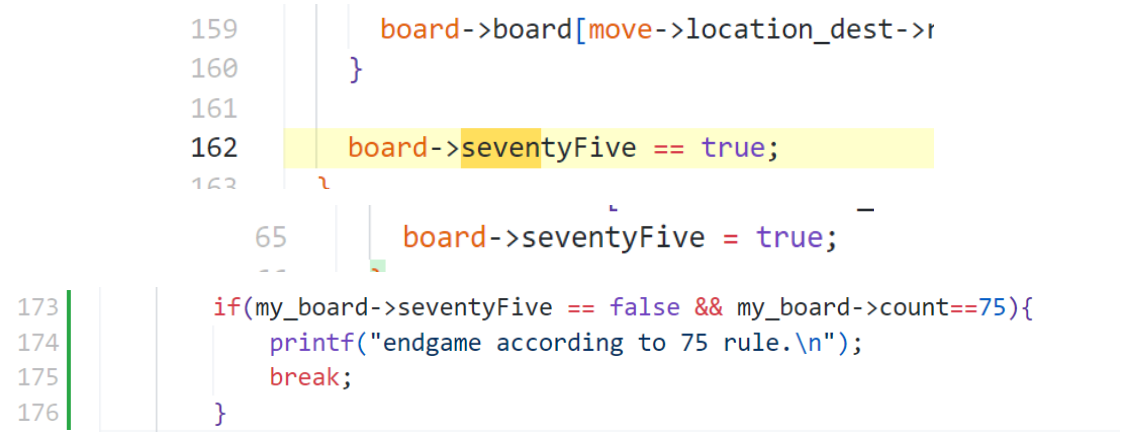
Check: When the king is being threatened it is called check. When an opposing piece has the opportunity to capture the King in the next turn if no preventative action is taken this is called

“Check”. Check forces the threatened player to take action to prevent his king from being captured in the next turn.

Checkmate: When in Check the defending player has no protective action available to save their King.

Draws:

Stalemate: This occurs when a player has no possible moves and is not in Check.
Impossibility of Checkmate: This occurs when there aren't enough pieces in each of the players to produce a Checkmate.
75-move-rule: If no pawn moves or capture occurs within 75 moves, it is an automatic draw.



(code for implementing 75 rule)

Draw agreement: Players are allowed to agree on a draw at any point in the game.

Software Architecture Overview

1.1 Main Data types and Structures

```
struct Board
{
    struct Piece *board[8][8];
    enum Player current_player;
    bool whiteHasCastled;
    bool blackHasCastled;
};
```

```
enum PieceType
{
    KING,
    QUEEN,
    ROOK,
    BISHOP,
    KNIGHT,
    PAWN,
    EMPTY_Piece
};
```

```
enum Player
{
    BLACK,
    WHITE,
    EMPTY_Player
};
```

```
struct Piece
{
    char name[3];
    enum PieceType t_piece;
    enum Player player;
    bool moved;
    bool captured;
    bool enpass;
};
struct Move
```

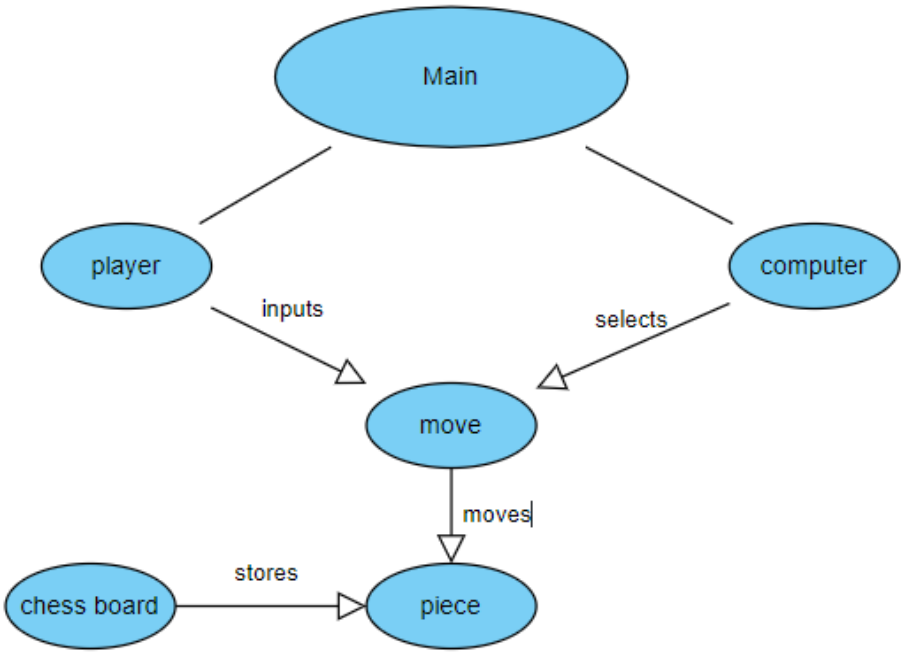
```
{
    struct Location *location_src;
    struct Location *location_dest;
    struct Piece *captured_piece;
    bool castling;
    bool enpassmove;
};
```

```
    struct Moves
    {
        int size;
        struct Move moveList[300];
    };
```

```
    struct Location
    {
        int rank;
        int file;
    };
```

