



Department of Computer Science and Engineering
University of Dhaka

Project Report
Fundamentals of Programming Lab(CSE-1211)

Project Name
CHESS AI
Game Name
QUEEN'S GAMBIT

Submitted By
Group-25
Team Members

- Mahmudul Hasan (Roll: AE-20)
- Ahad Bin Islam Shoeb (Roll: FH-27)

INTRODUCTION

This is a conventional computer chess game. The whole game is written in C programming language. For Graphical user interface, SDL2 libraries have been used. The game can be played both in GUI and console mode. In GUI mode, two players can play the game with each other at the same time on the same computer. A player can also play against the computer in the Single Player mode. Conventional Chess rules have been implemented except Castling and EnPassant. A game can be postponed at any time and played further as the player wishes.

OBJECTIVES

The main objectives of the project are:

- i. To improve programming skills in C.
- ii. To improve problem solving skills.
- iii. To learn SDL and graphical programming.
- iv. To improve chess playing skill.
- v. Most importantly, for enjoyment.

PROJECT FEATURES

The main features are:

1. TWO Playing modes.
2. Four difficulty levels in ONE PLAYER or AI mode.
3. In NOOB and EASY difficulty a player can visualise all of his possible moves and threats.
4. Game can be saved for further playing
5. Moves can be undone in ONE PLAYER mode upto 3.

1. TWO PLAYING MODES:

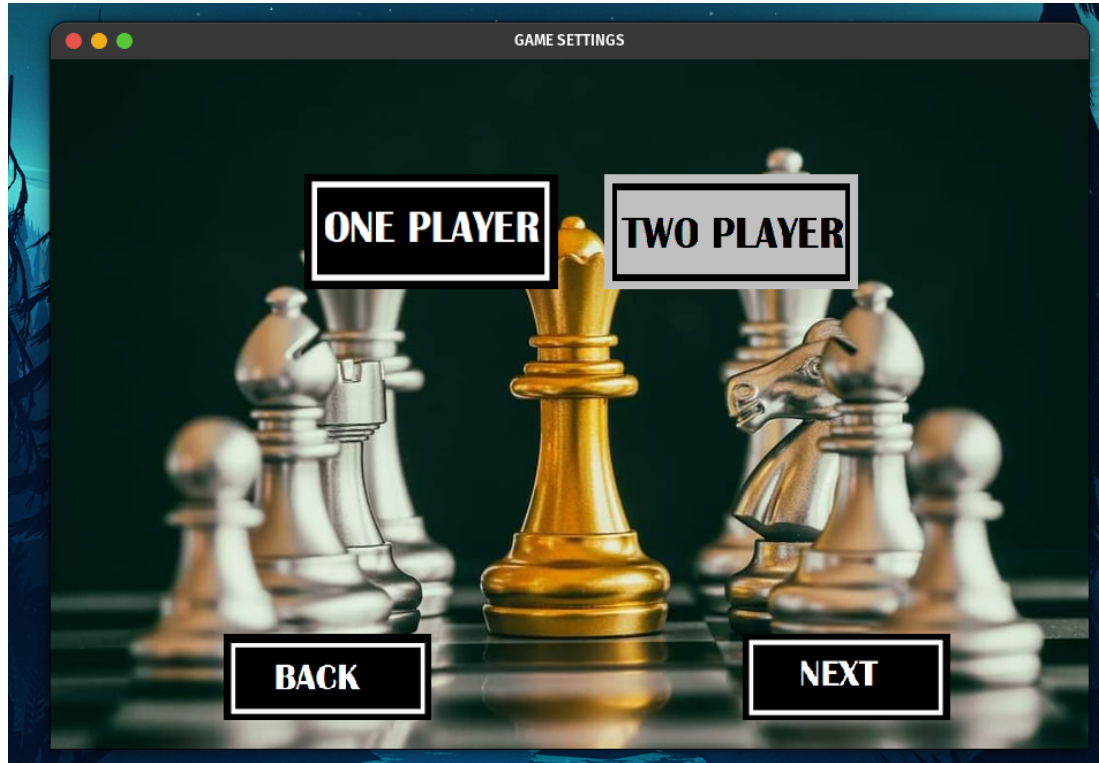


fig. Setting Window(Game Mode Setting)

ONE PLAYER: In ONE PLAYER mode, a chess player can play against the computer. The hardness of the game will be dependent on the level of selected difficulty. To implement the feature, MiniMax algorithm with alpha-beta pruning has been used.

TWO PLAYER: In TWO PLAYER mode, a person can play chess with another person. In two player mode the UNDO option is disabled.

