Chess Game with AI

Jack Hawkins, Candidate Number 5793

Hills Road Sixth Form College, Centre Number 22147

**Table of Contents**

[1 Analysis 2](#_Toc129430212)

[**2** Documented Design 8](#_Toc129430213)

[**3** Technical Solution 45](#_Toc129430214)

[**4** Testing 96](#_Toc129430215)

[**5** Evaluation 101](#_Toc129430216)

[**6** Screenshots and Videos 104](#_Toc129430217)

[**7** References 110](#_Toc129430218)

### Analysis

**Table of Contents**

[**1.1** **The project** 2](#_Toc129022087)

[**1.2** **Background to Chess** 2](#_Toc129022089)

[**1.3** **End user requirements** 3](#_Toc129022119)

[**1.4** **Research** 4](#_Toc129022132)

[**1.5** **Objectives** 6](#_Toc129022175)

[**1.6** **Modelling** 7](#_Toc129022197)

* 1. **The project**
     1. The end user and an introduction to the project

The project is to create a Chess game using C# in Windows Forms which allows the user to play against an AI.

The end user is Anders Flack, an avid chess player, who often plays online with friends. Anders plays on chess.com, a popular chess website, so therefore my research on existing chess products has largely been based on chess.com. As Anders regularly plays chess, he knows which features are fun or useful, and which features he would like to see in a chess game.

The features that Anders wanted to see in my project were identified in a conversation which is stated in 1.\_. He hopes to see some of his favourite features from chess.com, such as playing against other users against a clock, and other features which aren’t present on chess.com, such as being able to save a game and resume at a later date.

* 1. **Background to Chess**
     1. Square

        Description automatically generatedOverview

Chess is a two-player board game in which the aim of the game is to ‘checkmate’ the other player’s king. This is achieved by putting their king in ‘check’. This means that the opposing king is in a position where it could be taken in the next turn. For a king to be in checkmate, it must be in check, and it must be impossible for the player to move their king out of check, or to block or take the piece that has put the king into check.

The user will have the choice between playing against another user, or against an AI. The AI will search for the best possible move in order to beat the user.

* + 1. Rules

Each player starts with two rooks, two knights, two bishops, a king, a queen, and eight pawns, laid out as seen here, with the pawns starting on the rows in front of the other pieces.

Moves:

* Pawns can move one space vertically – up for white, down for black – but can move two spaces on their first move.
* Rooks can move up to seven spaces either vertically or horizontally.
* Bishops can move up to seven spaces diagonally.
* Knights can move in an L-shape, moving two spaces either vertically or horizontally, and one space the other way.
* Kings can move one space in any direction.
* Queens can move up to seven spaces in any direction.
* When taking another piece, pawns move one space diagonally.
* All other pieces take a piece the same way they would move.

If a pawn reaches the other end of the board, it becomes a queen.

Check and checkmate:

* The aim of the game is to put the opponent’s king into checkmate. To do this, the player must put their opponent’s king in check.
* Check is when the king is under threat from an opponent’s piece. For example, if a rook is in a position where it would be able to take the opponent’s king, the king is in check.
* For the king to be in checkmate, it must already be in check, and all of the following conditions must be true:
  + The king cannot move out of check.
  + The player cannot take the opponent’s piece which has put the king in check, without the king being in check from another opposing piece.
  + The player cannot block the opponent’s piece which has put the king in check, without the king being in check from another opposing piece.

Stalemate:

* Stalemate is when neither player can put their opponent’s king in checkmate with the current state of the board.
* This may be that there are insufficient or inadequate pieces on the board to checkmate the opponent’s king, or the king may be in a position where it is not in check, but all possible moves would result in it being in check.
* It is also stalemate if there are no possible moves for the player to make.
* The other requisites for a stalemate are dependent on what combinations of pieces are left on both sides of the board:
  + King and knight:
    - As the knight can only move in L shapes, the king can always move out of check when the opponent only has a knight.
  + King and two knights:
    - Even with two knights, it is impossible to checkmate the opponent’s king.
  + King and bishop:
    - As bishops can only move diagonally, they are limited to only being able to move to either white or black spaces, depending on their starting position. Therefore, it is impossible to checkmate the opponent’s king with just a bishop, as the king could just move between spaces of the opposite colour. [[1]](#footnote-1)

Playing against the clock:

* When playing against the clock, each player is given the same starting time limit. This is normally five minutes, but in this project, there are also the options of one minute or ten minutes.
* When the game starts, it is white’s turn, so white’s clock will start ticking down, but black’s clock does not. This continues until white completes its first move. Their clock then pauses, and the black clock will begin to tick.
* If a player’s clock reaches zero before the game reaches a checkmate or stalemate, that player will be able to finish their current move, and if after that move there is still no checkmate or stalemate, the player whose clock has reached zero loses.
  1. **End user requirements**
     1. Communication with end user

I communicated with my end user, Anders, during a conversation about my project. The results of this conversation influenced my research and my objectives.

1. What app or website do you usually play chess on?

**Anders** – I normally play on chess.com. They have an app and a website so I can play on both my phone and my laptop, and I can login on both devices and use the same account.

1. What do you like about chess.com?

**Anders** – Chess.com gives me the choice between playing against another person online or against the computer. I prefer to play online because then I can play against the clock, so I play five-minute games which mean I can play lots of short, fast-paced games.

1. Is there anything you don’t like about chess.com?

**Anders** – There are different levels of computer difficulty, but they all play really well, including the level one bot. The bots go up to around level 3,000, and I struggle to beat them. Other than that, it is almost perfect.

1. Are there any particular features on chess.com that you would like to be implemented in my project?

**Anders** – The choice between being able to play against either the computer or another player; the ability to play against the clock when playing against another player; an account and leader board system, where I can see my stats and compare them to other players.

1. Are there any features which aren’t on chess.com which you would like implemented?

**Anders** – The ability to save a game and be able to continue playing at a later date would be nice, as it means longer games can be continued later.

* + 1. Key features identified during communication

1. Player versus player games.
2. Player versus computer games.
3. Time limited games when playing against another player.
4. Accounts and leader boards, with stats.
5. Save games and load them at a later date.

In an unrecorded later conversation, Anders said that the interface should look nice and should be clear and easy to use.

* 1. **Research**
     1. Researching the rules

The main rules of chess, such as the moves, check, checkmate, and stalemate are widely known, and therefore required little research. However, I did have to research the combinations of pieces that result in a stalemate, mentioned in 1.2.2.

* + 1. Researching existing chess programs

Following my conversation with Anders, I researched chess.com, the website he plays chess on. It has the following features:

* Play against a computer:
  + There are multiple difficulty levels, ranging from 1, to the highest of 2,850.
  + Against the computer there is the option to allow the player to undo a poor move or a mistake.
* Play against another player:
  + Users can play against any player across the world.
  + There is a ranking system, which enables users to play against someone of a similar skill level to them.
  + There is a friend system, which allows users to add friends and play against them online.
* Solve chess problems:
  + There are small puzzles in which the user has to figure out which moves to make in order to checkmate the opposing king.
* Chess lessons:
  + There are interactive lessons with videos for new players to learn the basics, or for more experienced players to improve.
    1. Researching algorithms

In order to implement the AI, I needed to research how a chess AI works and what algorithms are needed.[[2]](#footnote-2)

A chess AI works by generating all the possible moves that the AI could make and searching for the best one. This is done by evaluating the state of the board after each of these moves, and also following moves, so that the move with the best possible outcome in a few moves time is selected.

The program will generate the possible moves for each piece for either black or white individually, and for each of these moves, the ‘score’ will be evaluated by adding up the strength values of each piece on the board. If the score is positive, white is in a stronger position, if the score is negative, black is in a stronger position. This is done for every single possible move, using a minimax search tree. This filters through each move and finds the best one. This search is optimised using alpha-beta pruning, which removes the need to search nodes of the tree which would never be played.

Move generation:

* The computer will generate all possible moves for each piece.
* The algorithm will check whether the piece can legally move to a position, checking whether that position is within the move set of that piece, and if that position is occupied, whether the piece that occupies that position is an opposing piece. The algorithm also checks if a move would result in the king being in check and removes moves which would have this result.
* For rooks, bishops, and queens, which can move up to seven spaces, if one of the spaces is blocked, the piece cannot move to a position on the other side of the blocking piece.
  + For example, if there is a pawn in position (2, 2), and a rook in (2, 5), the rook could move to (2, 3) and (2, 4), but not (2, 1).
  + If the rook is a different colour to the pawn, the rook could move to (2, 2) and take the pawn.
* The algorithm will be used to determine possible moves for both the user’s turn and the computer’s turn.

Positional evaluation:

* Positional evaluation will only be used on the computer’s turn.
* Each piece is given a relative strength as an integer number. The more significant the number, the stronger the piece.
* White pieces have a positive strength, and black pieces have a negative strength.
* The strengths for each piece will be as follows:
  + Pawn: ±10.
  + Bishop: ±30.
  + Knight: ±30.
  + Rook: ±50.
  + Queen: ±90.
  + King: ±900.
* The evaluation algorithm will add up all of these relative strengths and calculate who is in the stronger position.
* If the result is zero, neither side is in a stronger position. If the result is positive, white is in a stronger position, and if the result is negative, black is in a stronger position.

Minimax search tree:

* The search tree filters through the generated moves and determines the outcomes of each possible combination of moves for a predetermined number of turns.[[3]](#footnote-3)
* The name minimax comes from the idea that one player will try to minimise the outcome, while the other will try to maximise it.
* In the case of this chess project, the white player is trying to maximise the outcome, as they will have a positive score if they are in a stronger position than black. Therefore, black is trying to minimise the outcome.
* As the AI will always play as black, the algorithm will select the move with the minimum outcome.
* As the algorithm searches depth first, it will search to a depth of three, the algorithm will call itself recursively, and will return the result to the previous call of the algorithm to determine which moves are the best options.
* This search can be optimised by using alpha-beta pruning.

Alpha-beta pruning:

* Alpha-beta pruning optimises the search by pruning nodes from the tree which have a worse result than ones previously searched, as those moves will never be played by the computer.
  + For example, if a move has a score of -20, and there is another move which has a score of +20, the computer will look to minimise the outcome, and therefore choose the move which has a result of -20.
  + As the move with a score of +20 will never be played, there is no need to search the possible moves made after that move.
* This significantly reduces the number of searches required, especially for larger search depths, and therefore reduces the time taken to search the tree.
  1. **Objectives**

The objectives are colour-coded to represent the importance of the objective. Red is crucial, amber is important, green is less important.

* + 1. **To create a chess game that allows a user to play a game of chess against an AI.**
       1. The algorithms for the AI will contain the following:
          1. **Move Generation** – all possible moves for each AI-controlled piece will be generated. These moves must be legal. This will be determined by whether the move is within the move set of the piece, whether the king of the same colour will be in check, and whether an occupying piece in a position is of the same colour.
          2. **Positional Evaluation** – the relative strength of each piece will be added to calculate the state of the board after each move generated. If the result is zero, neither side is in a stronger position than the other. If the result is positive, white is in a stronger position, and if the result is negative, black is in a stronger position.
          3. **Minimax Search Tree** – this will search through all of the possible moves to find the one with the best outcome for the AI, and this will be the selected move.
          4. **Alpha-Beta Pruning** – an optimisation to the minimax tree search. This removes moves which would never be played and removes the need to search moves after that removed move.
    2. **To implement algorithms which determine the state of the game, i.e., check, checkmate, and stalemate.**
       1. **Check** – the move generation algorithm will be used to search all of the possible moves that the opponent could make. The check algorithm will return true if the any of the opposing pieces are able to move to the position which the king occupies.
       2. **Checkmate** – an extension of the check algorithm, the checkmate algorithm will search through the possible moves for the king and determine whether the king can move out of check. If it cannot, it will determine whether the piece which has put the king in check can be taken or blocked, and hence remove the king from check. If none of these options can be done, the checkmate algorithm will return true.
       3. **Stalemate** – the stalemate algorithm will check for the combinations of pieces that result in a checkmate, as previously mentioned in 1.2.2. It will then check the other stalemate conditions – if there are no possible moves, or if the king is not in check, but moving a piece would result in a check.
    3. **To allow the user to save their progress and come back to the game at a later date, through the use of file handling.**
       1. **Saving a game** – when saving the game, the program will write to a .txt file called ‘savedGame’. The program will write the details of each piece on the board; whether the game is being played with one or two players; and if the game is being played against the clock, the timers for both players.
       2. **Loading a game** – when loading the game, the program will read in the same order it wrote to the file. Using the information read, it will then create each piece; set the game to one or two players; and if the game was being played against the clock, the timers of each player.
    4. **To allow users to create a profile, which will have a unique name, so that they can save their stats and compare them to other users.**
       1. The user will be able to save their stats and profile to a unique .txt file named after their username.
       2. This file will store the user’s username, number of games played, and number of wins, draws, and losses.
       3. Using the stats of all users, a leader board will be created, ranking users by their wins and draws.
    5. **To allow users to play against the clock when playing in two player.**
       1. The user will have the option to set a timer for one, five, or ten minutes, which will be set for both players. There will also be the option to play without a timer.
       2. The time will start ticking at the beginning of a player’s turn and stop when their turn ends. The timer will continue on their next turn.
       3. During each player’s turn, their clock will tick down using the stopwatch function. If a player’s time reaches zero, that player will be able to finish their turn. If their turn results in a stalemate, the game will end as a draw. If their turn results in a checkmate, they will win. If neither of these happen, they will lose.
  1. **Modelling**
     1. Class Diagram:

Diagram

Description automatically generated

* + 1. Flowchart:

Diagram, schematic

Description automatically generated

### Documented Design

**Table of Contents**

[**2.1** **Classes** 8](#_Toc129029114)

[**2.2** **Algorithms** 13](#_Toc129029115)

[**2.3** **Data Structures** 42](#_Toc129029116)

[**2.4** **User Interface** 42](#_Toc129029117)

[**2.5** **File Structure and Organisation** 45](#_Toc129029118)

* 1. **Classes**
     1. **Class Diagram**

Since the modelling stage in the analysis, the class diagram changed, with the implementation of the AI class. I also included the Form1 class in the diagram.

Diagram

Description automatically generated

* + 1. **Piece**

Each type of piece has its own class, which inherits from the parent class, Piece. All of the pieces inherit all of the attributes and have no extra attributes specific to that class. All of the pieces inherit both methods from Piece, and override PossibleMoves(). The Pawn class has one extra method which is specific to that class, and the King class has two extra methods.