Log: text file

1.2 Major Software components

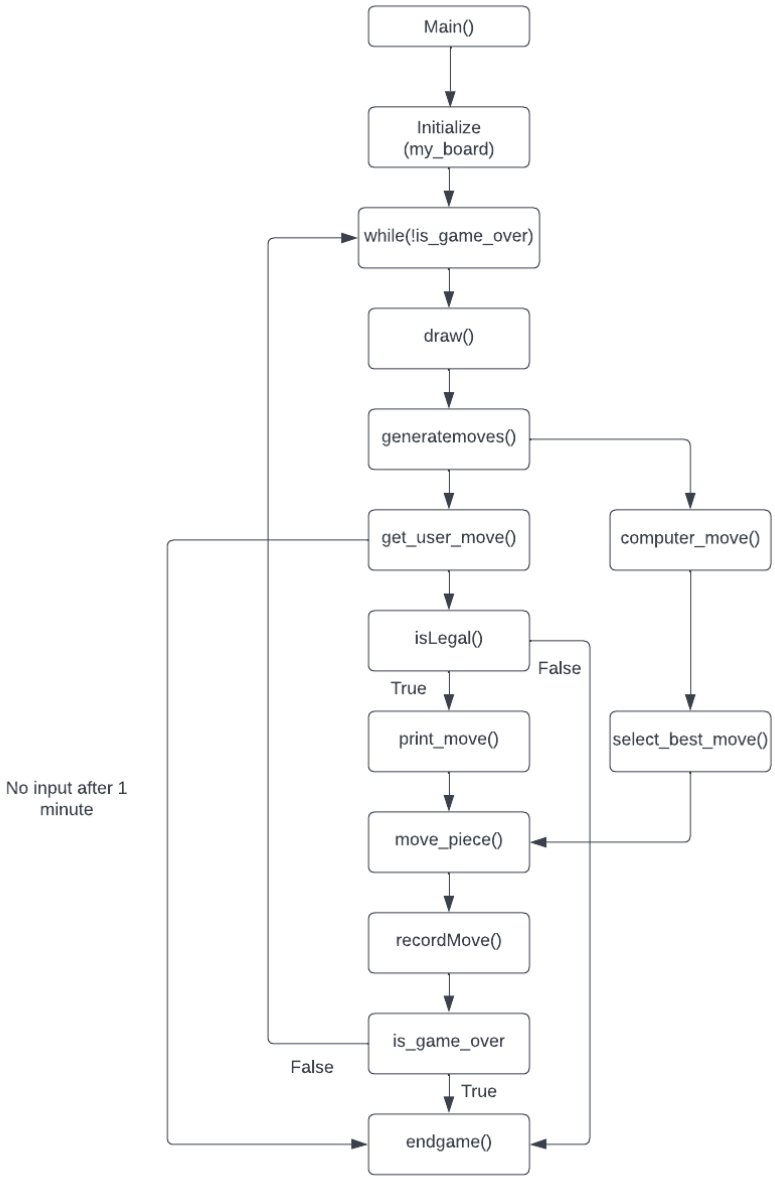


1.3 Module Interfaces

```
extern void initialize(struct Board *board);
extern void print_moves(struct Moves *moves, struct Board *position);
extern bool isGameOver(struct Board *board);
extern int inCheck(struct Board *board, enum Player playerColor, struct Moves *moves);
extern void generatemoves(struct Board *position, struct Moves *p_list, enum Player playerturn);
extern int straight_moves_count(struct Board *position, int dir, int rank, int file);
extern void move_piece(struct Move *move, struct Board *board);
extern void draw(struct Board *board);
extern bool isLegal(struct Move *move, struct Moves *moves);
extern void recordMove(struct Move *move, enum Player curr_player, FILE *fp);
extern int get_user_move(struct Move *move);
extern int reload(FILE *log, struct Board *board);
extern void print_move(struct Move* move);
extern int print_files();
```

1.4 Overall program control flow

After opening the chess program, the user is asked to choose between two game modes through the use of the main function. The first game mode is Player vs. Player while the second game mode is Player vs. AI. Once the user chooses a game mode, they will be asked if they want to start black or white. After choosing the side, the game will begin. The chess board will then be generated with the function draw(). The player or computer that chose the white side will have the first turn. The function get_user_move() will be used for the program to read the user's input. Function isLegal() will then be used to decide if the user input is considered a legal move. Additionally, if the player does not input anything within the one minute, the game will just end, with the opposing side being announced the winner. If it is a legal move, the function move will move the chess piece towards the desired location with move_piece(). If the computer is the white side, the computer's move will be chosen through the select_best_move function. Then the move will be recorded in the log file with recordMove(). However, if it is an illegal move, the game will end with endgame(). After the legal move, the program will check if the user input leads to a checkmate or go back to reading the user input. Once is_game_over() becomes true, the game will end and the winner will be announced.



Installation

2.1 System requirements, compatibility

A functional computer (Either Windows or Mac) that is capable of running Linux. The machine can run this program in both a 32-bit and 64-bit operating system. This program requires little processing power to function. Additionally, an insignificant amount of hard drive space is required. It should be compatible with other computers running a similar program to be able to function against each other.

2.2 Setup and configuration