2. FOUR DIFFICULTY LEVELS:

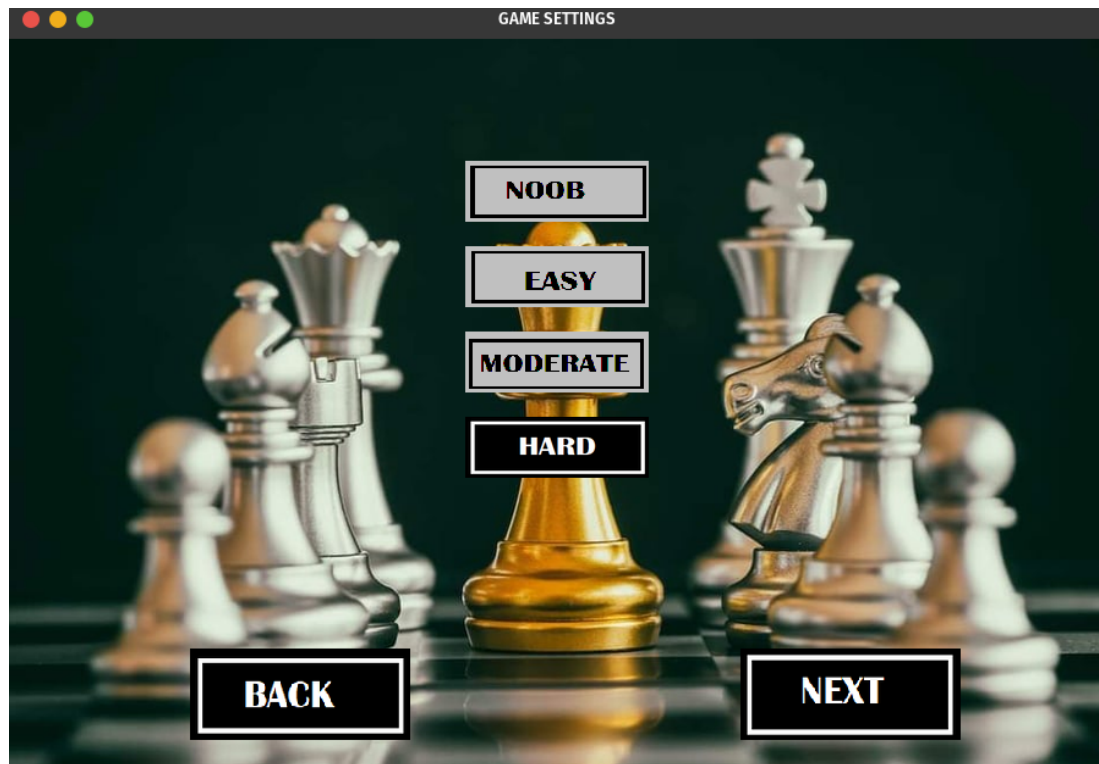


fig. Setting Window(Difficulty Setting)

NOOB: This is for absolute beginners who know a little about Chess. He/she can undo at most three previous moves and also can see the possible moves of a piece and it's threat.

EASY: Everything is the same as NOOB, but the difficulty increased a little bit.

MODERATE: Some features are unavailable in this level of difficulty. A player cannot undo his/her any move and also cannot visualise the possible moves of a particular piece.

HARD: Everything is the same as MODERATE, but the difficulty increased a little bit.

3. POSSIBLE MOVES and THREATS VISUALISER:



fig. Game Window

In the picture, it's being seen that some of the grids in the board are red, some of these are green, and one of these is yellow. The red grids are the grids, if the white queen makes move there, most probably it will be captured. The green grids are safe grids, if the queen makes move there, it will not be captured by the opponent in the next move. And the yellow grid is the grid where the queen can capture a piece without being captured by the opponent.