Attributes:

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Name** | **Initialised to** | **Information** |
| string | pieceType | “” | This is updated to the name of the type of piece when the individual pieces are created. |
| string | colour | “” | This is updated to either “black” or “white” when the piece is created. |
| int | xGridPos | 0 | The x-ordinate for the piece. This is updated when the piece is put onto the board. |
| int | yGridPos | 0 | The y-ordinate for the piece. This is updated when the piece is put onto the board. |
| int | newX | 0 | This is used when changing the position of the piece. |
| int | newY | 0 | This is used when changing the position of the piece. |
| int | strength | 0 | This is updated depending on the piece type. |
| bool[,] | possibleMoves | false | The locations are updated to true if the piece can move to that position on the board. |

Methods:

|  |  |  |
| --- | --- | --- |
| **Return type** | **Name** | **Description** |
| bool[,] | PossibleMoves(bool[,] possibleMoves, Piece[,] board) | Finds all the possible moves for a piece. The default method in the Piece class sets all values to false. |
| bool | DifferentColour(int otherX, int otherY, Piece[,] board) | Returns true if the piece is a different colour to the one stored in a specified position. |
| void | AddPiece(Piece piece, Piece[,] board) | Adds a piece to the board. |
| void | RemovePiece(Piece piece, Piece[,] board) | Removes a piece from the board. |
| void | UpdatePosition(Piece piece, int currentX, int currentY, int newX, int newY, Piece[,] board) | Updates the position of a piece. If the new position is occupied, the occupying piece is removed. |

* + - 1. **Pawn**

Attributes:

The Pawn class inherits all of the attributes from the Piece class and has no more attributes.

Methods:

|  |  |  |
| --- | --- | --- |
| **Return type** | **Name** | **Description** |
| bool[,] | PossibleMoves(bool[,] possibleMoves, Piece[,] board) | Overridden from the Piece class. Sets values to true if the pawn can move to that position. |
| void | QueenConvert(Piece[,] board) | Converts the pawn into a queen if the pawn reaches the other end of the board. |

* + - 1. **Bishop**

Attributes:

The Bishop class inherits all of the attributes from the Piece class and has no more attributes.

Methods:

|  |  |  |
| --- | --- | --- |
| **Return type** | **Name** | **Description** |
| bool[,] | PossibleMoves(bool[,] possibleMoves, Piece[,] board) | Overridden from the Piece class. Sets values to true if the bishop can move to that position. |

* + - 1. **Rook**

Attributes:

The Rook class inherits all of the attributes from the Piece class and has no more attributes.

Methods:

|  |  |  |
| --- | --- | --- |
| **Return type** | **Name** | **Description** |
| bool[,] | PossibleMoves(bool[,] possibleMoves, Piece[,] board) | Overridden from the Piece class. Sets values to true if the rook can move to that position. |

* + - 1. **Knight**

Attributes:

The Knight class inherits all of the attributes from the Piece class and has no more attributes.

Methods:

|  |  |  |
| --- | --- | --- |
| **Return type** | **Name** | **Description** |
| bool[,] | PossibleMoves(bool[,] possibleMoves, Piece[,] board) | Overridden from the Piece class. Sets values to true if the knight can move to that position. |

* + - 1. **King**

Attributes:

The King class inherits all of the attributes from the Piece class and has no more attributes.

Methods:

|  |  |  |
| --- | --- | --- |
| **Return type** | **Name** | **Description** |
| bool[,] | PossibleMoves(bool[,] possibleMoves, Piece[,] board) | Overridden from the Piece class. Sets values to true if the pawn can move to that position. |
| bool | Check(int x, int y, Piece[,] board) | Returns true if the king is in check. |
| bool | Checkmate(int x, int y, Piece[,] board) | Returns true if the king is in checkmate. |

* + - 1. **Queen**

Attributes:

The Queen class inherits all of the attributes from the Piece class and has no more attributes.

Methods:

|  |  |  |
| --- | --- | --- |
| **Return type** | **Name** | **Description** |
| bool[,] | PossibleMoves(bool[,] possibleMoves, Piece[,] board) | Overridden from the Piece class. Sets values to true if the pawn can move to that position. |

* + 1. **AI**

Attributes:

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Name** | **Initialised to** | **Information** |
| int | searchDepth | 3 | Sets the default search depth for the minimax search. |
| List<int> | moveScores | List.Empty | Stores the scores of each move to determine the best move. |
| List<Piece> | movedPieces | List.Empty | Stores the pieces which move in the first layer of the search. |
| int | resultPos | 0 | Stores the position of the result in movedPieces. |

Methods:

|  |  |  |
| --- | --- | --- |
| **Return type** | **Name** | **Description** |
| Piece[,] | DuplicateBoard(Piece[,] newBoard, Piece[,] oldBoard) | Duplicates a board to use in move generating or checking. |
| List<Piece[,]> | GenerateMoves(Piece [,] board, int depth, bool whiteTurn) | Generates all possible moves for every piece of a specified colour. |
| int | Evaluation(Piece[,] board, bool checkForWhite) | Calculates the balance of play. |
| int | Minimax(Piece[,] board, int depth, bool maximisingPlayer, bool whiteTurn, int alpha, int beta) | Searches through the possible moves to find the best one. |
| void | Turn() | Combines the above algorithms to create allow the AI to make a turn. |

* + 1. **Form1**

Attributes:

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Name** | **Initialised to** | **Information** |
| string | fileSave | “savedGame.txt” | The name of the file which will store saved games. |
| bool | whiteWin | false | Stores whether white has won. |
| bool | timeEnabled | false | Stores whether the users are playing against the clock. |
| bool | gamePaused | false | Stores whether the game is paused. |
| bool | gameOver | false | Stores whether the game is finished. |
| bool | playerVComputer | false | Stores whether the game is one or two player. |
| bool | stalemate | false | Stores whether the game has resulted in stalemate. |
| bool | checkmate | false | Stores whether the game has resulted in checkmate. |
| bool | moveMade | false | Stores whether a move has been made by the player. |
| bool | pieceMoved | false | Stores whether the player has moved a piece. |
| int | whiteTimer | 0 | Stores the time the white player has left. |
| int | blackTimer | 0 | Stores the time the black player has left. |
| int | timeLimit | 0 | Stores the time limit for each player. |
| Stopwatch | whiteStopwatch |  | Stopwatch to measure white’s time. |
| Stopwatch | blackStopwatch |  | Stopwatch to measure black’s time. |
| Piece | currentPiece | null | The piece selected by the user. |

Methods:

|  |  |  |
| --- | --- | --- |
| **Return type** | **Name** | **Description** |
| void | CreatePawn(int i, int j) | Creates a pawn. |
| void | CreateKnight(int i, int j) | Creates a knight. |
| void | CreateRook(int i, int j) | Creates a rook. |
| void | CreateBishop(int i, int j) | Creates a bishop. |
| void | CreateKing(int i) | Creates a king. |
| void | CreateQueen(int i) | Creates a queen. |
| void | CreatePieces() | Creates all the pieces by calling each of the above methods. |
| void | CreatePieceImages(Piece piece, PictureBox newPictureBox, string colour) | Creates the images for each piece. |
| void | AddLabel(int i) | Creates a label, one to eight. |
| void | AddPanel(int i, int j) | Creates the board. |
| void | Form1\_Load(object sender, EventArgs e) | Loads the form. |
| void | NewGameButton\_Click(object sender, EventArgs e) | Calls CreatePieces() and sets some settings to prepare a new game. |
| void | SaveGameButton\_Click(object sender, EventArgs e) | Writes to savedGame.txt the state of the game. |
| void | LoadGameButton\_Click(object sender, EventArgs e) | Reads from savedGame.txt the state of the game. |
| void | GameOver() | Ends the game if it is in a state of checkmate or stalemate. |
| void | SetTimerButton\_Click(object sender, EventArgs e) | Sets the timers for the players. |
| void | StartGameButton\_Click(object sender, EventArgs e) | Starts or resumes the game. |
| void | PauseGameButton\_Click(object sender, EventArgs e) | Pauses the game. |
| void | UpdateTimer() | Updates the timer of a player. |
| void | Turn() | Calls PlayerTurn() on a player’s turn and calls Turn() from the AI class on the computer’s turn. |
| void | EndOfTurn() | Ends a turn and checks the state of the game. |
| void | PlayerTurn() | Runs until the player makes a move. |
| void | PickOrDropPiece(MouseEventArgs e) | The algorithm used to allow the player to move a piece. |
| void | NewPictureBox\_MouseDown(object sender, MouseEventArgs e) | Activates the PickOrDropPiece() method when the user clicks a piece. |
| void | SetModeButton\_Click(object sender, EventArgs e) | Sets the mode to either one or two player. |
| bool | Stalemate(Piece[,] board) | Returns true if the game is in stalemate. |
| bool | StalemateCombinations(string colour, Piece[,] board) | Checks for specific piece combinations and returns true if present. |

* + 1. **Globals**

The Globals class stores all the global variables and methods used in the program.

Attributes:

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Name** | **Initialised to** | **Information** |
| Piece[,] | grid | null | Stores the location of pieces. |
| Piece | takenPiece | null | Stores a removed piece. |
| PictureBox | takenPieceImage | null | Stores the image of a removed piece. |
| PictureBox[,] | pieceImages | null | Stores the images of pieces. |
| bool | whiteTurn | true | Stores whose turn it is. |
| int | kingX | 0 | Stores the location of the fetched king. |
| int | kingY | 0 | Stores the location of the fetched king. |

Methods:

|  |  |  |
| --- | --- | --- |
| **Return type** | **Name** | **Description** |
| bool | IsPositionEmpty(int x, int y, Piece[,[ board) | Returns true if the position specified is empty. |
| void | GetKing(string colour, Piece[,] board) | Fetches the king of a specified colour and updates the kingX and kingY variables. |
| int | GetPieceCount(Piece[,] board) | Returns the number of pieces on the board. |

* 1. **Algorithms**
     1. **Basic algorithms**
        1. **PossibleMoves**

PossibleMoves is an algorithm in the piece class which is inherited and overridden in each of the six subclasses. The algorithm returns an eight-by-eight 2D array of Booleans which represent the positions on the board. If a position is true, the piece to which the array belongs can legally move to that position. If the position is false, the piece cannot move to that position.

|  |  |
| --- | --- |
| **Pseudocode**  SUBROUTINE PossibleMoves(possibleMoves, board)  FOR i <- 0, i < 8, i++  FOR j <- 0, j < 8, j++  possibleMoves[i, j] <- false  ENDFOR  ENDFOR  RETURN possibleMoves  ENDSUBROUTINE | |
| **Input**  possibleMoves – the array which will be returned with the results of the subroutine.  board – the board which is being used at that stage in the program. | **Output**  possibleMoves – the array has been filled with true for the positions which the piece can move to. |

**/**

* + - * 1. **PossibleMoves – Pawn**

The pseudocode for this Pawn algorithm will only include the white side of the algorithm to save writing out the same algorithm twice with different numbers and signs.

|  |  |
| --- | --- |
| **Pseudocode**  SUBROUTINE PossibleMoves(possibleMoves, board)  King king <- NEW King()  IF !IsPositionEmpty(kingX, kingY, board)  IF board[kingX, kingY].pieceType != “king”  GetKing(colour, board)  king.xGridPos = kingX  king.yGridPos = kingY  ELSE  king <- board[kingX, kingY]  ENDIF  ENDIF  IF colour = “white”  IF yGridPos > 0  IF IsPositionEmpty(xGridPos, yGridPos – 1, board)  Piece[,] duplicateBoard <- NEW Piece[8, 8]  duplicateBoard <- DuplicateBoard(duplicateBoard, board)  UpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos, yGridPos – 1, duplicateBoard)  IF !king.Check(king.xGridPos, king.yGridPos, duplicateBoard)  possibleMoves[xGridPos, yGridPos – 1] <- true  ENDIF  IF yGridPos = 6  UpdatePosition(duplicateBoard[xGridPos, yGridPos - 1], xGridPos, yGridPos, xGridPos, yGridPos – 2, duplicateBoard)  IF IsPositionEmpty(xGridPos, yGridPos – 2, board && !king.Check(king.xGridPos, king.yGridPos, duplicateBoard))  possibleMoves[xGridPos, yGridPos - 2] <- true  ENDIF  ENDIF  IF xGridPos > 0 && yGridPos > 0  IF !IsPositionEmpty(xGridPos – 1, yGridPos – 1, board)  IF DifferentColour(xGridPos – 1, yGridPos – 1, board)  Piece[,] duplicateBoard <- NEW Piece[8, 8]  duplicateBoard <- DuplicateBoard(duplicateBoard, board)  UpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos - 1, yGridPos – 1, duplicateBoard)  IF !king.Check(king.xGridPos, king.yGridPos, duplicateBoard)  possibleMoves[xGridPos – 1, yGridPos - 1] <- true  ENDIF  ENDIF  ENDIF  IF xGridPos < 7 && yGridPos > 0  IF !IsPositionEmpty(xGridPos + 1, yGridPos – 1, board)  IF DifferentColour(xGridPos – 1, yGridPos – 1, board)  Piece[,] duplicateBoard <- NEW Piece[8, 8]  duplicateBoard <- DuplicateBoard(duplicateBoard, board)  UpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos - 1, yGridPos + 1, duplicateBoard)  IF !king.Check(king.xGridPos, king.yGridPos, duplicateBoard)  possibleMoves[xGridPos + 1, yGridPos - 1] <- true  ENDIF  ENDIF  ENDIF  ENDIF  ELSE IF colour = “black”  …  RETURN possibleMoves  ENDSUBROUTINE | |
| **Input**  possibleMoves – the array which will be returned with the results of the subroutine.  board – the board which is being used at that stage in the program. | **Output**  possibleMoves – the array has been filled with true for the positions which the pawn can move to. |