For the setup of the chess program, simply download the tar.gz file and choose the location of where you want the program to be placed. Then simply unpack the tar.gz and run the executable file in order to open up the program.

2.3 Building, compilation, installation

To install the chess program, the user will compile the chess program through its tar.gz file. The makefile will be used for the Linux system to compile all the necessary files required for the program to function. Typing “make all” will allow the command prompt to initialize the compiling process and compile the files needed. The installation is finished after all the files are compiled. To uninstall the program, simply delete the files that were compiled from the tar.gz file.

Packages, Modules, and Features

3.1 Data Structures

```
// Define the different piece types.
enum PieceType
{
   KING,
   QUEEN,
   ROOK,
   BISHOP,
   KNIGHT,
   PAWN,
   EMPTY_Piece
};
```

PieceType is an enum that defines the different types of pieces in chess.

```
// Define the different players
enum Player
{
   BLACK,
   WHITE,
   EMPTY_Player
};
```

Player is an enum that defines the two players in chess. If neither players' chess pieces, it will be categorized as EMPTY_Piece.

```
// Define a struct to represent a chess piece
struct Piece
{
   char name[3];
   enum PieceType t_piece;
   enum Player player;
   bool moved;
   bool captured;
   bool enpass;
};
```

Piece is a struct that represents a chess piece. It has a name (which is a char array of size 3), a type (as defined by PieceType), a player (as defined by Player), and two boolean values to track whether the piece has moved and whether it has been captured.

```
// Define a struct to represent a chess move
struct Move
{
   struct Location *location_src;
   struct Location *location_dest;
   struct Piece *captured_piece;
   bool castling;
   bool enpassmove;
};
```

Move is a struct that represents a move in chess. It has two Location structs to represent the starting and ending positions of the move, and a Piece struct to represent any captured piece.

```
//Define a struct to store a list of moves
struct Moves
{
   int size;
   struct Move moveList[300];
};
```

Moves is a struct that stores a list of the available Move structs.

```
// Define a struct to represent a chess board location
struct Location
{
   int rank;
   int file;
};
```

Location is a struct that represents a position on the chess board. It has two integers to represent the row and column.

```
// Define the chess board as an 8x8 array of Piece structs
struct Board{
   struct Piece* board[8][8];
   bool whiteHasCastled;
   bool blackHasCastled;
}
```

Board is a struct that represents the chess board. It is an 8x8 array of pointers to Piece structs, and it also has two boolean values to track whether each player has castled.