4. GAME SAVING:

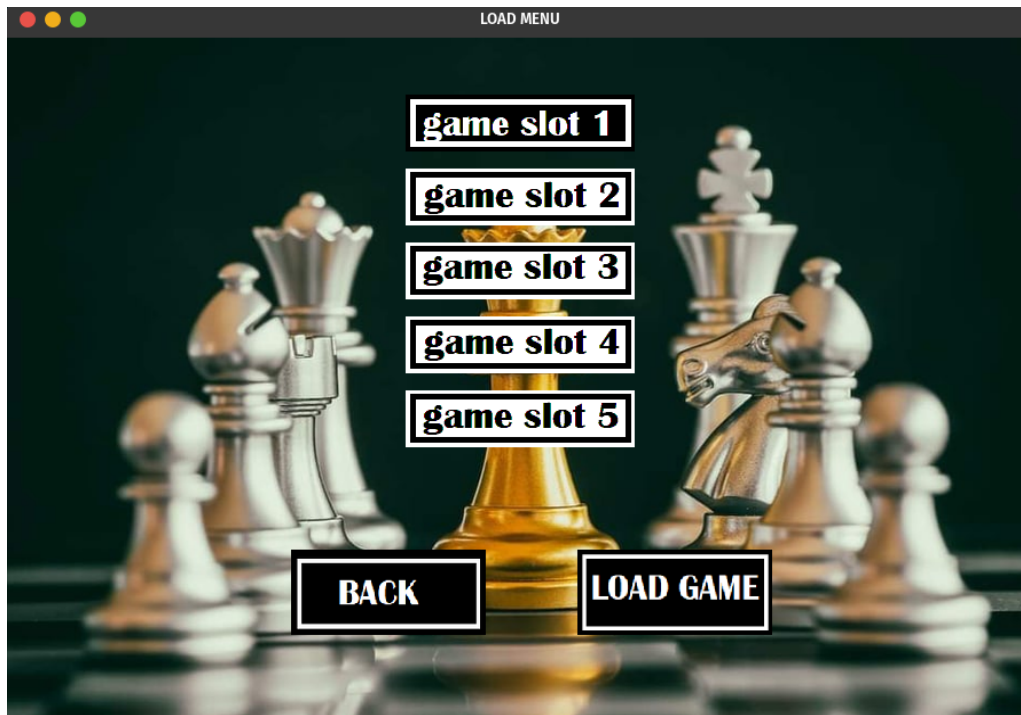


fig. Load window

A player can save his game and can continue the game anytime he/she wants. Games can be imported from external sources if the file format and data are represented in a proper way.

PROJECT MODULES

The whole game is divided into two main sections:

- i. **ENGINE:** The game logic, board data managing part.
- ii. **GUI:** Visual representation and user interaction managing part. It also controls and manages the game data.

ENGINE: The whole game runs depending on the core logic of the part. To run the game, ENGINE isn't dependent on GUI, rather GUI is dependent on ENGINE and can't do anything except it. The game can be played without GUI in console mode.

In this module, there're 8 C files and 7 header files. The uses of these header files and their containing functions are given below.

ArrayList.h

In brief, it contains the structure and arrays to hold the information about a particular move. It also contains the functions to push data to an array, to get data from an array, to erase data from an array etc. The details of the header are given below.

1. Contains the structure elem to save move history.
2. Contains the structure SPArrayList to store the list of moves.
3. ArrayListCreate(int maxSize) function creates an array with specified size to store move lists. The function has a parameter which is the maxSize of the array and it's an integer.
4. copyParams(SPArrayList *source, SPArrayList *target) function copies data source parameter to target parameter.
5. ArrayListIsFull function(SPArrayList *src) returns true if the list is full.
6. ArrayListIsEmpty(SPArrayList *src) Returns true if the list is empty, that is the number of elements in the list.
7. ArrayListSize(SPArrayList *src) Returns the number of elements in the list. The function is called with the assertion that all arguments are valid. If any of the arguments is invalid then an undefined value is returned.
8. ArrayListMaxCapacity(SPArrayList *src) Returns the maximum capacity of the list.
9. ArrayListCopy(SPArrayList *src) with the assertion that all arguments are valid. If any of the arguments is invalid then an undefined value is returned.
10. ArrayListDestroy(SPArrayList *src) frees all memory resources associated with the source array list. If the source array is NULL, then the function does nothing.
11. ARRAY_LIST_MESSAGE ArrayListClear(SPArrayList *src) Clears all elements from the source array list. After invoking this function, the size of the source list will be reduced to zero and maximum capacity is not affected.
12. ArrayListAddAt(SPArrayList *src, elem *element, int index) Inserts element at a specified index. The elements residing at and after the specified index will be shifted to make place for the new element. If the array list reached its

maximum capacity and error message is returned and the source list is not affected

13. `ArrayListAddFirst(SPArrayList *src, elem *element)` inserts element at the beginning of the source element. The elements will be shifted to make place for the new element. If the array list reaches its maximum capacity an error message is returned and the source list is not affected.
14. `ArrayListAddLast(SPArrayList *src, elem *element)` Inserts element at the end of the source element. If the array list reaches its maximum capacity an error message is returned and the source list is not affected.
15. `ArrayListRemoveAt(SPArrayList *src, int index)` Removes an element from a specified index. The elements residing after the specified index will be shifted to make the list continuous. If the array list is empty then an error message is returned and the source list is not affected.
16. `ArrayListRemoveFirst(SPArrayList *src)` Removes an element from the beginning of the list.
17. `ArrayListRemoveLast(SPArrayList *src)` Returns the element at the end of the list. The function is called with the assertion that all arguments are valid. If any of the arguments is invalid then an undefined value is returned.

ChessGame.h

It's the most important header file of the game. All the logical parts of the chess game and moves are controlled by the functions of the header. The details of the header are given below.