* + - * 1. **PossibleMoves – Bishop**

|  |  |
| --- | --- |
| **Pseudocode**  SUBROUTINE PossibleMoves(possibleMoves, board)  King king <- NEW King()  IF !IsPositionEmpty(kingX, kingY, board)  IF board[kingX, kingY].pieceType != “king”  GetKing(colour, board)  king.xGridPos = kingX  king.yGridPos = kingY  ELSE  king <- board[kingX, kingY]  ENDIF  ENDIF  FOR i <- 1, i < 8, i++  IF xGridPos + i <= 7 && yGridPos + i <= 7  IF IsPositionEmpty(xGridPos + i, yGridPos + i, board) || (!IsPositionEmpty(xGridPos + i, yGridPos + i, board) && DifferentColour(xGridPos + i, yGridPos + i, board))  Piece[,] duplicateBoard <- NEW Piece[8, 8]  duplicateBoard <- DuplicateBoard(duplicateBoard, board)  UpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos + i, yGridPos + i, duplicateBoard)  IF !king.Check(king.xGridPos, king.yGridPos, duplicateBoard)  possibleMoves[xGridPos + i, yGridPos + i] <- true  ENDIF  ENDIF  IF !possibleMoves[xGridPos + i, yGridPos + i]  BREAK  ENDIF  ENDIF  IF xGridPos + i <= 7 && yGridPos - i >= 0  IF IsPositionEmpty(xGridPos + i, yGridPos - i, board) || (!IsPositionEmpty(xGridPos + i, yGridPos - i, board) && DifferentColour(xGridPos + i, yGridPos - i, board))  Piece[,] duplicateBoard <- NEW Piece[8, 8]  duplicateBoard <- DuplicateBoard(duplicateBoard, board)  UpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos + i, yGridPos - i, duplicateBoard)  IF !king.Check(king.xGridPos, king.yGridPos, duplicateBoard)  possibleMoves[xGridPos + i, yGridPos - i] <- true  ENDIF  ENDIF  IF !possibleMoves[xGridPos + i, yGridPos - i]  BREAK  ENDIF  ENDIF  IF xGridPos - i >= 0 && yGridPos - i >= 0  IF IsPositionEmpty(xGridPos - i, yGridPos - i, board) || (!IsPositionEmpty(xGridPos - i, yGridPos - i, board) && DifferentColour(xGridPos - i, yGridPos - i, board))  Piece[,] duplicateBoard <- NEW Piece[8, 8]  duplicateBoard <- DuplicateBoard(duplicateBoard, board)  UpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos - i, yGridPos - i, duplicateBoard)  IF !king.Check(king.xGridPos, king.yGridPos, duplicateBoard)  possibleMoves[xGridPos - i, yGridPos - i] <- true  ENDIF  ENDIF  IF !possibleMoves[xGridPos - i, yGridPos - i]  BREAK  ENDIF  ENDIF  IF xGridPos - i >= 0 && yGridPos + i <= 7  IF IsPositionEmpty(xGridPos - i, yGridPos + i, board) || (!IsPositionEmpty(xGridPos - i, yGridPos + i, board) && DifferentColour(xGridPos - i, yGridPos + i, board))  Piece[,] duplicateBoard <- NEW Piece[8, 8]  duplicateBoard <- DuplicateBoard(duplicateBoard, board)  UpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos - i, yGridPos + i, duplicateBoard)  IF !king.Check(king.xGridPos, king.yGridPos, duplicateBoard)  possibleMoves[xGridPos - i, yGridPos + i] <- true  ENDIF  ENDIF  IF !possibleMoves[xGridPos - i, yGridPos + i]  BREAK  ENDIF  ENDIF  ENDFOR  RETURN possibleMoves  ENDSUBROUTINE | |
| **Input**  possibleMoves – the array which will be returned with the results of the subroutine.  board – the board which is being used at that stage in the program. | **Output**  possibleMoves – the array has been filled with true for the positions which the bishop can move to. |

* + - * 1. **PossibleMoves – Rook**

|  |  |
| --- | --- |
| **Pseudocode**  SUBROUTINE PossibleMoves(possibleMoves, board)  King king <- NEW King()  IF !IsPositionEmpty(kingX, kingY, board)  IF board[kingX, kingY].pieceType != “king”  GetKing(colour, board)  king.xGridPos = kingX  king.yGridPos = kingY  ELSE  king <- board[kingX, kingY]  ENDIF  ENDIF  FOR i <- 1, i < 8, i++  IF xGridPos – i >= 0  IF IsPositionEmpty(xGridPos – i, yGridPos, board) || (!IsPositionEmpty(xGridPos – i, yGridPos, board) && DifferentColour(xGridPos – i, yGridPos, board))  Piece[,] duplicateBoard <- NEW Piece[8, 8]  duplicateBoard <- DuplicateBoard(duplicateBoard, board)  UpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos - i, yGridPos, duplicateBoard)  IF !king.Check(king.xGridPos, king.yGridPos, duplicateBoard)  possibleMoves[xGridPos – i, yGridPos] <- true  ENDIF  ENDIF  IF !possibleMoves[xGridPos – i, yGridPos]  BREAK  ENDIF  ENDIF  IF xGridPos + i <= 7  IF IsPositionEmpty(xGridPos + i, yGridPos, board) || (!IsPositionEmpty(xGridPos + i, yGridPos, board) && DifferentColour(xGridPos + i, yGridPos, board))  Piece[,] duplicateBoard <- NEW Piece[8, 8]  duplicateBoard <- DuplicateBoard(duplicateBoard, board)  UpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos + i, yGridPos, duplicateBoard)  IF !king.Check(king.xGridPos, king.yGridPos, duplicateBoard)  possibleMoves[xGridPos + i, yGridPos] <- true  ENDIF  ENDIF  IF !possibleMoves[xGridPos + i, yGridPos]  BREAK  ENDIF  ENDIF  ENDFOR  FOR i <- 1, i < 8, i++  IF yGridPos – i >= 0  IF IsPositionEmpty(xGridPos, yGridPos - i, board) || (!IsPositionEmpty(xGridPos, yGridPos - i, board) && DifferentColour(xGridPos, yGridPos - i, board))  Piece[,] duplicateBoard <- NEW Piece[8, 8]  duplicateBoard <- DuplicateBoard(duplicateBoard, board)  UpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos, yGridPos - i, duplicateBoard)  IF !king.Check(king.xGridPos, king.yGridPos, duplicateBoard)  possibleMoves[xGridPos, yGridPos - i] <- true  ENDIF  ENDIF  IF !possibleMoves[xGridPos, yGridPos - i]  BREAK  ENDIF  ENDIF  IF yGridPos + i <= 7  IF IsPositionEmpty(xGridPos, yGridPos + i, board) || (!IsPositionEmpty(xGridPos, yGridPos + i, board) && DifferentColour(xGridPos, yGridPos + i, board))  Piece[,] duplicateBoard <- NEW Piece[8, 8]  duplicateBoard <- DuplicateBoard(duplicateBoard, board)  UpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos, yGridPos + i, duplicateBoard)  IF !king.Check(king.xGridPos, king.yGridPos, duplicateBoard)  possibleMoves[xGridPos, yGridPos + i] <- true  ENDIF  ENDIF  IF !possibleMoves[xGridPos, yGridPos + i]  BREAK  ENDIF  ENDIF  ENDFOR  RETURN possibleMoves  ENDSUBROUTINE | |
| **Input**  possibleMoves – the array which will be returned with the results of the subroutine.  board – the board which is being used at that stage in the program. | **Output**  possibleMoves – the array has been filled with true for the positions which the rook can move to. |

* + - * 1. **PossibleMoves – Knight**

|  |  |
| --- | --- |
| **Pseudocode**  SUBROUTINE PossibleMoves(possibleMoves, board)  King king <- NEW King()  IF !IsPositionEmpty(kingX, kingY, board)  IF board[kingX, kingY].pieceType != “king”  GetKing(colour, board)  king.xGridPos = kingX  king.yGridPos = kingY  ELSE  king <- board[kingX, kingY]  ENDIF  ENDIF  IF xGridPos != 7 && yGridPos < 6  IF IsPositionEmpty(xGridPos + 1, yGridPos + 2, board) || (!IsPositionEmpty(xGridPos + 1, yGridPos + 2, board) && DifferentColour(xGridPos + 1, yGridPos + 2, board))  Piece[,] duplicateBoard <- NEW Piece[8, 8]  duplicateBoard <- DuplicateBoard(duplicateBoard, board)  UpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos + 1, yGridPos + 2, duplicateBoard)  IF !king.Check(king.xGridPos, king.yGridPos, duplicateBoard)  possibleMoves[xGridPos + 1, yGridPos + 2] <- true  ENDIF  ENDIF  ENDIF  IF xGridPos < 6 && yGridPos != 7  IF IsPositionEmpty(xGridPos + 2, yGridPos + 1, board) || 0028!IsPositionEmpty(xGridPos + 2, yGridPos + 1, board) && DifferentColour(xGridPos + 2, yGridPos + 1, board))  Piece[,] duplicateBoard <- NEW Piece[8, 8]  duplicateBoard <- DuplicateBoard(duplicateBoard, board)  UpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos + 2, yGridPos + 1, duplicateBoard)  IF !king.Check(king.xGridPos, king.yGridPos, duplicateBoard)  possibleMoves[xGridPos + 2, yGridPos + 1] <- true  ENDIF  ENDIF  ENDIF  IF xGridPos < 6 && yGridPos != 0  IF IsPositionEmpty(xGridPos + 2, yGridPos - 1, board) || (!IsPositionEmpty(xGridPos + 2, yGridPos - 1, board) && DifferentColour(xGridPos + 2, yGridPos - 1, board))  Piece[,] duplicateBoard <- NEW Piece[8, 8]  duplicateBoard <- DuplicateBoard(duplicateBoard, board)  UpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos + 2, yGridPos - 1, duplicateBoard)  IF !king.Check(king.xGridPos, king.yGridPos, duplicateBoard)  possibleMoves[xGridPos + 2, yGridPos - 1] <- true  ENDIF  ENDIF  ENDIF  IF xGridPos != 7 && yGridPos > 1  IF IsPositionEmpty(xGridPos + 1, yGridPos - 2, board) || (!IsPositionEmpty(xGridPos + 1, yGridPos - 2, board) && DifferentColour(xGridPos + 1, yGridPos - 2, board))  Piece[,] duplicateBoard <- NEW Piece[8, 8]  duplicateBoard <- DuplicateBoard(duplicateBoard, board)  UpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos + 1, yGridPos - 2, duplicateBoard)  IF !king.Check(king.xGridPos, king.yGridPos, duplicateBoard)  possibleMoves[xGridPos + 1, yGridPos - 2] <- true  ENDIF  ENDIF  ENDIF  IF xGridPos != 0 && yGridPos > 1  IF IsPositionEmpty(xGridPos - 1, yGridPos - 2, board) || (!IsPositionEmpty(xGridPos - 1, yGridPos - 2, board) && DifferentColour(xGridPos - 1, yGridPos - 2, board))  Piece[,] duplicateBoard <- NEW Piece[8, 8]  duplicateBoard <- DuplicateBoard(duplicateBoard, board)  UpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos - 1, yGridPos - 2, duplicateBoard)  IF !king.Check(king.xGridPos, king.yGridPos, duplicateBoard)  possibleMoves[xGridPos - 1, yGridPos - 2] <- true  ENDIF  ENDIF  ENDIF  IF xGridPos > 1 && yGridPos != 0  IF IsPositionEmpty(xGridPos - 2, yGridPos - 1, board) || (!IsPositionEmpty(xGridPos - 2, yGridPos - 1, board) && DifferentColour(xGridPos - 2, yGridPos - 1, board))  Piece[,] duplicateBoard <- NEW Piece[8, 8]  duplicateBoard <- DuplicateBoard(duplicateBoard, board)  UpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos - 2, yGridPos - 1, duplicateBoard)  IF !king.Check(king.xGridPos, king.yGridPos, duplicateBoard)  possibleMoves[xGridPos - 2, yGridPos - 1] <- true  ENDIF  ENDIF  ENDIF  IF xGridPos > 1 && yGridPos != 7  IF IsPositionEmpty(xGridPos - 2, yGridPos + 1, board) || (!IsPositionEmpty(xGridPos - 2, yGridPos + 1, board) && DifferentColour(xGridPos - 2, yGridPos + 1, board))  Piece[,] duplicateBoard <- NEW Piece[8, 8]  duplicateBoard <- DuplicateBoard(duplicateBoard, board)  UpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos - 2, yGridPos + 1, duplicateBoard)  IF !king.Check(king.xGridPos, king.yGridPos, duplicateBoard)  possibleMoves[xGridPos - 2, yGridPos + 1] <- true  ENDIF  ENDIF  ENDIF  IF xGridPos != 0 && yGridPos < 6  IF IsPositionEmpty(xGridPos - 1, yGridPos + 2, board) || (!IsPositionEmpty(xGridPos - 1, yGridPos + 2, board) && DifferentColour(xGridPos - 1, yGridPos + 2, board))  Piece[,] duplicateBoard <- NEW Piece[8, 8]  duplicateBoard <- DuplicateBoard(duplicateBoard, board)  UpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos - 1, yGridPos + 2, duplicateBoard)  IF !king.Check(king.xGridPos, king.yGridPos, duplicateBoard)  possibleMoves[xGridPos - 1, yGridPos + 2] <- true  ENDIF  ENDIF  ENDIF  RETURN possibleMoves  ENDSUBROUTINE | |
| **Input**  possibleMoves – the array which will be returned with the results of the subroutine.  board – the board which is being used at that stage in the program. | **Output**  possibleMoves – the array has been filled with true for the positions which the knight can move to. |

* + - * 1. **PossibleMoves – King**

|  |  |
| --- | --- |
| **Pseudocode**  SUBROUTINE PossibleMoves(possibleMoves, board)  King king <- NEW King()  IF !IsPositionEmpty(kingX, kingY, board)  IF board[kingX, kingY].pieceType != “king”  GetKing(colour, board)  king.xGridPos = kingX  king.yGridPos = kingY  ELSE  king <- board[kingX, kingY]  ENDIF  ENDIF  IF xGridPos !+ 7 && yGridPos != 7  IF IsPositionEmpty(xGridPos + 1, yGridPos + 1, board) || (!IsPositionEmpty(xGridPos + 1, yGridPos + 1, board) && DifferentColour(xGridPos + 1, yGridPos + 1, board))  IF !Check(xGridPos + 1, yGridPos + 1, board)  possibleMoves[xGridPos + 1, yGridPos + 1] <- true  ENDIF  ENDIF  ENDIF  IF xGridPos != 7  IF IsPositionEmpty(xGridPos + 1, yGridPos, board) || (!IsPositionEmpty(xGridPos + 1, yGridPos, board) && DifferentColour(xGridPos + 1, yGridPos, board))  IF !Check(xGridPos + 1, yGridPos, board)  possibleMoves[xGridPos + 1, yGridPos] <- true  ENDIF  ENDIF  ENDIF  IF xGridPos != 7 && yGridPos != 0  IF IsPositionEmpty(xGridPos + 1, yGridPos - 1, board) || (!IsPositionEmpty(xGridPos + 1, yGridPos - 1, board) && DifferentColour(xGridPos + 1, yGridPos - 1, board))  IF !Check(xGridPos + 1, yGridPos - 1, board)  possibleMoves[xGridPos + 1, yGridPos - 1] <- true  ENDIF  ENDIF  ENDIF  IF yGridPos != 0  IF IsPositionEmpty(xGridPos, yGridPos - 1, board) || (!IsPositionEmpty(xGridPos, yGridPos - 1, board) && DifferentColour(xGridPos, yGridPos - 1, board))  IF !Check(xGridPos, yGridPos - 1, board)  possibleMoves[xGridPos, yGridPos - 1] <- true  ENDIF  ENDIF  ENDIF  IF xGridPos != 0 && yGridPos != 0  IF IsPositionEmpty(xGridPos - 1, yGridPos - 1, board) || (!IsPositionEmpty(xGridPos - 1, yGridPos - 1, board) && DifferentColour(xGridPos - 1, yGridPos - 1, board))  IF !Check(xGridPos - 1, yGridPos - 1, board)  possibleMoves[xGridPos - 1, yGridPos - 1] <- true  ENDIF  ENDIF  ENDIF  IF xGridPos != 0  IF IsPositionEmpty(xGridPos - 1, yGridPos, board) || (!IsPositionEmpty(xGridPos - 1, yGridPos, board) && DifferentColour(xGridPos - 1, yGridPos, board))  IF !Check(xGridPos - 1, yGridPos, board)  possibleMoves[xGridPos - 1, yGridPos] <- true  ENDIF  ENDIF  ENDIF  IF xGridPos != 0 && yGridPos != 7  IF IsPositionEmpty(xGridPos - 1, yGridPos + 1, board) || (!IsPositionEmpty(xGridPos - 1, yGridPos + 1, board) && DifferentColour(xGridPos - 1, yGridPos + 1, board))  IF !Check(xGridPos - 1, yGridPos + 1, board)  possibleMoves[xGridPos - 1, yGridPos + 1] <- true  ENDIF  ENDIF  ENDIF  RETURN possibleMoves  ENDSUBROUTINE | |
| **Input**  possibleMoves – the array which will be returned with the results of the subroutine.  board – the board which is being used at that stage in the program. | **Output**  possibleMoves – the array has been filled with true for the positions which the knight can move to. |

* + - * 1. **PossibleMoves – Queen**

The PossibleMoves algorithm for the Queen class combines the two PossibleMoves algorithms from the Bishop and Rook classes, as queens can move in all directions. For the diagonal moves, see 2.2.1.1.2, and for the horizontal and vertical moves, see 2.2.1.1.3.