3.2 Functions

int main()

- Takes input of which game mode was chosen
- After deciding the gamemode, it will initialize the game
- There are 3 game modes (Player vs. Player, Player vs Computer, Computer vs. Computer)
- It will setup the chessboard and asks user which side they want (Black or White)
- Will call upon the draw function to create the chess board.
- The opposite side will be assigned to the computer
- After the game is finished, will create a log file of all moves chosen, including the chess piece type. Log file will be stored in the log folder and be titled as “chess_game.txt”
- Once the player wins or loses, will call upon the endgame function to end the game.

void move_piece(Move* move, Board *board)

Input:

- move: A pointer to the move to be made.
- board: A pointer to the current board state.

Process:

- This function uses the input information to move a certain piece to a certain location, if there’s capture involved, the capture will be automatically done and recorded in the log file
- Covers how pieces will move in special cases (en passant, castling, pawn promotion)

Move* computer_move(Player player, Board* board)

- Takes in the input of all the piece types on the board
- Uses a complex algorithm to decide the computer player’s best move
- Designed to predict the user’s moves several steps ahead
- Will output the computer player’s best move after evaluation
- Invoked in the main function when it is the computer player’s turn to move

void generatemoves(Player player, Board* board, Moves *p_list)

- Takes in the input of all the location of the pieces on the board and what piece type, including their color, is moving
- Returns an instance of struct Move, containing an array of all valid possible moves for each piece on the board for specified player color

void gentlemoves(int srcrank, int srcfile, int destrank, int destfile, struct Moves *p_list)

- This is where defensive moves can be initialized and stored

void attackmoves(int srcrank, int srcfile, int destrank, int destfile, struct Moves *p_list, struct Board *position)

- This is where offensive moves can be initialized and stored

void castlingmoves(int srcrank, int srcfile, int destrank, int destfile, struct Moves *p_list, struct Board *position)

- This is where castling moves can be initialized and stored

int get_user_move(struct Move *move)

Input:

- Takes the input of the user as a 4 character array like “e2e4”.

Process:

- It breaks it down into the source rank and file and the destination rank and file
- And then the move_piece function to make the move.
- Prompts the user to enter a move in algebraic notation, parses the input and stores the move in the move struct.
- Has a one minute timer for the user to input their move
- If the user does not input a move within the one minute, the player that ran out of time will lose

Output:

- Returns an integer value indicating whether the move was successfully parsed and stored in the move struct.
- Returns 1 if successful.
- Returns 0 if the move is invalid.

isLegal()

Input:

struct Board *board: a pointer to a Board struct representing the current chess board state.

- struct Move *move: a pointer to a Move struct containing the details of the move to be checked.
- struct Moves *moves: a pointer to a Moves struct containing the list of legal moves for the current player.
- enum Player playerColor: the color of the player making the move.

Process:

- Checks if the given move is legal for the given player, based on the current board state and the list of legal moves for the player.

Output:

- A boolean value of true if the move is legal, and false otherwise.

generateLegal()

Input:

- struct Board *board: a pointer to a Board struct representing the current chess board state.
- struct Moves *moves: a pointer to a Moves struct that will be filled with the list of legal moves for the current player.

extern void generateLegal(struct Board *board, struct Moves *moves, enum Player playerColor);

Input:

- board: A pointer to the current board state.
- moves: A pointer to a struct to store the legal moves.
- playerColor: The color of the player whose turn it is.

Process:

- Generates all legal moves for the player whose turn it is and stores them in the moves struct.

int inCheck(Board* board, Player playerColor)

Input:

- struct Board *board: a pointer to a Board struct representing the current chess board state.
- enum Player playerColor: the color of the player to be checked for check.