1. `struct ch_game` represents whole game data.
2. `CH_Game* gameCreate(int historySize, int diff, int color, int mode)` Creates a new game with a specified history size. The history size is a parameter which specifies the number of previous moves to store. If the number of moves played so far exceeds this parameter, then first moves stored will be discarded in order for new moves to be stored.
3. `boardInitialize(CH_Game *game)` Initializes a new game board with all the chess figures in their initial places.
4. `argInitialize(CH_Game* game)` Initializes the game default start arguments.
5. `CH_Game* gameCopy(CH_Game* src)` Creates a copy of a given game.
The new copy has the same fields and parameters as the src game..
6. `copyParameters(CH_Game* src, CH_Game* gamecopy)` copies specific game params.
7. `getCurrentPlayer(CH_Game* src)` Returns the current player of the specified game.
8. `gameDestroy(CH_Game* src)` Frees all memory allocation associated with a given game. If `src==NULL` the function does nothing.
9. `CH_GAME_MESSAGE gamePrintBoard(CH_Game* src)` the function prints the board game. If an error occurs, then the function does nothing. The characters

represent different figures. white player(player 1) represented by small letters ,black player(player 2) represented by capital letters.

10. setMove(CH_Game* src, int* move) sets the next move in a given game by specifying x and y move coordinates. The moves are 0-based and in the range [0,GRID -1].

11. doMove(CH_Game* src, elem* element) does the move given-updates the game board, the pieces vector array and kings' positions switches player, updates the history array list.

12. elem* transfer(CH_Game* src ,int* move) Makes an element that represents a move with all relevant fields.

13. int statusToInt(CH_STATUS st) return an enum that represents the game's status by status name and not by int number

14. CH_STATUS intToStatus(int n) returns an enum that represents the game's status by status name and not by int number.

15. playerFig(CH_Game* src, int row, int col) Checks if the figure in the current move is the current player's figure.

16. figureMove(CH_Game* src, elem* element) Checks which piece is making a move and checks if the direction of the move is valid.

17. noOverlap(CH_Game* src, elem* element) checks which piece is making a move and checks if the move doesn't overlap other pieces.

18.isValidMove(CH_Game* src , elem* element) Checks if a move can be done

19.pawnMove(CH_Game* src, elem* element) Checks if the move is a valid move for a pawn.

20. bishopMove(elem* element);Checks if the move is a valid move for a bishop

21.rookMove(elem* element) Checks if the move is a valid move for a rook

22.knightMove(elem* element) Checks if the move is a valid move for a knight

23.queenMove(elem* element) Checks if the move is a valid move for a queen.

24.kingMove(elem* element) Checks if the move is a valid move for a king.

25.isDiagonal(elem* element) Checks if the move is a valid move for a diagonal.

26.isOppPiece(CH_Game* src, int row, int col) Checks if the dst cell has an opponent's piece.

27. isCellEmpty(CH_Game* src, int row, int col) Checks if the given cell represented by row and col is empty.

28.pawnOverlap(CH_Game* src,elem* element) Checks if the given pawn move doesn't overlap pieces.

29.bishopOverlap(CH_Game* src,elem* element);

30.queenOverlap(CH_Game* src,elem* element);

31.kingOrKnightOverlap(CH_Game* src,elem* element);

These functions check overlapping of pieces.

32.CH_GAME_MESSAGE undoPrevMove(CH_Game* src);Undoes the previous move - returns to board the captured figure, if needed, and switches player's turn. Also, updates the kings' positions to the previous one.If the user invoked this command more than historySize times in a row then an error occurs.

33..checkWinner(CH_Game* src) Checks if there's a winner in the specified game status. The function returns either CH_PLAYER_2 or CH_PLAYER_1 in case there's a winner, where the value returned is the symbol of the winner. If the game is over and there's a tie then the value CH_TIE is returned. in any other case the null character is returned.

34. switchPlayer(CH_Game* src) switches the current player

35. pieceUnderAttack(CH_Game* src,int row, int col)Checks if the king of the current player is under attack - can be captured by some opp piece.

36.kingUpdate(CH_Game* src, char fig, int row, int col)Updates the kings' positions.

37. CH_STATUS updateStatus(CH_Game* src) Updates the game's status and returns it.

38.int** possibleKingMoves(CH_Game* src,int row, int col)-Computes all possible moves of a king in the position given. saves all possible moves to an array.

39.int** possibleQueenMoves(CH_Game* src, int row, int col);

40.int** possibleKnightMoves(CH_Game* src, int row, int col);

41.int** possibleRookMoves(CH_Game* src, int row, int col);

42.int** possibleBishopMoves(CH_Game* src, int row, int col);

43.int** possiblepawnMoves(CH_Game* src, int row, int col);

{(39-43)These functions compute all possible moves of a bishop in the position given ,saves all possible moves to an array}

44.addToMoveArray(CH_Game* src,int row, int col,int i, int j,int** arr, int index) If the move is valid - Adds coordinates that represent a move to the relevant move array and returns true.

45.int** arrayInit(int size) -Returns an initialized two dimensional array.

46. isTie(CH_Game* src) Returns true if the game is in tie mode.

47. isCheck(CH_Game* src) Returns true if the game is in check mode.

48..isCheckMate(CH_Game* src) Returns true if the game is in checkmate mode

49.piecesUpdate(CH_Game* src,char figure,bool add,char player)Updates the pieces' vector according to the game board (the last move).