* + - 1. **IsPositionEmpty**

|  |  |
| --- | --- |
| **Pseudocode**  SUBROUTINE IsPositionEmpty(x, y, board)  IF board[x, y] = NULL  RETURN true  ELSE  RETURN false  ENDIF  ENDSUBROUTINE | |
| **Input**  x – the x-ordinate for the position to be checked.  y – the y-ordinate for the position to be checked.  board – the board which is being used at this stage in the program. | **Output**  True or false |

* + - 1. **GetKing**

|  |  |
| --- | --- |
| **Pseudocode**  SUBROUTINE GetKing(colour, board)  FOR i <- 0, i < 8, i++  FOR j <- 0, j < 8, j++  IF !IsPositionEmpty(i, j, board)  IF board[i, j].pieceType = “king” && board[i, j].colour = colour  kingX <- i  kingY <- j  kingFound <- true  ENDIF  ENDIF  IF kingFound  BREAK  ENDIF  ENDFOR  IF kingFound  BREAK  ENDFOR  ENDSUBROUTINE | |
| **Input**  colour – the colour of the king to be fetched.  board – the board which is being used at this stage in the program. | **Output**  N/A |

* + - 1. **GetPieceCount**

|  |  |
| --- | --- |
| **Pseudocode**  SUBROUTINE GetPieceCount(board)  int pieceCount <- 0  FOR i <- 0, i < 8, i++  FOR j <- 0, j < 8, j++  IF !IsPositionEmpty(i, j, board)  pieceCount++  ENDIF  IF pieceCount = 32  BREAK  ENDIF  ENDFOR  IF pieceCount = 32  BREAK  ENDIF  ENDFOR  RETURN pieceCount  ENDSUBROUTINE | |
| **Input**  board – the board which is being used at this stage in the program. | **Output**  pieceCount – the number of pieces on the board. |

* + - 1. **Check**

|  |  |
| --- | --- |
| **Pseudocode**  SUBROUTINE Check(x, y, board)  FOR i <- 0, i < 8, i++  FOR j <- 0, j < 8, j++  IF !IsPositionEmpty(i, j, board)  possibleMoves <- board[i, j].PossibleMoves(possibleMoves, board)  IF possibleMoves[x, y]  RETURN true  ENDIF  ENDIF  ENDFOR  ENDFOR  RETURN false  ENDSUBROUTINE | |
| **Input**  x – the x-ordinate of the position to be checked.  y – the y-ordinate of the position to be checked.  board – the board which is being used at this stage in the program. | **Output**  True or false |

During testing, I discovered this algorithm resulted in a stack overflow. Therefore, I had to rewrite the algorithm without using PossibleMoves.

|  |  |
| --- | --- |
| **Pseudocode**  SUBROUTINE Check(x, y, board)  FOR i <- x + 1, i < 8, i++  bool breakLoop <- false  FOR j <- y + 1, j < 8, j++  IF i - x = j - y  IF !IsPositionEmpty(i, j, board)  IF DifferentColour(i, j, board)  IF board[i, j].pieceType = “bishop” || board[i, j].pieceType = “queen” || (board[i, j].pieceType = “pawn” && i = x + 1 && j = y + 1)  RETURN true  ELSE  BREAK  ENDIF  ENDIF  ENDIF  ENDIF  ENDFOR  IF breakLoop  BREAK  ENDIF  ENDFOR  FOR i <- x - 1, i > -1, i--  bool breakLoop <- false  FOR j <- y - 1, j > -1, j--  IF i - x = j - y  IF !IsPositionEmpty(i, j, board)  IF DifferentColour(i, j, board)  IF board[i, j].pieceType = “bishop” || board[i, j].pieceType = “queen” || (board[i, j].pieceType = “pawn” && i = x - 1 && j = y - 1)  RETURN true  ELSE  BREAK  ENDIF  ENDIF  ENDIF  ENDIF  ENDFOR  IF breakLoop  BREAK  ENDIF  ENDFOR  FOR i <- x + 1, i < 8, i++  bool breakLoop <- false  FOR j <- y - 1, j > -1, j--  IF i - x = -(j – y)  IF !IsPositionEmpty(i, j, board)  IF DifferentColour(i, j, board)  IF board[i, j].pieceType = “bishop” || board[i, j].pieceType = “queen” || (board[i, j].pieceType = “pawn” && i = x + 1 && j = y - 1)  RETURN true  ELSE  BREAK  ENDIF  ENDIF  ENDIF  ENDIF  ENDFOR  IF breakLoop  BREAK  ENDIF  ENDFOR  FOR i <- x - 1, i > -1, i--  bool breakLoop <- false  FOR j <- y + 1, j < 8, j++  IF –(i – x) = j - y  IF !IsPositionEmpty(i, j, board)  IF DifferentColour(i, j, board)  IF board[i, j].pieceType = “bishop” || board[i, j].pieceType = “queen” || (board[i, j].pieceType = “pawn” && i = x - 1 && j = y + 1)  RETURN true  ELSE  BREAK  ENDIF  ENDIF  ENDIF  ENDIF  ENDFOR  IF breakLoop  BREAK  ENDIF  ENDFOR  FOR i <- x + 1, i < 8, i++  IF !IsPositionEmpty(i, y, board)  IF DifferentColour(i, y, board)  IF board[i, y].pieceType = “rook” || board[i, y].pieceType = “queen”  RETURN true  ELSE  BREAK  ENDIF  ENDIF  ENDIF  ENDFOR  FOR i <- x - 1, i > -1, i--  IF !IsPositionEmpty(i, y, board)  IF DifferentColour(i, y, board)  IF board[i, y].pieceType = “rook” || board[i, y].pieceType = “queen”  RETURN true  ELSE  BREAK  ENDIF  ENDIF  ENDIF  ENDFOR  FOR i <- y + 1, i < 8, i++  IF !IsPositionEmpty(x, i, board)  IF DifferentColour(x, i, board)  IF board[x, i].pieceType = “rook” || board[i, y].pieceType = “queen”  RETURN true  ELSE  BREAK  ENDIF  ENDIF  ENDIF  ENDFOR  FOR i <- y - 1, i > -1, i--  IF !IsPositionEmpty(x, i, board)  IF DifferentColour(x, i, board)  IF board[x, i].pieceType = “rook” || board[i, y].pieceType = “queen”  RETURN true  ELSE  BREAK  ENDIF  ENDIF  ENDIF  ENDFOR  IF x < 7 && y < 6  IF !IsPositionEmpty(x + 1, y + 2, board)  IF board[x + 1, y + 2, board].pieceType = “knight”  RETURN true  ENDIF  ENDIF  ENDIF  IF x < 7 && y > 1  IF !IsPositionEmpty(x + 1, y - 2, board)  IF board[x + 1, y - 2, board].pieceType = “knight”  RETURN true  ENDIF  ENDIF  ENDIF  IF x > 0 && y < 6  IF !IsPositionEmpty(x - 1, y + 2, board)  IF board[x - 1, y + 2, board].pieceType = “knight”  RETURN true  ENDIF  ENDIF  ENDIF  IF x > 0 && y > 1  IF !IsPositionEmpty(x - 1, y - 2, board)  IF board[x - 1, y - 2, board].pieceType = “knight”  RETURN true  ENDIF  ENDIF  ENDIF  IF x > 1 && y < 7  IF !IsPositionEmpty(x - 2, y + 1, board)  IF board[x - 2, y + 1, board].pieceType = “knight”  RETURN true  ENDIF  ENDIF  ENDIF  IF x > 1 && y > 0  IF !IsPositionEmpty(x - 2, y - 1, board)  IF board[x - 2, y - 1, board].pieceType = “knight”  RETURN true  ENDIF  ENDIF  ENDIF  IF x < 6 && y > 0  IF !IsPositionEmpty(x + 2, y - 1, board)  IF board[x + 2, y - 1, board].pieceType = “knight”  RETURN true  ENDIF  ENDIF  ENDIF  IF x < 6 && y < 7  IF !IsPositionEmpty(x + 2, y + 1, board)  IF board[x + 2, y + 1, board].pieceType = “knight”  RETURN true  ENDIF  ENDIF  ENDIF  RETURN false  ENDSUBROUTINE | |
| **Input**  x – the x-ordinate of the position to be checked.  y – the y-ordinate of the position to be checked.  board – the board which is being used at this stage in the program. | **Output**  True or false |

* + - 1. **Checkmate**

|  |  |
| --- | --- |
| **Pseudocode**  SUBROUTINE Checkmate(x, y, board)  IF !Check(x, y, board)  RETURN false  ENDIF  PossibleMoves(possibleMoves, board)  FOR i <- 0, i < 8, i++  FOR j <- 0, j < 8, j++  IF possibleMoves[i, j]  RETURN false  ENDIF  ENDFOR  ENDFOR  FOR i <- 0, i < 8, i++  FOR j <- 0, j < 8, j++  IF !IsPositionEmpty(i, j, board) && DifferentColour(i, j, board)  possibleMoves = board[i, j].PossibleMoves(possibleMoves, board)  FOR k <- 0, k < 8, k++  FOR l <- 0, l < 8, l++  IF possibleMoves[k, l]  RETURN false  ENDIF  ENDFOR  ENDFOR  ENDIF  ENDFOR  ENDFOR  RETURN true  ENDSUBROUTINE | |
| **Input**  x – the x-ordinate of the position to be checked.  y – the y-ordinate of the position to be checked.  board – the board which is being used at this stage in the program. | **Output**  True or false |

* + - 1. **QueenConvert**

|  |  |
| --- | --- |
| **Pseudocode**  SUBROUTINE QueenConvert(pawn, board)  RemovePiece(pawn, board)  Queen newQueen = NEW Queen()  newQueen.colour = colour  newQueen.pieceType = “queen”  newQueen.xGridPos = xGridPos  newQueen.yGridPos = yGridPos  IF board = grid  IF colour = “white”  pieceImages[xGridPos, yGridPos].Image = queen\_white  ELSE  pieceImages[xGridPos, yGridPos].Image = queen\_black  ENDIF  ENDIF  AddPiece(newQueen, board)  ENDSUBROUTINE | |
| **Input**  pawn – the pawn to be removed.  board – the board which is being used at this stage in the program. | **Output**  N/A |

During testing, a separate version of this code was made in the AI class for the AI to use without affecting the main board.

* + - 1. **DifferentColour**

|  |  |
| --- | --- |
| **Pseudocode**  SUBROUTINE DifferentColour(x, y, board)  IF colour != board[x, y].colour  RETURN true  ELSE  RETURN false  ENDIF  ENDSUBROUTINE | |
| **Input**  x – the x-ordinate of the position to be checked.  y – the y-ordinate of the position to be checked.  board – the board which is being used at this stage in the program. | **Output**  True or false |

* + - 1. **AddPiece**

|  |  |
| --- | --- |
| **Pseudocode**  SUBROUTINE AddPiece(piece, board)  board[xGridPos, yGridPos] <- piece | |
| **Input**  piece – the piece to be added.  board – the board which is being used at this stage in the program. | **Output**  N/A |

* + - 1. **RemovePiece**

|  |  |
| --- | --- |
| **Pseudocode**  SUBROUTINE RemovePiece(piece, board)  IF board = grid  takenPieceImage <- pieceImages[piece.xGridPos, piece.yGridPos]  grid[piece.xGridPos, piece.yGridPos] <- NULL  pieceImages[piece.xGridPos, piece.yGridPos].Visible <- false  pieceImages[piece.xGridPos, piece.yGridPos] <- NULL  ELSE  board[piece.xGridPos, piece.yGridPos] <- NULL  ENDIF  ENDSUBROUTINE | |
| **Input**  piece – the piece to be removed.  board – the board which is being used at this stage in the program. | **Output**  N/A |

During testing, a separate version of this code was made in the AI class for the AI to use without affecting the main board.