Process:

- Prints out warning to computer or to player if previous move resulted in a check
  Input: Board with current status
- This function takes the input data of the previous move and evaluates whether that move resulted in a check
- It will print a warning on the screen indicating that a check has happened and you must resolve the check
- If the board is incheck, it will return the number of checks

Outputs:

- An integer value of 1 if the player is in check, and 0 otherwise.
- struct Board *board: a pointer to a Board struct representing the current chess board state.
- struct Moves *moves: a pointer to a Moves struct containing the list of legal moves for the current player.

- enum Player playerColor: the color of the player to check for game over.

isGameOver()

Input:

- struct Board *board: a pointer to a Board struct representing the current chess board state.
- struct Moves *moves: a pointer to a Moves struct containing the list of legal moves for the current player.
- enum Player playerColor: the color of the player to check for game over.

Process:

- Checks if the game is over for the given player, based on whether they have any legal moves left.

Output:

- A boolean value of true if the game is over for the given player, and false otherwise.

void draw(Board *board)

Inputs:

- When it is called upon by the main function, it will print out the 8 by 8 chessboard created through a 2D array
- struct Board *board: a pointer to a Board struct representing the current chess board state.
- It will also print out all the piece types and their location on the board
- Uses a “for loop” to print out the information

void endgame()

Input:

- enum Player currentPlayer: the color of the player who lost the game.

Process:

- Will be invoked in the main function when the game ends or in function isLegal if an illegal move was performed
- Announces the result of the game and tells the user who wins the game
- Prints our white is the winner or black is the winner

void recordMove(struct Move *move, enum Player curr_player, FILE *fp)

Inputs:

- struct Move *move: A pointer to a Move struct representing the move to be recorded.
- enum Player curr_player: An enum Player representing the current player.
- FILE *fp: A file pointer to the log file to which the move will be recorded.

Process:

- Will print a message that will notify the user that the move has been recorded
- If the move can not be recorded due to an error, it will print “incorrect player information”

Outputs:

- Records the move to the specified log file.

int piece_value(enum PieceType t_piece)

- Gives each piece type a value so the computer ai can know how important each chess piece type is
- This will allow the ai to make intelligent decisions when playing against the player

int evaluate_move(struct Move *move, struct Board *board)

- This will evaluate the overall value of each move
- The value calculated is from the difference between the piece moving and the value of the piece it is trying to capture

Input:

- struct Move *move: a pointer to a Move struct containing the details of the move to be evaluated.
- struct Board *board: a pointer to a Board struct representing the current chess board state.

Outputs:

- An integer value representing the score of the move.

struct Move *select_best_move(struct Moves *p_list, struct Board *board)

- After the ai evaluates what move is the best to perform, this function allows the best move to be returned to the main file
- Function also accounts for predictability so it does not always choose the best move. (Will choose between several of the best moves)
- Invoked in the main file when it is the computer’s turn to move

int reload(FILE *log, struct Board* board)

Inputs:

- FILE *log: A file pointer to the log file from which to reload the game state.
- struct Board *board: A pointer to a Board struct representing the current state of the chess board.

Process:

- Used to store and record log files
- Can be be used to restore a game state if the user never finishes the game
- Reads all moves to see if they can be stored in the log file (Will notify the user if there is an error with storing the moves)

Outputs:

- Returns an integer value indicating the success of the reload operation. If the operation was successful, the function returns 0. Otherwise, it returns -1.

void sighandler(int signum)

- Used to notify the user if they run out of time during their turn

void print_moves(struct Moves *moves, struct Board *position)

Inputs:

- struct Moves *moves: A pointer to a Moves struct representing the list of moves to be printed.
- struct Board *position: A pointer to a Board struct representing the current state of the chess board.

Process:

- When invoked, the function will print out all the possible moves that the current player has
- Prints the details of the move in the format of "<piece name> from <source location> to <destination location>" (to console)

Outputs:

- Prints the list of moves to the console.

void initialize(struct Board *board)

- struct Board *board: a pointer to a Board struct representing the current chess board state.
- When invoked by the main function, it will print out the chessboard, setting up with all the chess piece types in the original locations

debug()

Input:

- struct Board *board: a pointer to a Board struct representing the current chess board state.

Process:

- Prints debugging information about the current board state.

Output:

- An integer value representing the number of pieces on the board.