- 50. noPossibleMoves(CH_Game* src)Returns true if there are no valid possible moves.
- 51.int numOfMoves(char ch)Returns maximum number moves..
- 52.char playerPC(CH_Game* src)Returns which player is the PC.
- 53.freeArray(int** arr,int size) Frees all two dimensional array memory resources.
- 54.possibleMoves(CH_Game* src, int row, int col) Returns all possible moves of a specific figure on board.
- 55. CH_GAME_MESSAGE getMovesBonus(CH_Game* src, int row, int col,char** board,bool gui) Prints all figure's possible moves sorted + *-if threatened, ^-if captured opponent figure.
- 56.comp(const void* arg1, const void* arg2) Compares two tuples.
- 57.gameAssign(CH_Game* src,CH_Game* gamecopy)assigns to the gamecopy the src's fields and params.
- 58.failGame(CH_Game *game,bool destroy); destroys the game in case of a malloc fail + prints a suitable message.

ConsoleMode.h

It's the bridge between the logical part of the game and the gui. The game can be played without the gui through the ConsoleMode.c. ConsoleMode connects the searching header to the game which can make the computer move.

- 1. void startGame() initialize the Game.
- 2. settingState(CH_Game* game) invokes the setting state - if the user enters a valid command game_mode, use_color, load, difficulty, quit, default or print etc.until the start command is entered.
- 3.gameState(CH_Game* game, GameCommand command) invokes the game state - the user enters a valid game command: move, get moves, save, undo reset until the quit command is entered.
- 4.SetCommand getNewSetCommand(): returns the set command that the user entered after being parsed.
- 5.GameCommand getNewGameCommand(CH_Game* game, bool to_print): returns the game command that the user entered after being parsed.
- 6. gameMode(CH_Game* game, int arg): changes the game mode according to the user's wishes, prints a suitable message
- 7.gameDifficulty(CH_Game* game, int arg): Change the game difficulty according to the user's wishes, prints a suitable message.
- 8. gameLoad(CH_Game* game, char* path)-Loads a game from the file that the user specified and prints a message if there was an error..
- 9. gameDefault(CH_Game* game): sets the game default params.

10. gamePrintSetting(CH_Game* game): prints the game setting according to the mode.
11. gameQuit(CH_Game* game, bool check): Quits the game and frees all resources.
12. gameMove(CH_Game* game, int* move): sets a game move - moves the players' piece according to the given move, checks whether the move is valid and prints a message accordingly.
13. gameGetMoves(CH_Game* game, int* move): Prints all possible game moves of the given piece, checks whether the invoke of the command is valid and prints accordingly.
14. gameSave(CH_Game* game, char* path): saves the current game to the given path, tries to do the command is valid and prints accordingly.
15. gameUndoMove(CH_Game* game): undo the last 2 moves in the game.
16. gameReset(CH_Game* game): resets the game to the setting mode.
17. fail(CH_Game* game, bool destroy): prints a message if there was an allocation error.
18. computerMove(CH_Game* game): Function that calls the miniMAX and computes the best move for the computer the function makes that move and prints the information.
19. onePlayerMode(CH_Game* game): management of one player mode.
20. twoPlayerMode(CH_Game* game): manages two player modes: get the commands, checks whether there was a winner and executes the command given by the user.
21. char* color(char ch): the function returns the player's color according to the char that it represents .
22. char* figure(char ch): The function returns the figure name according to the char that it represents .
23. statusCheck(CH_Game* game): checks whether the game status changed to tie/check/checkmate and prints a message
24. gameColor(CH_Game* game, int arg): changes the game difficulty according to the user's wishes, prints a suitable message.

Search.h

It is one of the most important headers of the game that have included extraordinary features to the game. It contains the functions that can calculate the possible best move for the computer using the minimax algorithm with alpha-beta pruning.

1. `miniMax(CH_Game* currentGame, unsigned int maxDepth)`: Creates the root of the minimax tree and calls the recursion function due to the desired level which is represented by the MAXDEPTH argument.
2. `miniMaxRec(Node* src, int level, int maxDepth, bool maximize, int alpha, int beta)`: The recursive function that computes the best move according to the minimax.
3. `Node* nodeCreate(CH_Game* src)`: Creates a new node for the minimax tree. The tree contains a copy of the game, which it gets from its parent node in the minimax tree or if it's the root node - a copy of the current game .The node also contains the player and the best move for the current node.
4. `nodeDestroy(Node* src)`: Frees all memory allocation associated with a given node. if `src==NULL` the function does nothing.
5. `nodeCopy(Node* src)`: Creates a copy of the given node.
6. `scoreFunction(CH_Game* src)`: computes the score according to the weight vector defined.
7. `callerWin(CH_Game* src)`: Checks the player who called the function.
8. `value(int alpha, int beta, bool maximize)`: Decides whether the recursive function's depth currently computes the maximum or the minimum.