* + - 1. **UpdatePosition**

|  |  |
| --- | --- |
| **Pseudocode**  SUBROUTINE UpdatePosition(piece, currentX, currentY, newX, newY, board)  IF piece != null  IF IsPositionEmpty(newX, newY, board)  board[currentX, currentY] <- NULL  board[newX, newY] <- piece  ELSE  RemovePiece(board[newX, newY], board)  board[currentX, currentY] <- NULL  board[newX, newY] <- piece  ENDIF  IF board = grid  pieceImages[newX, newY] <- pieceImages[currentX, currentY]  pieceImages[currentX, currentY] <- NULL  pieceImages[newX, newY].Location <- NEW Point(100 + (150 \* newX), 100 + (150 \* newY))  piece.xGridPos <- newX  piece.yGridPos <- newY  IF (newX = 0 || newX = 2 || newX = 4 || newX = 6) && (newY = 0 || newY = 2 || newY = 4 || newY = 6)  pieceImages[newX, newY].BackColor = LightYellow  ELSE IF (newX = 1 || newX = 3 || newX = 5 || newX = 7) && (newY = 1 || newY = 3 || newY = 5 || newY = 7)  pieceImages[newX, newY].BackColor = LightYellow  ELSE  pieceImages[newX, newY].BackColor = Brown  ENDIF  ENDIF  IF pieceType = “pawn”  Pawn pawnConvert <- NEW Pawn()  IF colour = “white”  IF yGridPos = 0  pawnConvert.QueenConvert(pawnConvert, board)  ENDIF  ELSE  IF yGridPos = 7  pawnConvert.QueenConvert(pawnConvert, board)  ENDIF  ENDIF  ENDIF  ENDSUBROUTINE | |
| **Input**  Piece – the piece to have its position updated.  currentX - the current x-ordinate of the piece.  currentY – the current y-ordinate of the piece.  newX – the new x-ordinate of the piece.  newY – the new y-ordinate of the piece.  board – the board which is being used at this stage in the program. | **Output**  N/A |

During testing, a separate version of this code was made in the AI class for the AI to use without affecting the main board.

* + - 1. **CreatePieces**

|  |  |
| --- | --- |
| **Pseudocode**  SUBROUTINE CreatePieces  FOR i <- 0, i < 2, i++  CreateQueen(i)  CreateKing(i)  FOR j <- 0, j < 2, j++  CreateRook(i, j)  CreateKnight(i, j)  CreateBishop(i, j)  ENDFOR  FOR j <- 0, j < 8, j++  CreatePawn(i, j)  ENDFOR  ENDFOR  ENDSUBROUTINE | |
| **Input**  N/A | **Output**  N/A |

* + - * 1. **CreatePawn**

|  |  |
| --- | --- |
| **Pseudocode**  SUBROUTINE CreatePawn(i, j)  Pawn newPawn <- NEW Pawn()  newPawn.pieceType <- “pawn”  PictureBox newPictureBox = NEW PictureBox()  IF i = 0  newPawn.colour <- “white”  newPawn.xGridPos <- j  newPawn.yGridPos <- 6  newPawn.strength <- 10  ELSE  newPawn.colour <- “black”  newPawn.xGridPos <- j  newPawn.yGridPos <- 1  newPawn.strength <- -10  ENDIF  newPawn.AddPiece(newPawn, grid)  CreatePieceImages(newPawn, newPictureBox, newPawn.colour)  ENDSUBROUTINE | |
| **Input**  i – if i = zero the pawn will be white. If i = one the pawn will be black.  j – determines the x position of the pawn. | **Output**  N/A |

* + - * 1. **CreateBishop**

|  |  |
| --- | --- |
| **Pseudocode**  SUBROUTINE CreateBishop(i, j)  Bishop newBishop <- NEW Bishop()  newBishop.pieceType <- “bishop”  PictureBox newPictureBox <- NEW PictureBox()  IF i = 0  IF j = 0  newBishop.colour <- “white”  newBishop.xGridPos <- 2  newBishop.yGridPos <- 7  ELSE  newBishop.colour <- “white”  newBishop.xGridPos <- 5  newBishop.yGridPos <- 7  ENDIF  newBishop.strength <- 30  ELSE  IF j = 0  newBishop.colour <- “black”  newBishop.xGridPos <- 2  newBishop.yGridPos <- 0  ELSE  newBishop.colour <- “black”  newBishop.xGridPos <- 5  newBishop.yGridPos <- 0  ENDIF  newBishop.strength <- -30  ENDIF  newBishop.AddPiece(newBishop, grid)  CreatePieceImages(newBishop, newPictureBox, newBishop.colour)  ENDSUBROUTINE | |
| **Input**  i – if i = zero the bishop will be white. If i = one the bishop will be black.  j – determines the x position of the bishop. If j = zero the x position will be two. If j = one the x position will be five. | **Output**  N/A |

* + - * 1. **CreateRook**

|  |  |
| --- | --- |
| **Pseudocode**  SUBROUTINE CreateRook(i, j)  Rook newRook <- NEW Rook()  newRook.pieceType <- “rook”  PictureBox newPictureBox <- NEW PictureBox()  IF i = 0  IF j = 0  newRook.colour <- “white”  newRook.xGridPos <- 0  newRook.yGridPos <- 7  ELSE  newRook.colour <- “white”  newRook.xGridPos <- 7  newRook.yGridPos <- 7  ENDIF  newRook.strength <- 50  ELSE  IF j = 0  newRook.colour <- “black”  newRook.xGridPos <- 0  newRook.yGridPos <- 0  ELSE  newRook.colour <- “black”  newRook.xGridPos <- 7  newRook.yGridPos <- 0  ENDIF  newRook.strength <- -50  ENDIF  newRook.AddPiece(newRook, grid)  CreatePieceImages(newRook, newPictureBox, newRook.colour)  ENDSUBROUTINE | |
| **Input**  i – if i = zero the rook will be white. If i = one the rook will be black.  j – determines the x position of the rook. If j = zero the x position will be zero. If j = one the x position will be seven. | **Output**  N/A |

* + - * 1. **CreateKnight**

|  |  |
| --- | --- |
| **Pseudocode**  SUBROUTINE CreateKnight(i, j)  Knight newKnight <- NEW Knight()  newKnight.pieceType <- “knight”  PictureBox newPictureBox <- NEW PictureBox()  IF i = 0  IF j = 0  newKnight.colour <- “white”  newKnight.xGridPos <- 1  newKnight.yGridPos <- 7  ELSE  newKnight.colour <- “white”  newKnight.xGridPos <- 6  newKnight.yGridPos <- 7  ENDIF  newKnight.strength <- 30  ELSE  IF j = 0  newKnight.colour <- “black”  newKnight.xGridPos <- 1  newKnight.yGridPos <- 0  ELSE  newBishop.colour <- “black”  newKnight.xGridPos <- 6  newKnight.yGridPos <- 0  ENDIF  newKnight.strength <- -30  ENDIF  newKnight.AddPiece(newKnight, grid)  CreatePieceImages(newKnight, newPictureBox, newKnight.colour)  ENDSUBROUTINE | |
| **Input**  i – if i = zero the knight will be white. If i = one the knight will be black.  j – determines the x position of the knight. If j = zero the x position will be one. If j = one the x position will be six. | **Output**  N/A |

* + - * 1. **CreateKing**

|  |  |
| --- | --- |
| **Pseudocode**  SUBROUTINE CreateKing(i)  King newKing <- NEW King()  newKing.pieceType <- “king”  PictureBox newPictureBox <- NEW PictureBox()  IF i = 0  newKing.colour <- “white”  newKing.xGridPos <- 4  newKing.yGridPos <- 7  newKing.strength <- 900  ELSE  newKing.xGridPos <- 4  newKing.yGridPos <- 0  newKing.strength <- -900  ENDIF  newKing.AddPiece(newKing, grid)  CreatePieceImages(newKing, newPictureBox, newKing.colour)  ENDSUBROUTINE | |
| **Input**  i – if i = zero the king will be white. If i = one the king will be black. | **Output**  N/A |

* + - * 1. **CreateQueen**

|  |  |
| --- | --- |
| **Pseudocode**  SUBROUTINE CreateQueen(i)  Queen newQueen <- NEW Queen()  newQueen.pieceType <- “queen”  PictureBox newPictureBox <- NEW PictureBox()  IF i = 0  newQueen.colour <- “white”  newQueen.xGridPos <- 3  newQueen.yGridPos <- 7  newQueen.strength <- 90  ELSE  newQueen.xGridPos <- 3  newQueen.yGridPos <- 0  newQueen.strength <- -90  ENDIF  newQueen.AddPiece(newQueen, grid)  CreatePieceImages(newQueen, newPictureBox, newQueen.colour)  ENDSUBROUTINE | |
| **Input**  i – if i = zero the queen will be white. If i = one the queen will be black. | **Output**  N/A |

* + - 1. **GameOver**

|  |  |
| --- | --- |
| **Pseudocode**  SUBROUTINE GameOver()  IF checkmate  IF whiteWin  MessageBox.Show “Checkmate! White wins!”  ELSE  MessageBox.Show “Checkmate! Black wins!”  ENDIF  ELSE IF stalemate  MessageBox.Show “Stalemate.”  ELSE IF whiteTimer <= 0  MessageBox.Show “White ran out of time. Black wins!”  ELSE IF blackTimer <= 0  MessageBox.Show “Black ran out of time. White wins!”  ENDIF  FOR i <- 0, i < 8, i++  FOR j <- 0, j < 8, j++  IF !IsPositionEmpty(i, j, grid)  grid[i, j].RemovePiece(grid[i, j], grid)  ENDIF  ENDFOR  ENDFOR  NewGameButton.Enabled <- true  LoadGameButton.Enabled <- true  SaveGameButton.Enabled <- false  IF timeEnabled  WhiteTimer.Text <- “0:00”  BlackTimer.Text <- “0:00”  WhiteTimer.Visible <- false  BlackTimer.Visible <- false  ENDIF  ENDSUBROUTINE | |
| **Input**  N/A | **Output**  N/A |

* + - 1. **UpdateTimer**

|  |  |
| --- | --- |
| **Pseudocode**  SUBROUTINE UpdateTimer()  IF whiteTurn  TimeSpan ts <- whiteStopwatch.Elapsed  whiteTimer –= ((ts.Minutes \* 60) + ts.Seconds)  IF whiteTimer % 60 >= 10  whiteTimer.Text <- Convert.ToString(whiteTimer / 60) + “:” + Convert.ToString(whiteTimer % 60)  ELSE  whiteTimer.Text <- Convert.ToString(whiteTimer / 60) + “:0” + Convert.ToString(whiteTimer % 60)  ENDIF  whiteStopwatch.Reset()  ELSE  TimeSpan ts <- blackStopwatch.Elapsed  blackTimer -= (ts.Minutes \* 60) + ts.Seconds)  IF blackTimer % 60 >= 10  blackTimer.Text <- Convert.ToString(blackTimer / 60) + “:” + Convert.ToString(blackTimer % 60)  ELSE  blackTimer.Text <- Convert.ToString(blackTimer / 60) + “:0” + Convert.ToString(blackTimer % 60)  ENDIF  blackStopwatch.Reset()  ENDIF  ENDSUBROUTINE | |
| **Input**  N/A | **Output**  N/A |

* + - 1. **Turn**

|  |  |
| --- | --- |
| **Pseudocode**  SUBROUTINE Turn()  IF playerVComputer  DO  IF whiteTurn  PlayerTurn()  ELSE  AI.Turn()  ENDIF  EndOfTurn()  WHILE !gameOver  ELSE  DO  IF whiteTurn  IF timeEnabled  WhiteTimer.ForeColor <- Red  whiteStopwatch.Start()  PlayerTurn()  ENDIF  ELSE  IF timeEnabled  BlackTimer.ForeColor <- Red  blackStopwatch.Start()  PlayerTurn()  ENDIF  ENDIF  WHILE !gameOver  ENDIF  ENDSUBROUTINE | |
| **Input**  N/A | **Output**  N/A |

During testing this algorithm was removed as it resulted in the UI freezing and the PlayerTurn algorithm didn’t really do anything.

* + - 1. **EndOfTurn**

|  |  |
| --- | --- |
| **Pseudocode**  SUBROUTINE EndOfTurn()  GetKing(“white”, grid)  King whiteKing <- NEW King()  whiteKing.xGridPos <- kingX  whiteKing.yGridPos <- kingY  whiteKing.colour <- “white”  GetKing(“black”, grid)  King blackKing <- NEW King()  blackKing.xGridPos <- kingX  blackKing.yGridPos <- kingY  blackKing.colour <- “black”  IF whiteKing.Checkmate(whiteKing.xGridPos, whiteKing.yGridPos, grid) || blackKing.Checkmate(blackKing.xGridPos, blackKing.yGridPos, grid) || Stalemate(grid)  IF whiteKing.Checkmate(whiteKing.xGridPos, whiteKing.yGridPos, grid)  whiteWin <- false  checkmate <- true  stalemate <- false  gameOver <- true  ELSE IF blackKing.Checkmate(blackKing.xGridPos, blackKing.yGridPos, grid)  whiteWin <- true  checkmate <- true  stalemate <- false  gameOver <- true  ELSE IF Stalemate(grid)  whiteWin <- false  checkmate <- false  stalemate <- true  gameOver <- true  ENDIF  GameOver()  RETURN GameOver  IF whiteTurn  IF timeEnabled  whiteStopwatch.Stop()  WhiteTimer.ForeColor <- Green  UpdateTimer()  IF whiteTimer <= 0  gameOver <- true  whiteWin <- false  stalemate <- false  checkmate <- false  GameOver()  RETURN gameOver  ENDIF  ENDIF  whiteTurn <- false  ELSE  IF timeEnabled  blackStopwatch.Stop()  BlackTimer.ForeColor <- Green  UpdateTimer()  IF blackTimer <= 0  gameOver <- true  whiteWin <- true  stalemate<- false  checkmate <- false  GameOver  RETURN gameOver  ENDIF  ENDIF  whiteTurn <- true  ENDIF  IF takenPiece != NULL  IF takenPiece.colour = “white”  WhiteTakenListBox.Items.Add(takenPiece)  ELSE  BlackTakenListBox.Items.Add(takenPiece)  ENDIF  takenPiece <- NULL  ENDIF  ENDIF  RETURN gameOver  ENDSUBROUTINE | |
| **Input**  N/A | **Output**  gameOver - True or false |

* + - 1. **PickOrDropPiece**

The following algorithm is taken from a stackoverflow user called aybe.[[4]](#footnote-4) I have modified their code to fit with my existing methods. aybe’s code is located in the solution at the bottom of the webpage linked in the footnote.

|  |  |
| --- | --- |
| **Pseudocode**  SUBROUTINE PickOrDropPiece(e)  Point location <- e.Location  int x <- location.X / 150  int y <- location.Y / 150  bool pickOrDrop  string colour  IF whiteTurn  colour <- “white”  ELSE  colour <- “black”  ENDIF  IF currentPiece = NULL  pickOrDrop <- true  ELSE  pickOrDrop <- false  ENDIF  IF pickOrDrop  Piece piece <- grid[x, y]  IF piece != NULL  IF piece.colour = colour  MoveLabel.Text <- Convert.ToString(colour + “ has selected a “ + piece.colour + “ “ + piece.pieceType + “ at (“ + x + “,” + y + “).”)  ENDIF  ELSE  MoveLabel.Text <- Convert.ToString(colour + “ has not selected a piece.”)  ENDIF  currentPiece <- piece  ELSE  possibleMoves <- currentPiece.PossibleMoves(possibleMoves, grid)  IF possibleMoves[x, y]  currentPiece.UpdatePosition(currentPiece, currentPiece.xGridPos, currentPiece.yGridPos, x, y, grid)  MoveLabel.Text <- Convert.ToString(colour + “ has moved a “ + piece.colour + “ “ + piece.pieceType + “ to (“ + x + “,” + y + “).”)  currentPiece <- NULL  pieceMoved <- true  ELSE  MoveLabel.Text <- “Illegal move.”  ENDIF  ENDIF  ENDSUBROUTINE | |
| **Input**  e – a MouseEventArgs, stores data about the mouse click. | **Output**  N/A |

During testing, this algorithm was changed to OnMouseClick\_newPictureBox, as it is linked to an event handler I added for picture boxes created in CreatePieceImages. This algorithm was later removed altogether to implement movement by asking the user to enter co-ordinates.