Function: int straight_moves_count(struct Board *position, int dir, int rank, int file)

Inputs:

- struct Board *position: A pointer to a Board struct representing the current state of the chess board.
- int dir: An integer representing the direction of the move (either 1 or -1).
- int rank: An integer representing the rank of the starting position of the move.
- int file: An integer representing the file of the starting position of the move.

extern int print_files();

Process:

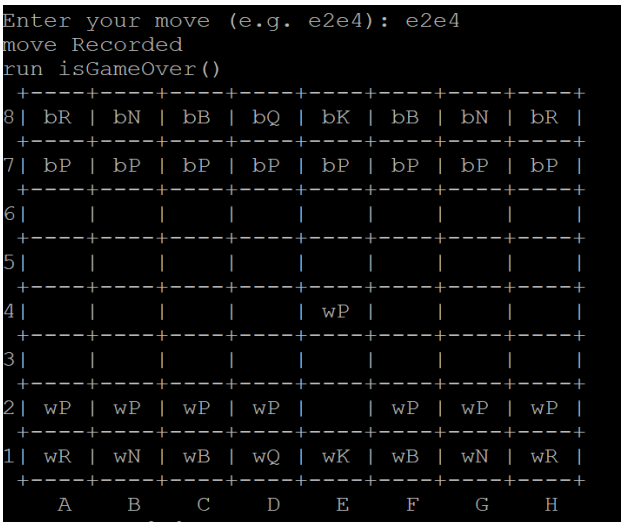
- Prints the names of all saved games in the current directory.

Outputs:

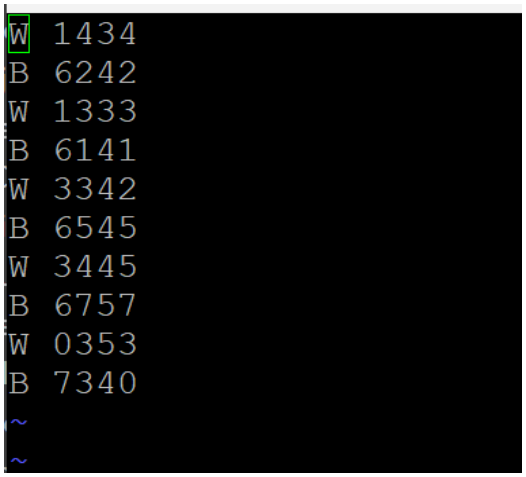
- Returns an integer value indicating the number of saved games.

3.3 Input and Output Formatting

User input: The input would be read in the form of a 4 character string like “A1A4” It would be stored in variables to store the initial position and the resulting position. Once the main function reads the user’s input, the input will then be checked to see if it is a legal move or not. If it is not a valid move, the game will end and the opposing side will win. If it is a valid move, the chess piece will move to its desired position from its initial position. Below is what entering the user input would look like.



Output: The log file would be recorded in a text file. It would be a list of 4 character strings. Each row would alternate between W and B for white and black. Then 4 numbers as the starting coordinates and the ending coordinates like “W 2444”. Below is an example of what the log file would look like.



Development Plan and Timeline

4.1 Partitioning of Tasks

All group members are tasked to complete the User Manual and Software Specification.

Data Structure, Game Modes, Log Files: Chenyu Wang
AI Creator: Davis Nguyen
User Input, Interface: Nathan Bae
Valid User Inputs: Anthony Flores
Checkmate and Ending Game: Timothy Chan

4.2 Team Member Responsibilities

The responsibilities of all team members will be divided upon completing the stated functions.

Anthony Flores: findAllMoves(), isLegal() – structures.h
Chenyu Wang: main(), structure header file,log file function –chess.c , log.c
Davis Nguyen: computer_move(), select_best_move(), main() — ai.c
Nathan Bae: move_piece(), draw() — board.c
Timothy Chan: inCheck(), endgame() –statusCheck.c

Back matter

• Copyright

• Error messages

Illegal Moves

- “There is no piece at selected location”
- “That piece cannot move to selected position”
- “That piece is not yours”
- “Your king is in check, you must resolve your check”

• Index

• References

http://www.wachusettchess.org/ChessGlossary.pdf
https://cdn.shptrn.com/media/mfg/1725/media_document/8976/Imp_ChessR.pdf?1401227342