XMLGameParser.h

1. `parserIsTuple(const char* str, GameCommand *curr, int cell)`: Checks if a specified string represents a valid tuple.
2. `GameCommand parserGameParseLine(char* str)`: Parses a specified line. If the line is a command which has an argument that represents a path then the argument is parsed and is saved in the field path and the field validArg is set to true. In any other case then 'validArg' is set to false and the values 'path' and 'move' are undefined.
3. `GameCommand parserGameParseLine(char* str)`:

A parsed line such that `cmd` - contains the command type, if the line is invalid then this field is set to `INVALID_LINE` `validArg` - is set to true if the command has a relevant extra information: path, number of moves.`path` - the path argument in

case validArg is set to true move - the move argument in case validArg is set to true.

4. GAME_COMMAND_NAME parserIsGameCommand(const char* str) checks which command the user wrote. The function gets a string that represents the command, if the command is unknown returns INVALID_LINE.

XMLREADWRITE.h

1. XML_MESSAGE gameToFile(CH_Game *src, FILE *xml) gets a game and writes to file the game params according to the Conventions.

2. char *boardRowToLine(CH_Game *src, int row) return a string that represent a gameboard row.

3. CH_Game *fileToGame(FILE *f) gets a file and parser each line that represents game's params.

XMLSettingsparser.h

1. parserIsSetCommand(const char *str) Checks which command the user wrote. The function gets a string that represents the command, if the command is unknown returns INVALID_LINE.

2. parserIsInt(const char *str) Checks if a specified string represents a valid integer.

3. parserSetParseLine(char *str) Parses a specified line.

GUI

Button.h

Contains the functions to create buttons.

1. Button *Create_Button(SDL_Rect *location, SDL_Renderer buttonRenderer, const char *enable_image, const char *disable_image, bool isEnabled, bool toShow, bool isClicked): creates a button for GUI mode, if an error occurs prints a message.
2. initButtons(Button **buttonsList, int size): initializes all given buttons to not clicked mode.
3. destroyButton(Button *button): Frees all memory resources associated with the source button. If the source button is NULL, then the function does nothing.
4. drawButton(Button *button): Draws the button according to the button state - is the button enabled or not. If the source button is NULL, then the function does nothing.
5. SDL_Rect *copyLocation(SDL_Rect *src): Copies a given rectangle.
6. failMessage(char *str) function that prints a fail message (ERROR + given string the string that represents the error).

Cell.h

1. `*CreateCell(SDL_Rect *location, SDL_Renderer *renderer, const char *cell_image);` -> creates cell textures.
2. `DestroyCell(Cell *cell);` -> destroys cell
3. `extern void DrawCell(Cell *cell);` -> Draws board cells
4. `textureUpdate(Cell *cell, const char *cell_image);` -> updates board textured during game.

Gamewindow.h

Contains all the functions that are necessary for dealing with game windows.

1. `GameWindowCreate(CH_Game *game, const char **board_images);` Creates a game window, if an error occurred, prints a suitable message.
2. `GameWindowDraw(GameWindow *src);` Function that draws the game window.
3. `GameWindowDestroy(GameWindow *src);` Frees all memory resources associated with the source window.
4. `GameWindowHandleEvent(GameWindow *src, SDL_Event *event, const char **board_images);` Handles the game window according to the game event.
5. `GameWindowHide(GameWindow *src);` Hides the relevant window.
6. `GameWindowShow(GameWindow *src);` Shows the relevant window.
7. `isClickedOnGame(int x, int y, GameWindow *src);` Checks whether there was a click on a button/cell in the window screen.
8. `boardUpdate(Panel *panel, CH_Game *game, const char **board_images);` Updates the board textures that are shown according to the current state.
9. `*cellToImage(char ch, const char **board_images);` Gets a char that represents a game board cell and return the relevant image to be presented on board
10. `getMovesGui(GameWindow *src, int row, int col, const char **board_images);` Updated the board after the player invokes the get moves command (mouse right click).
11. `initDragArgs(GameWindow *src);` Initialized the drag argument to -1 (as default)
12. `pixelToIndex(int x, int y);` Gets a pixel and returns the index of the relevant cell param x - the x coordinate of the pixel .
13. `PCMove(GameWindow *src, const char **board_images);` makes the computer move.
14. `statusMessage(GameWindow *src);` prints game status message,
15. `exitMessage(GameWindow *src, bool flag);` prints exit message(would you like to save the game).
16. `undoUpdate(Panel *panel, CH_Game *game);` Update undo button if needed.
17. `Moving(GameWindow *src, int cell_src, int win, const char **board_images);` makes the computer move.
18. `Drag(GameWindow *src);` updates dragging params.
19. `buttonUp(GameWindow *src, SDL_Event *event, int win, const char **board_images);` Makes the necessary changes following the button up event.
20. `buttonDown(GameWindow *src, SDL_Event *event, int win, const char **board_images);` makes the necessary changes following the button down event.