* + - 1. **Stalemate**

|  |  |
| --- | --- |
| **Pseudocode**  SUBROUTINE Stalemate(board)  bool whiteStalemateCombination <- StalemateCombinations(“white”, board)  bool blackStalemateCombination <- StalemateCombinations(“black”, board)  IF whiteStalemateCombination && blackStalemateCombination  stalemate <- true  RETURN stalemate  ENDIF  GetKing(“white”, board)  King whiteKing <- NEW King()  whiteKing.xGridPos <- kingX  whiteKing.yGridPos <- kingY  GetKing(“black”, board)  King blackKing <-NEW King()  blackKing.xGridPos <- kingX  blackKing.yGridPos <- kingY  FOR i <- 0, i < 8, i++  FOR j <- 0, j < 8, j++  IF !IsPositionEmpty(i, j, board)  FOR k <- 0, k < 8, k++  FOR l <- 0, l < 8, l++  IF board[i, j].possibleMoves[k, l]  stalemate <- false  RETURN stalemate  ENDIF  ENDFOR  ENDFOR  ENDIF  ENDFOR  ENDFOR  IF whiteKing.Check(whiteKing.xGridPos, whiteKing.yGridPos, board) || blackKing.Check(blackKing.xGridPos, blackKing.yGridPos, board)  stalemate <- false  RETURN stalemate  ENDIF  stalemate <- true  RETURN stalemate  ENDSUBROUTINE | |
| **Input**  board – the board which is being used at this stage in the program. | **Output**  stalemate – true or false. |

* + - 1. **StalemateCombinations**

|  |  |
| --- | --- |
| **Pseudocode**  SUBROUTINE StalemateCombinations(colour, board)  int pawnCount <- 0  int bishopCount <- 0  int knightCount <- 0  int rookCount <- 0  int kingCount <- 0  int queenCount <- 0  FOR i <- 0, i < 8, i++  FOR j <- 0, j < 8, j++  IF !IsPositionEmpty(i, j, board)  IF board[i, j].colour = colour  IF board[i, j].pieceType = “pawn”  pawnCount++  ELSE IF board[i, j].pieceType = “bishop”  bishopCount++  ELSE IF board[i, j].pieceType = “knight”  knightCount++  ELSE IF board[i, j].pieceType = “rook”  rookCount++  ELSE IF board[i, j].pieceType = “king”  kingCount++  ELSE IF board[i, j].pieceType = “queen”  queenCount++  ENDIF  ENDIF  ENDIF  ENDFOR  ENDFOR  IF kingCount = 1 && knightCount = 1 && pawnCount = 0 && bishopCount = 0 && rookCount = 0 && queenCount = 0  RETURN true  ELSE IF kingCount = 1 && knightCount = 2 && pawnCount = 0 && bishopCount = 0 && rookCount = 0 && queenCount = 0  RETURN true  ELSE IF kingCount = 1 && knightCount = 0 && pawnCount = 0 && bishopCount = 1 && rookCount = 0 && queenCount = 0  RETURN true  ENDIF  RETURN false  ENDSUBROUTINE | |
| **Input**  colour – the colour of the pieces to be checked.  board – the board which is being used at this stage in the program. | **Output**  True or false. |

* + 1. **AI algorithms**

The following algorithms are grouped as they are in the AI class.

* + - 1. **DuplicateBoard**

|  |  |
| --- | --- |
| **Pseudocode**  SUBROUTINE DuplicateBoard(newBoard, oldBoard)  FOR i <- 0, i < 8, i++  FOR j <- 0, j < 8, j++  IF !IsPositionEmpty(i, j, oldBoard)  newBoard[i, j] <- oldBoard[i, j]  ENDIF  ENDFOR  ENDFOR  RETURN newBoard  ENDSUBROUTINE | |
| **Input**  newBoard – the duplicated board.  oldBoard – the board to be duplicated. | **Output**  newBoard – the duplicated board. |

* + - 1. **GenerateMoves**

|  |  |
| --- | --- |
| **Pseudocode**  SUBROUTINE GenerateMoves(board, depth, whiteTurn)  FOR i <- 0, i < 8, i++  FOR j <- 0, j < 8, j++  IF whiteTurn  IF !IsPositionEmpty(i, j, board) && board[i, j].colour = “white”  possibleMoves <- board[i, j].PossibleMoves(possibleMoves, board)  ENDIF  ELSE  IF !IsPositionEmpty(i, j, board) && board[i, j].colour = “black”  possibleMoves <- board[i, j].PossibleMoves(possibleMoves, board)  ENDIF  ENDIF  FOR k <- 0, k < 8, k++  FOR l <- 0, l < 8, l++  IF possibleMoves[k, l]  IF !IsPositionEmpty(i, j, board)  board[i, j].UpdatePosition(board[i, j], i, j, k, l, board)  Piece[,] move <- NEW Piece[8, 8]  Move <- DuplicateBoard(move, board)  moves.Add(move)  IF depth = searchDepth && board[i, j].colour = “black”  movedPieces.Add(board[i, j])  ENDIF  ENDIF  ENDIF  ENDFOR  ENDFOR  ENDFOR  ENDFOR  RETURN moves  ENDSUBROUTINE | |
| **Input**  board – the board which is being used at this stage in the program.  depth – the number of layers to be searched.  whiteTurn – whose turn it is. | **Output**  moves – a list of the moves generated. |

* + - 1. **Evaluation**

|  |  |
| --- | --- |
| **Pseudocode**  SUBROUTINE Evaluation(board, checkForWhite)  int evaluation <- 0  FOR i <- 0, i < 8, i++  FOR j <- 0, j < 8, j++  IF !IsPositionEmpty(i, j, Board)  IF board[i, j].colour = “white”  evaluation += board[i, j].strength  ELSE  Evaluation -= board[i, j].strength  ENDIF  ENDIF  ENDFOR  ENDFOR  IF checkForWhite  RETURN evaluation  ELSE  RETURN evaluation \* -1  ENDIF  ENDSUBROUTINE | |
| **Input**  board – the board which is being used at this stage in the program.  checkForWhite – determines whether the evaluation is being done for the white or black player. | **Output**  evaluation – the result of adding up the strengths of all of the pieces on the board. |

* + - 1. **Minimax**

The following algorithm is taken from a stackoverflow user called foRei.[[5]](#footnote-5) I have added alpha-beta pruning to their algorithm to optimise the search. foRei’s code is located in the solution at the bottom of the webpage linked in the footnote.

|  |  |
| --- | --- |
| **Pseudocode**  SUBROUTINE Minimax(board, depth, maximisingPlayer, whiteTurn, alpha, beta)  IF depth = 0  result <- Evaluation(board, whiteTurn)  RETURN result  ENDIF  moves <- GenerateMoves(board, depth, whiteTurn)  IF maximisingPlayer  value <- -1290  FOREACH move IN moves  minimaxResult <- Minimax(move, depth – 1, false, !whiteTurn, alpha, beta)  value <- MAX(value, minimaxResult)  alpha <- MAX(alpha, value)  IF beta <= alpha  BREAK  ENDIF  IF depth = searchDepth  moveScores.Add(minimaxResult)  ENDIF  ENDFOREACH  RETURN value  ELSE  value <- 1290  FOREACH move IN moves  minimaxResult <- Minimax(move, depth – 1, true, !whiteTurn, alpha, beta)  value <- MIN(value, minimaxResult)  beta <- MIN(beta, value)  IF beta <= alpha  BREAK  ENDIF  IF depth = searchDepth  moveScores.Add(minimaxResult)  ENDIF  ENDFOREACH  RETURN value  ENDIF  ENDSUBROUTINE | |
| **Input**  board – the board which is being used at this stage in the program.  depth – the search depth.  maximisingPlayer – determines whether the search is being done for white or black.  whiteTurn – determines whose turn it is.  alpha – used for pruning.  beta – used for pruning. | **Output**  minimaxResult – the result of the algorithm. |

* + - 1. **Turn**

|  |  |
| --- | --- |
| **Pseudocode**  SUBROUTINE Turn()  IF moveScores.Count > 0  moveScores.Clear()  ENDIF  IF movedPieces.Count > 0  movedPieces.Clear()  ENDIF  Piece[,] board <- DuplicateBoard(board, grid)  Minimax(board, searchDepth, false, false, -1290, 1290)  movingPiece <- movedPieces[resultPos]  grid[movingPiece.xGridPos, movingPiece.yGridPos].UpdatePosition(movingPiece, movingPiece.xGridPos, movingPiece.yGridPos, movingPiece.newX, movingPiece.newY, grid)  ENDSUBROUTINE | |
| **Input**  N/A | **Output**  N/A |

* 1. **Data Structures**
     1. **Tree**

The minimax algorithm searches for the best move using a tree. The root node of this tree is the current state of the board using the evaluation algorithm, and every leaf node represents the state of the board after a move is played. The minimax algorithm searches the tree depth-first. A simplified example can be seen here:

A picture containing graphical user interface

Description automatically generated

In this example, the algorithm is searching to a depth of two. The value returned by the evaluation algorithm at the root node is 50. White tries to maximise the outcome, and black tries to minimise it.

Therefore, on a white turn, the move with the highest outcome will be selected, and on a black turn, the move with the lowest outcome will be selected. In this example, black would choose the node which results in -40, as this is the lowest available. From there, white would choose -10, as this is the highest outcome available.

* + 1. **Stack**

The minimax algorithm also creates a stack, as it calls itself recursively. The first call of the algorithm calls the minimax algorithm again, and the result of this call is required in order to return the result of the first call. As a stack is first in, last out structure, the first call of the stack will be the last one to be solved.

* 1. **User Interface**
     1. **Design**

When the program starts running, this is what the interface looks like:

A picture containing background pattern

Description automatically generated

Initially, the buttons to start, pause, and save the game are disabled, as well as the buttons to set the mode and the timer. Upon pressing New Game, the buttons to set the mode are enabled, and the chess pieces are created and placed on the screen.

A picture containing square

Description automatically generated

From here, there are two different options: set the game to one player, or two player. If the game is set to one player, the start button is enabled. If the game is set to two player, the buttons to set the timer are enabled, and once the timer is set, two timers appear on the screen and the start button is enabled. Once the start button is enabled, the game is ready to begin.

A picture containing square

Description automatically generated

* + 1. **Components**

Buttons:

* Start – the start button is the ‘>’ button in the top right corner of the interface. It is used to start the game or to resume the game when paused.
* Pause – the pause button is the ‘||’ button in the top right corner of the interface. It is used to pause the game after it has been started.
* New Game – creates the pieces and ensures all variables are correctly set for a new game starting.
* Save Game – saves the state of the game to a .txt file called ‘savedGame’.
* Load Game – loads the state of the game from savedGame.txt and creates the pieces and sets the variables appropriately to continue the game.
* Set Mode – sets the mode to either one or two player using the appropriate radio buttons.
* Set Timer – sets the timer to either one, five, or ten minutes using the appropriate radio buttons. This is only available when playing two player.

Board and Pieces:

* The board is created at runtime using panels.
* Each panel is 150 pixels by 150 pixels.
* Labels from one to eight are added to both axis of the board.
* The pieces are also created at runtime.
* Once each piece has been made, the corresponding image is created and added to the interface.
* The images are also 150 pixels by 150 pixels.
  + Despite setting the background of the images to transparent, their background would always be grey, so there would be four rows in the middle of the board with the correct colours, and then the two rows at each end where the pieces are where there would be no board.
  + I fixed this by setting the background colour of the images to match the colour of the corresponding panel.
  + Therefore, every time a piece moves, its background colour is updated to match that of the panel it is moving to.

Other:

* List boxes:
  + The white boxes on either side of the interface are list boxes.
  + These list boxes are updated to show all of the pieces a player has taken.
  + For instance, if white takes a black pawn, “pawn” will be added to the list box on the white side of the interface.
* Labels:
  + There are two labels for each player’s timer.
  + These are made visible when the timer is set for two player games.
  + When it is white’s turn, their timer will change colour to red. When they finish their turn, the timer will update to green, and the same process will happen to the black timer on black’s turn.
  1. **File Structure and Organisation**
     1. **savedGame.txt**

When saving the game, the data is written to a .txt file called ‘savedGame’. The data is written in the following order:

* Pieces:
  + Piece type.
  + Colour.
  + x-ordinate.
  + y-ordinate.
* Timers:
  + whiteTimer.
  + blackTimer.
* Mode:
  + “One player” or “Two player”.

The timers are only written to the file if timeEnabled = true.

When loading the game, the data is read as follows:

* The first line is read.
* As the first line written to the file is a type of piece, an if statement is used to compare the line read to all six different piece types.
* Depending on the piece type read, a piece of that type is created.
* As the next line written to the file is the colour of that piece, the next line is read and the data in that line is used to set the colour.
* This repeats for the next two lines, where the co-ordinates of the piece were written.
* A picture box is created, and the piece and its picture are added to the board.
* This repeats until all the pieces have been read.
* The next line to be read will either be an integer or a string which says either “One player” or “Two player”. If the line is an integer, the white timer is set to that integer, and the black timer is set to the integer on the next line. If the next line is a string, the mode will be set to either one or two players.

### Technical Solution

**Table of Contents**

[**3.1** **Areas of interest** 45](#_Toc129185728)

[**3.2** **Code** 46](#_Toc129185729)

* 1. **Areas of interest**

|  |  |  |  |
| --- | --- | --- | --- |
| **Technique** | **Description** | **File** | **Lines** |
| Depth-first tree search, Recursion, and stacks in the minimax algorithm | The algorithm searches a tree of different moves that each player could make to find the best move for the computer to make. | AI.cs | 141 – 204 |
| Inheritance (OOP) | The Piece class is a parent class, and the Pawn, Knight, Rook, Bishop, King, and Queen classes are all subclasses. | Pawn.cs, Knight.cs, Rook.cs, Bishop.cs, King.cs, Queen.cs | 3 |
| Polymorphism (OOP) | The PossibleMoves method in the Piece class is overridden in all six subclasses. | Piece.cs, Pawn.cs, Knight.cs, Rook.cs, Bishop.cs, King.cs, Queen.cs | 15 in Piece.cs, 5 in the subclasses |
| Dynamic generation of objects based on complex user-defined use of OOP | All pieces are created dynamically through the CreatePieces algorithm, which calls individual algorithms for each class. | Form1.cs | 32 – 352 |
| Multi-dimensional arrays | Multi-dimensional arrays are used throughout the program to represent the board. An example is grid, in the Globals class. | Globals.cs | 7 |
| Writing to and reading from text files | Text files are used to save the state of a game so that users can come back to a game at a later date. | Form1.cs | 428 – 672 |
| Linear search | A linear search is used on a multi-dimensional array to find the position of a king on the board. | Globals.cs | 26 – 61 |
| **Coding style** | **Description** | **File** | **Lines** |
| Cohesive modules | Modules only do the task they are required to do. | Globals.cs, AI.cs, Form1.cs, Piece.cs and its subclasses | Globals.cs: 14 – 25, 26 – 61.  AI.cs: 13 – 41, 42 - 108, 109 – 140, 141 – 204 |
| Subroutines with common purpose grouped | All the AI subroutines are grouped together in the AI class, and all subroutines to create pieces, images, labels, etc., are grouped. | AI.cs, Form1.cs | AI.cs: 13 – 204, 205 - 257.  Form1.cs: 31 – 352, 353 – 400, 416 – 672, 673 – 868, 869 – 1054, 1055 - 1237 |
| Defensive programming | One example of defensive programming is null checks. The IsPositionEmpty algorithm is used as a null check throughout the program as there are many operations which require a position not to be null. | Globals.cs | 14 – 25 |
| Well-designed user interface | The user interface is clear and easy to use. | Shown in section 2.4. | N/A |
| Minimal use of global variables | Global variables and methods are only used where necessary and are grouped together in the Globals class. There are seven global variables and two global methods. | Globals.cs | N/A |

* 1. **Code**
     1. **Table of Contents**

[**3.2.2.1 Globals.cs** 46](#_Toc129190761)

[**3.2.2.2 AI.cs** 48](#_Toc129190762)

[**3.2.2.3 Piece.cs** 52](#_Toc129190763)

[**3.2.2.4 Pawn.cs** 54](#_Toc129190764)

[**3.2.2.5 Bishop.cs** 57](#_Toc129190765)

[**3.2.2.6 King.cs** 59](#_Toc129190766)

[**3.2.2.7 Knight.cs** 67](#_Toc129190767)

[**3.2.2.8 Queen.cs** 69](#_Toc129190768)

[**3.2.2.9 Rook.cs** 73](#_Toc129190769)

[**3.2.2.10 Form1.cs** 75](#_Toc129190770)

* + 1. **.cs Files**

|  |
| --- |
| **3.2.2.1 Globals.cs** |
| using System.Windows.Forms;  namespace ChessProject  {  public static class Globals  {  public static Piece[,] grid = new Piece[8, 8]; // Stores the location of pieces  public static Piece takenPiece; // Stores a removed piece  public static PictureBox[,] pieceImages = new PictureBox[8, 8]; // Stores the location of images  public static bool whiteTurn = true; // Stores whose turn it is: true if white's turn, false if black's turn  public static int kingX = 0; // Stores the x position of the king which has been fetched  public static int kingY = 0; // Stores the y position of the king which has been fetched  public static bool IsPositionEmpty(int x, int y, Piece[,] board)  {  if (board[x, y] == null)  {  return true;  }  else  {  return false;  }  }  public static void GetKing(string colour, Piece[,] board)  {  // Fetches the king of either white or black and sets the global variables kingX and kingY to the co-ordinates of the king  // These co-ordinates are used when checking for check and checkmate outside of the King class  bool kingFound = false;  do  {  for (int i = 0; i < 8; i++)  {  for (int j = 0; j < 8; j++)  {  if (!IsPositionEmpty(i, j, board))  {  if (board[i, j].pieceType == "king" && board[i, j].colour == colour)  {  kingX = i;  kingY = j;  kingFound = true;  }  }  if (kingFound)  {  break;  }  }  if (kingFound)  {  break;  }  }  } while (!kingFound);  }  public static int GetPieceCount(Piece[,] board)  {  int pieceCount = 0;  for (int i = 0; i < 8; i++)  {  for (int j = 0; j < 8; j++)  {  if (!IsPositionEmpty(i, j, board))  {  pieceCount++;  }  if (pieceCount == 32)  {  break;  }  }  if (pieceCount == 32)  {  break;  }  }  return pieceCount;  }  }  } |

|  |
| --- |
| **3.2.2.2 AI.cs** |
| using System;  using System.Collections.Generic;  namespace ChessProject  {  class AI  {  public static int searchDepth = 3; // Sets the default search depth. This is decremented during the minimax search  public static List<int> moveScores = new List<int>(); // Stores the scores of each move to determine the best move  public static List<Piece> movedPieces = new List<Piece>(); // Stores the pieces which make the moves in the search  public static int resultPos; // Stores the position of the result in movedPieces  public static Piece[,] DuplicateBoard(Piece[,] newBoard, Piece[,] oldBoard) // Duplicates a board to use in move generation or checking  {  int pieceCount = Globals.GetPieceCount(oldBoard);  int piecesDuplicated = 0;  for (int i = 0; i < 8; i++)  {  for (int j = 0; j < 8; j++)  {  if (!Globals.IsPositionEmpty(i, j, oldBoard))  {  newBoard[i, j] = oldBoard[i, j];  piecesDuplicated++;  }  if (pieceCount == piecesDuplicated)  {  break;  }  }  if (pieceCount == piecesDuplicated)  {  break;  }  }  return newBoard;  }  public static List<Piece[,]> GenerateMoves(Piece[,] board, int depth, bool whiteTurn) // Generates all possible moves for every piece of a specified colour  {  bool[,] possibleMoves = new bool[8, 8];  List<Piece[,]> moves = new List<Piece[,]>();  int pieceCount = Globals.GetPieceCount(board);  int piecesChecked = 0;  for (int i = 0; i < 8; i++)  {  for (int j = 0; j < 8; j++)  {  if (whiteTurn)  {  if (!Globals.IsPositionEmpty(i, j, board) && board[i, j].colour == "white")  {  possibleMoves = board[i, j].PossibleMoves(possibleMoves, board);  piecesChecked++;  }  }  else  {  if (!Globals.IsPositionEmpty(i, j, board) && board[i, j].colour == "black")  {  possibleMoves = board[i, j].PossibleMoves(possibleMoves, board);  piecesChecked++;  }  }  for (int k = 0; k < 8; k++)  {  for (int l = 0; l < 8; l++)  {  if (possibleMoves[k, l])  {  if (!Globals.IsPositionEmpty(i, j, board))  {  Piece[,] move = new Piece[8, 8];  move = DuplicateBoard(move, board);  AIUpdatePosition(move[i, j], i, j, k, l, move);  moves.Add(move);  board[i, j].newX = k;  board[i, j].newY = l;  if (depth == searchDepth && board[i, j].colour == "black")  {  movedPieces.Add(board[i, j]);  }  }  }  }  }  if (pieceCount == piecesChecked)  {  break;  }  }  if (pieceCount == piecesChecked)  {  break;  }  }  return moves;  }  public static int Evaluation(Piece[,] board, bool checkForWhite) // Calculates the balance of play  {  int evaluation = 0;  for (int i = 0; i < 8; i++)  {  for (int j = 0; j < 8; j++)  {  if (!Globals.IsPositionEmpty(i, j, board))  {  if (board[i, j].colour == "white")  {  evaluation += board[i, j].strength;  }  else  {  evaluation -= board[i, j].strength;  }  }  }  }  if (checkForWhite)  {  return evaluation;  }  else  {  return evaluation \* -1;  }  }  public static int Minimax(Piece[,] board, int depth, bool maximisingPlayer, bool whiteTurn, int alpha, int beta) // Searches through the possible moves to find the best one  {  // This code is taken from a stackoverflow user called foRei  // I have added alpha-beta pruning to their algorithm to optimise the search  // The code used is the solution at the bottom of this webpage:  // https://stackoverflow.com/questions/66082633/minimax-algorithm-with-chess-in-c-sharp-not-working-properly  if (depth == 0) // Base case, evaluates the board passed into the minimax algorithm and returns the result  {  int result = Evaluation(board, whiteTurn);  return result;  }  List<Piece[,]> moves = GenerateMoves(board, depth, whiteTurn);  if (maximisingPlayer)  {  int value = -1290;  foreach (Piece[,] move in moves)  {  int minimaxResult = Minimax(move, depth - 1, false, !whiteTurn, alpha, beta);  value = Math.Max(value, minimaxResult);  alpha = Math.Max(alpha, value);  if (beta <= alpha)  {  break; // No need to continue searching  }  if (depth == searchDepth)  {  moveScores.Add(minimaxResult); // If this is the first layer of the search, add the score to the list to access later  }  }  return value;  }  else  {  int value = 1290;  foreach (Piece[,] move in moves)  {  int minimaxResult = Minimax(move, depth - 1, true, !whiteTurn, alpha, beta);  value = Math.Min(value, minimaxResult);  beta = Math.Min(beta, value);  if (beta <= alpha)  {  break;  }  if (depth == searchDepth)  {  moveScores.Add(minimaxResult);  resultPos = moveScores.IndexOf(minimaxResult);  }  }  return value;  }  }  public static void AIRemovePiece(Piece piece, Piece[,] board)  {  board[piece.xGridPos, piece.yGridPos] = null;  }  public static void AIUpdatePosition(Piece piece, int currentX, int currentY, int newX, int newY, Piece[,] board)  {  if (piece != null)  {  if (Globals.IsPositionEmpty(newX, newY, board))  {  board[currentX, currentY] = null;  board[newX, newY] = piece;  }  else  {  AIRemovePiece(board[newX, newY], board);  board[currentX, currentY] = null;  board[newX, newY] = piece;  }  if (piece.pieceType == "pawn")  {  Pawn pawnConvert = (Pawn)piece;  if (pawnConvert.colour == "white")  {  if (pawnConvert.yGridPos == 0)  {  AIQueenConvert(pawnConvert, board);  }  }  else  {  if (pawnConvert.yGridPos == 7)  {  AIQueenConvert(pawnConvert, board);  }  }  }  }  }  public static void AIQueenConvert(Pawn pawn, Piece[,] board)  {  AIRemovePiece(pawn, board);  Queen newQueen = new Queen();  newQueen.colour = pawn.colour;  newQueen.pieceType = "queen";  newQueen.xGridPos = pawn.xGridPos;  newQueen.yGridPos = pawn.yGridPos;  }  public static void Turn()  {  if (moveScores.Count > 0)  {  moveScores.Clear();  }  if (movedPieces.Count > 0)  {  movedPieces.Clear();  }  Piece[,] board = new Piece[8, 8];  board = DuplicateBoard(board, Globals.grid);  Minimax(board, searchDepth, false, false, -1290, 1290);  Piece movingPiece = movedPieces[resultPos];  bool[,] possibleMoves = new bool[8, 8];  possibleMoves = movingPiece.PossibleMoves(possibleMoves, Globals.grid);  if (possibleMoves[movingPiece.newX, movingPiece.newY])  {  Globals.grid[movingPiece.xGridPos, movingPiece.yGridPos].UpdatePosition(movingPiece, movingPiece.xGridPos, movingPiece.yGridPos, movingPiece.newX, movingPiece.newY);  }  }  }  } |

|  |
| --- |
| **3.2.2.3 Piece.cs** |
| using System.Drawing;  namespace ChessProject  {  public class Piece  {  public string pieceType; // Stores the type of piece  public string colour; // Stores the colour of the piece  public int xGridPos; // Stores the x-ordinate of the piece (0 - 7)  public int yGridPos; // Stores the y-ordinate of the piece (0 - 7)  public int newX; // Stores the new x-ordinate when moving  public int newY; // Stores the new y-ordinate when moving  public int strength; // Used to determine the relative strength of each piece type    public virtual bool[,] PossibleMoves(bool[,] possibleMoves, Piece[,] board)  {  for (int i = 0; i < 8; i++)  {  for (int j = 0; j < 8; j++)  {  possibleMoves[i, j] = false;  }  }  return possibleMoves;  }  public bool DifferentColour(int x, int y, Piece[,] board)  {  if (colour != board[x, y].colour)  {  return true;  }  else  {  return false;  }  }  public void AddPiece(Piece piece, Piece[,] board)  {  board[xGridPos, yGridPos] = piece;  }  public void RemovePiece(Piece piece)  {  Globals.grid[piece.xGridPos, piece.yGridPos] = null;  Globals.pieceImages[piece.xGridPos, piece.yGridPos].Visible = false;  Globals.pieceImages[piece.xGridPos, piece.yGridPos] = null;  Globals.takenPiece = piece;  }  public void UpdatePosition(Piece piece, int currentX, int currentY, int newX, int newY)  {  if (Globals.IsPositionEmpty(newX, newY, Globals.grid))  {  Globals.grid[currentX, currentY] = null;  Globals.grid[newX, newY] = piece;  }  else  {  RemovePiece(Globals.grid[newX, newY]);  Globals.grid[currentX, currentY] = null;  Globals.grid[newX, newY] = piece;  }  Globals.pieceImages[newX, newY] = Globals.pieceImages[currentX, currentY];  Globals.pieceImages[currentX, currentY] = null;  Globals.pieceImages[newX, newY].Location = new Point(100 + (150 \* newX), 100 + (150 \* newY));    if ((newX == 0 || newX == 2 || newX == 4 || newX == 6) && (newY == 0 || newY == 2 || newY == 4 || newY == 6))  {  Globals.pieceImages[newX, newY].BackColor = Color.LightYellow;  }  else if ((newX == 1 || newX == 3 || newX == 5 || newX == 7) && (newY == 1 || newY == 3 || newY == 5 || newY == 7))  {  Globals.pieceImages[newX, newY].BackColor = Color.LightYellow;  }  else  {  Globals.pieceImages[newX, newY].BackColor = Color.Brown;  }  piece.xGridPos = newX;  piece.yGridPos = newY;  }  }  } |