GuiManager.h

Contains the functions that handle all the events of the game and passes to the respective functions. It's the center of GUI and contains all other headers of GUI.

1. `GuiManager *ManagerCreate()`: Creates a gui manager.
2. `ManagerDestroy(GuiManager *src)`: Frees all memory resources associated with the source gui manager. If the source gui manager is NULL, then the function does nothing.
3. `ManagerDraw(GuiManager *src)`: Draws the active window.
4. `MANAGER_EVENT ManagerHandleEvent(GuiManager *src, SDL_Event *event)`: Handles the different events from the different windows. handles manager due to manager events (show/hide window) prints an error message if an allocation error occurred.
5. `handleManagerDueToGameEvent(GuiManager *src, GAME_EVENT event)`: Handles manager event.
6. `MANAGER_EVENT loadEvent(GuiManager *src)`: returns the event occurs in the Load window.
7. `handleManagerDueToMainEvent(GuiManager *src, MAIN_EVENT event)`: Handles the different events that main has.
8. `drawSetNextScreen(GuiManager *src)`: Draws next settings sub screen according to the flow.
9. `drawSetPrevScreen(GuiManager *src)`: Draws previous settings sub screen according to the flow.
10. `updateImages(GuiManager *src)`: init the game board images gui manager field.
11. `num_of_saved_files(GuiManager *src)`: computes the number of saved games.
12. `XML_MESSAGE saveGameGui(GuiManager *src)`: saves the gui game board to xml file to be used in the future
13. `saveUpdate(GuiManager *src)`: saves the gui game board to xml file to be used in the future.
14. `initSaves(GuiManager *src)`: initializes all save games paths
15. `undoGameGui(GuiManager *src)`: function that undoes the previous two moves on the game board.
16. `gameRestart(GuiManager *src)`: function that restarts the game board.
17. `HandleEventDueToInstrcWindow(GuiManager* src, INTRO_EVENT event)`: Handles event in the Instruction window.

GuiMode.h

Contains gui initialiser function and it can be said that it's the door of GUI.

1. `startGuiMode()`: Starts the GUI MODE
2. `currentState(GuiManager* manager)`: returns the current state of the game, is there any check, or is there any winner or a normal state

InstructionWin.h

Contains the functions that create the instruction window.

1. `InstrcWin* InstrcWindowCreate()` -> creates an instruction window.
2. `InstrcWindowDestroy(InstrcWin* src)`-> destroy instruction window.
3. `InstrcWindowDraw(InstrcWin* src)`->Draws the instruction window.
4. `isClickedOnBack(int x, int y, InstrcWin* src)`->detects whether the Back button was clicked or not.
5. `InstrcWindowHide(InstrcWin* src)`-> hides the instruction window.
6. `InstrcWindowShow(InstrcWin* src)`-> shows instruction window
7. `InstrcWindowHandleEvent(InstrcWin* src, SDL_Event *event)`;->Handles the events in the instruction window

Loadwindow.h

Contains previous game loading functions.

1. `isClickedOnLoad(int x, int y, LoadWin *src)`: function that checks whether there was a click on a button in the load screen.
2. `*LoadWindowCreate(int num_of_saved_games)`:Creates a load window, if an error occurred, prints a suitable message.
3. `LoadWindowDraw(LoadWin *)`: function that draws the load window, prints a suitable message if `src == NULL`.
4. `LoadWindowDestroy(LoadWin *src)`: Frees all memory resources associated with the source window. If the source window is `NULL`, then the function does nothing.
5. `LoadWindowHandleEvent(LoadWin *src, SDL_Eventnt)`: Handles the load window to the load event.
6. `loadButtonInit(LoadWin *win, int num_of_saved_games)`: initializes the buttons on load window.
7. `activateAfterClick(LoadWin *src, int event)`: Updates the buttons according to a click on a valid game slot.
8. `LoadWindowHide(LoadWin *src)`: Hides the relevant window.
9. `LoadWindowShow(LoadWin *src)`: Shows the relevant window.

MainMenu.h

Home window and home event controlling functions belong there.

1. `isClickedOnMain(int x,int y,MainWin* src)`: Function that checks whether there was a click on a button in the main screen.
2. `CreateMainWindow()`: Creates a main window, if an error occurred, prints a suitable message.
3. `DrawMainWindow(MainWin* src)`: Draws the main window.
4. `HandleMainWindowEvent(MainWin* src, SDL_Event* event)`: Handles the main window to the main event.

Panel.h

Contains that creates and manages Panel in the game window.

1. `CreatePanel(SDL_Renderer* renderer,SDL_Surface* image, SDL_Rect* location, bool isSetPanel)`: Create a Panel object.
2. `Cell** CreateGamePanelCells(SDL_Renderer* renderer)`: Create a Game Panel Cells object
3. `Button** CreateSetPanelButtons(SDL_Renderer* renderer)`: Create a Set Panel Buttons object

SettingWindow.h

Whole game settings managing functions belong there.

1. `SettingsWin* SettingsWindowCreate()`: Creates a settings window, if an error occurred, prints a suitable message.
2. `isClickedOnSetting(int x, int y,SettingsWin* src)`: function that checks whether there was a click on a button in the settings screen.
3. `SettingsWindowDraw(SettingsWin* src)`: function that draws the settings window, prints a suitable message if `src == NULL`.
4. `SettingsWindowDestroy(SettingsWin* src)`: Frees all memory resources associated with the source window. If the source window is `NULL`, then the function does nothing.
5. `SettingsWindowHandleEvent(SettingsWin* src, SDL_Event* event,CH_Game* game)`: `SP_SETTINGS_EVENT_INVALID_ARGUMENT` - if one of the arguments is invalid otherwise -the relevant event.
6. `SettingsWindowHide(SettingsWin* src)`: Hides the relevant window.
7. `SettingsWindowShow(SettingsWin* src)`: Shows the relevant window.
8. `gameModeChanges(SettingsWin* src, CH_Game* game,bool one_player)`: Changes the screen to the relevant sub screen - game mode.Shows the relevant buttons.
9. `gameDiffChanges(SettingsWin* src, CH_Game* game,int diff)`: Changes the screen to the relevant sub screen - difficulty.
10. `gameColorChanges(SettingsWin* src, CH_Game* game,int color)`: Changes the screen to the relevant sub screen - color.
11. `SettingsChangeToDifficulty(SettingsWin* src)`: Helper function that switches to the relevant screen - difficulty.
12. `SettingsChangeToGameMode(SettingsWin* src)`: Helper function that switches to the relevant screen - game mode.
13. `SettingsChangeToColor(SettingsWin* src)`: Helper function that switches to the relevant screen - color.
14. `setButtonsInit(SettingsWin* win)`: initializes settings buttons.

Sound.h

Whole games audio system controlling functions belong there. It contains functions that play audio, music and pause them.

1. `initAudio(void)` ->Initialize Audio Variable.
2. `createAudio(const char* file, uint8_t loop, int volume)`->Create an Audio object.
3. `playMusic(const char* file, int volume)`;-> plays music from file
4. `playSound(const char* file, int volume)`; -> plays sound from file

Team Member Responsibilities

Mahmudul Hasan: 1. ENGINE(except XML part)

2. GUI(except texture/image loading)

Ahad bin Islam Shoeb: 1. XML file management.

2. All texture/image loading and creating.

Platform, Library & Tools

Platform: LINUX Kernel based OSs like POP OS, Ubuntu etc.

Library: SDL2, SDL2_image

Language: C

Tools: VS code, Adobe Photoshop, Adobe Spark Post

Limitations

1. Able to search computers' move upto depth 7 very fast. But further depths take too much time to search.
2. Implementing socket though it was proposed by us.

Conclusions

It was my dream to make a chess game. I've made a chess engine before the project and it has given me a lot of confidence to start the project. When I first planned about the gui I was almost in the dark about how to implement a chess engine to build a gui application. The lazy foo site helped me very much to understand the method of gui programming in SDL2. The CS50's youtube video also helped me a lot to understand SDL2. Before learning SDL2, I tried to learn SFML. I coded a little in SFML. It seems a little bit automated. As there are restrictions in using automatic processes in our project, I didn't use it. After completing the whole project, now I can realize the differences between my C programming skill before and now. I think I've improved a lot in C programming and also in real life problem solving in programming. There was some ambiguity in my concept about Pointers, Structures, variable passing to functions etc. Now I can clarify my concepts. I didn't know about GUI programming, how games are made before the project. Now I can do this. I also got proper understanding about file handling in programming and games. After all, it can be said that it was a great project for me. I have learned a lot from that project and gave me confidence to be a good programmer as well as a good game developer.

Future plan

1. To Improve searching
2. To add multiplayer feature.


Repositories

GitHub Repository: <https://github.com/Eulers2020/CHESS-AI>

Youtube Video: <https://youtu.be/hsuMI18Xibo>

References

Learning references:

1. <https://www.chessprogramming.org>
2. [Programming A Chess Engine in C](#)
3.  Writing 2D Games in C using SDL by Thomas Lively
4. <https://www.linkedin.com/pulse/writing-simple-chess-engine-c-baran-can-%C3%B6ner-cfa/?articleId=6686567938985615360>
5. [Beginning Game Programming v2.0](#)
6. [Minimax Algorithm with Alpha-beta pruning](#)
7. And many more sites on google.

Materials and resources references:

1. One Love by Keys of Moon | <https://soundcloud.com/keysofmoon>
Attribution 4.0 International (CC BY 4.0) <https://creativecommons.org/licenses/by/4.0/>
Music promoted by <https://www.chosic.com/>
2. <https://mixkit.co/> for click sound