|  |
| --- |
| **3.2.2.4 Pawn.cs** |
| namespace ChessProject  {  class Pawn : Piece  {  public override bool[,] PossibleMoves(bool[,] possibleMoves, Piece[,] board)  {  King king = new King();  Globals.GetKing(colour, board);  king.xGridPos = Globals.kingX;  king.yGridPos = Globals.kingY;  king = (King)board[Globals.kingX, Globals.kingY];  if (colour == "white")  {  if (yGridPos > 0)  {  if (Globals.IsPositionEmpty(xGridPos, yGridPos - 1, board)) // up one  {  Piece[,] duplicateBoard = new Piece[8, 8];  duplicateBoard = AI.DuplicateBoard(duplicateBoard, board);  AI.AIUpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos, yGridPos - 1, duplicateBoard);  if (!king.Check(king.xGridPos, king.yGridPos, duplicateBoard))  {  possibleMoves[xGridPos, yGridPos - 1] = true;  }  if (yGridPos == 6)  {  AI.AIUpdatePosition(duplicateBoard[xGridPos, yGridPos - 1], xGridPos, yGridPos, xGridPos, yGridPos - 2, duplicateBoard);  if (Globals.IsPositionEmpty(xGridPos, yGridPos - 2, board) && !king.Check(king.xGridPos, king.yGridPos, duplicateBoard)) // up two  {  possibleMoves[xGridPos, yGridPos - 2] = true;  }  }  }  }    if (xGridPos > 0 && yGridPos > 0)  {  if (!Globals.IsPositionEmpty(xGridPos - 1, yGridPos - 1, board)) // take left  {  if (DifferentColour(xGridPos - 1, yGridPos - 1, board))  {  Piece[,] duplicateBoard = new Piece[8, 8];  duplicateBoard = AI.DuplicateBoard(duplicateBoard, board);  AI.AIUpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos - 1, yGridPos - 1, duplicateBoard);  if (!king.Check(king.xGridPos, king.yGridPos, duplicateBoard))  {  possibleMoves[xGridPos - 1, yGridPos - 1] = true;  }  }  }  }  if (xGridPos < 7 && yGridPos > 0)  {  if (!Globals.IsPositionEmpty(xGridPos + 1, yGridPos - 1, board)) // take right  {  if (DifferentColour(xGridPos + 1, yGridPos - 1, board))  {  Piece[,] duplicateBoard = new Piece[8, 8];  duplicateBoard = AI.DuplicateBoard(duplicateBoard, board);  AI.AIUpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos + 1, yGridPos + 1, duplicateBoard);  if (!king.Check(king.xGridPos, king.yGridPos, duplicateBoard))  {  possibleMoves[xGridPos + 1, yGridPos - 1] = true;  }  }  }  }  }  else if (colour == "black")  {  if (yGridPos < 7)  {  if (Globals.IsPositionEmpty(xGridPos, yGridPos + 1, board) && yGridPos < 7) // down one  {  Piece[,] duplicateBoard = new Piece[8, 8];  duplicateBoard = AI.DuplicateBoard(duplicateBoard, board);  AI.AIUpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos, yGridPos + 1, duplicateBoard);  if (!king.Check(king.xGridPos, king.yGridPos, duplicateBoard))  {  possibleMoves[xGridPos, yGridPos + 1] = true;  }  if (yGridPos == 1)  {  AI.AIUpdatePosition(duplicateBoard[xGridPos, yGridPos + 1], xGridPos, yGridPos, xGridPos, yGridPos + 2, duplicateBoard);  if (Globals.IsPositionEmpty(xGridPos, yGridPos + 2, board) && !king.Check(king.xGridPos, king.yGridPos, duplicateBoard)) // down two  {  possibleMoves[xGridPos, yGridPos + 2] = true;  }  }  }  }    if (xGridPos > 0 && yGridPos < 7)  {  if (!Globals.IsPositionEmpty(xGridPos - 1, yGridPos + 1, board)) // take left  {  if (DifferentColour(xGridPos - 1, yGridPos + 1, board))  {  Piece[,] duplicateBoard = new Piece[8, 8];  duplicateBoard = AI.DuplicateBoard(duplicateBoard, board);  AI.AIUpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos - 1, yGridPos + 1, duplicateBoard);  if (!king.Check(king.xGridPos, king.yGridPos, duplicateBoard))  {  possibleMoves[xGridPos - 1, yGridPos + 1] = true;  }  }  }  }  if (xGridPos < 7 && yGridPos < 7)  {  if (!Globals.IsPositionEmpty(xGridPos + 1, yGridPos + 1, board)) // take right  {  if (DifferentColour(xGridPos + 1, yGridPos + 1, board))  {  Piece[,] duplicateBoard = new Piece[8, 8];  duplicateBoard = AI.DuplicateBoard(duplicateBoard, board);  AI.AIUpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos + 1, yGridPos + 1, duplicateBoard);    if (!king.Check(king.xGridPos, king.yGridPos, duplicateBoard))  {  possibleMoves[xGridPos + 1, yGridPos + 1] = true;  }  }  }  }  }  return possibleMoves;  }  }  } |

|  |
| --- |
| **3.2.2.5 Bishop.cs** |
| namespace ChessProject  {  class Bishop : Piece  {  public override bool[,] PossibleMoves(bool[,] possibleMoves, Piece[,] board)  {  King king = new King();  Globals.GetKing(colour, board);  king.xGridPos = Globals.kingX;  king.yGridPos = Globals.kingY;  king = (King)board[Globals.kingX, Globals.kingY];  for (int i = 1; i < 8; i++)  {  if (xGridPos + i <= 7 && yGridPos + i <= 7) // diagonal down and right  {  if (Globals.IsPositionEmpty(xGridPos + i, yGridPos + i, board) || (!Globals.IsPositionEmpty(xGridPos + i, yGridPos + i, board) && DifferentColour(xGridPos + i, yGridPos + i, board)))  {  Piece[,] duplicateBoard = new Piece[8, 8];  duplicateBoard = AI.DuplicateBoard(duplicateBoard, board);  AI.AIUpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos + i, yGridPos + i, duplicateBoard);  if (!king.Check(king.xGridPos, king.yGridPos, duplicateBoard))  {  possibleMoves[xGridPos + i, yGridPos + i] = true;  }  if (!Globals.IsPositionEmpty(xGridPos + i, yGridPos + i, board) && possibleMoves[xGridPos + i, yGridPos + i])  {  break;  }  }  else  {  break;  }  }  else  {  break;  }  }  for (int i = 1; i < 8; i++)  {  if (xGridPos + i <= 7 && yGridPos - i >= 0) // diagonal up and right  {  if (Globals.IsPositionEmpty(xGridPos + i, yGridPos - i, board) || (!Globals.IsPositionEmpty(xGridPos + i, yGridPos - i, board) && DifferentColour(xGridPos + i, yGridPos - i, board)))  {  Piece[,] duplicateBoard = new Piece[8, 8];  duplicateBoard = AI.DuplicateBoard(duplicateBoard, board);  AI.AIUpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos + i, yGridPos - i, duplicateBoard);  if (!king.Check(king.xGridPos, king.yGridPos, duplicateBoard))  {  possibleMoves[xGridPos + i, yGridPos - i] = true;  }  if (!Globals.IsPositionEmpty(xGridPos + i, yGridPos - i, board) && possibleMoves[xGridPos + i, yGridPos - i])  {  break;  }  }  else  {  break;  }  }  else  {  break;  }  }  for (int i = 1; i < 8; i++)  {  if (xGridPos - i >= 0 && yGridPos - i >= 0) // diagonal up and left  {  if (Globals.IsPositionEmpty(xGridPos - i, yGridPos - i, board) || (!Globals.IsPositionEmpty(xGridPos - i, yGridPos - i, board) && DifferentColour(xGridPos - i, yGridPos - i, board)))  {  Piece[,] duplicateBoard = new Piece[8, 8];  duplicateBoard = AI.DuplicateBoard(duplicateBoard, board);  AI.AIUpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos - i, yGridPos - i, duplicateBoard);  if (!king.Check(king.xGridPos, king.yGridPos, duplicateBoard))  {  possibleMoves[xGridPos - i, yGridPos - i] = true;  }  if (!Globals.IsPositionEmpty(xGridPos - i, yGridPos - i, board) && possibleMoves[xGridPos - i, yGridPos - i])  {  break;  }  }  else  {  break;  }  }  else  {  break;  }  }  for (int i = 1; i < 8; i++)  {  if (xGridPos - i >= 0 && yGridPos + i <= 7) // diagonal down and left  {  if (Globals.IsPositionEmpty(xGridPos - i, yGridPos + i, board) || (!Globals.IsPositionEmpty(xGridPos - i, yGridPos + i, board) && DifferentColour(xGridPos - i, yGridPos + i, board)))  {  Piece[,] duplicateBoard = new Piece[8, 8];  duplicateBoard = AI.DuplicateBoard(duplicateBoard, board);  AI.AIUpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos - i, yGridPos + i, duplicateBoard);  if (!king.Check(king.xGridPos, king.yGridPos, duplicateBoard))  {  possibleMoves[xGridPos - i, yGridPos + i] = true;  }  if (!Globals.IsPositionEmpty(xGridPos - i, yGridPos + i, board) && possibleMoves[xGridPos - i, yGridPos + i])  {  break;  }  }  else  {  break;  }  }  else  {  break;  }  }  return possibleMoves;  }  }  } |

|  |
| --- |
| **3.2.2.6 King.cs** |
| namespace ChessProject  {  public class King : Piece  {  public override bool[,] PossibleMoves(bool[,] possibleMoves, Piece[,] board)  {  if (yGridPos != 7) // down one  {  if ((Globals.IsPositionEmpty(xGridPos, yGridPos + 1, board) || (!Globals.IsPositionEmpty(xGridPos, yGridPos + 1, board) && DifferentColour(xGridPos, yGridPos + 1, board))) && !Check(xGridPos, yGridPos + 1, board))  {  possibleMoves[xGridPos, yGridPos + 1] = true;  }  }  if (xGridPos != 7 && yGridPos != 7) // right one, down one  {  if ((Globals.IsPositionEmpty(xGridPos + 1, yGridPos + 1, board) || (!Globals.IsPositionEmpty(xGridPos + 1, yGridPos + 1, board) && DifferentColour(xGridPos + 1, yGridPos + 1, board))) && !Check(xGridPos + 1, yGridPos + 1, board))  {  possibleMoves[xGridPos + 1, yGridPos + 1] = true;  }  }  if (xGridPos != 7) // right one  {  if ((Globals.IsPositionEmpty(xGridPos + 1, yGridPos, board) || (!Globals.IsPositionEmpty(xGridPos + 1, yGridPos, board) && DifferentColour(xGridPos + 1, yGridPos, board))) && !Check(xGridPos + 1, yGridPos, board))  {  possibleMoves[xGridPos + 1, yGridPos] = true;  }  }  if (xGridPos != 7 && yGridPos != 0) // right one, up one  {  if ((Globals.IsPositionEmpty(xGridPos + 1, yGridPos - 1, board) || (!Globals.IsPositionEmpty(xGridPos + 1, yGridPos - 1, board) && DifferentColour(xGridPos + 1, yGridPos - 1, board))) && !Check(xGridPos + 1, yGridPos - 1, board))  {  possibleMoves[xGridPos + 1, yGridPos - 1] = true;  }  }  if (yGridPos != 0) // up one  {  if ((Globals.IsPositionEmpty(xGridPos, yGridPos - 1, board) || (!Globals.IsPositionEmpty(xGridPos, yGridPos - 1, board) && DifferentColour(xGridPos, yGridPos - 1, board))) && !Check(xGridPos, yGridPos - 1, board))  {  possibleMoves[xGridPos, yGridPos - 1] = true;  }  }  if (xGridPos != 0 && yGridPos != 0) // left one, up one  {  if ((Globals.IsPositionEmpty(xGridPos - 1, yGridPos - 1, board) || (!Globals.IsPositionEmpty(xGridPos - 1, yGridPos - 1, board) && DifferentColour(xGridPos - 1, yGridPos - 1, board))) && !Check(xGridPos - 1, yGridPos - 1, board))  {  possibleMoves[xGridPos - 1, yGridPos - 1] = true;  }  }  if (xGridPos != 0) // left one  {  if ((Globals.IsPositionEmpty(xGridPos - 1, yGridPos, board) || (!Globals.IsPositionEmpty(xGridPos - 1, yGridPos, board) && DifferentColour(xGridPos - 1, yGridPos, board))) && !Check(xGridPos - 1, yGridPos, board))  {  possibleMoves[xGridPos - 1, yGridPos] = true;  }  }  if (xGridPos != 0 && yGridPos != 7) // left one, down one  {  if ((Globals.IsPositionEmpty(xGridPos - 1, yGridPos + 1, board) || (!Globals.IsPositionEmpty(xGridPos - 1, yGridPos + 1, board) && DifferentColour(xGridPos - 1, yGridPos + 1, board))) && !Check(xGridPos - 1, yGridPos + 1, board))  {  possibleMoves[xGridPos - 1, yGridPos + 1] = true;  }  }    return possibleMoves;  }  public bool Check(int x, int y, Piece[,] board)  {  //diagonals  for (int i = x + 1; i < 8; i++) //down right  {  bool breakLoop = false;  for (int j = y + 1; j < 8; j++)  {  if (i - x == j - y)  {  if (!Globals.IsPositionEmpty(i, j, board))  {  if (DifferentColour(i, j, board))  {  if (board[i, j].pieceType == "bishop" || board[i, j].pieceType == "queen" || (board[i, j].pieceType == "pawn" && i == x + 1 && j == y + 1 && board[i, j].colour == "black"))  {  return true;  }  else  {  breakLoop = true;  break;  }  }  else  {  breakLoop = true;  break;  }  }  }  }  if (breakLoop)  {  break;  }  }  for (int i = x - 1; i > -1; i--) // up left  {  bool breakLoop = false;  for (int j = y - 1; j > -1; j--)  {  if (i - x == j - y)  {  if (!Globals.IsPositionEmpty(i, j, board))  {  if (DifferentColour(i, j, board))  {  if (board[i, j].pieceType == "bishop" || board[i, j].pieceType == "queen" || (board[i, j].pieceType == "pawn" && i == x - 1 && j == y - 1 && board[i, j].colour == "white"))  {  return true;  }  else  {  breakLoop = true;  break;  }  }  else  {  breakLoop = true;  break;  }  }  }  }  if (breakLoop)  {  break;  }  }  for (int i = x + 1; i < 8; i++) //down left  {  bool breakLoop = false;  for (int j = y - 1; j > -1; j--)  {  if (i - x == -(j - y))  {  if (!Globals.IsPositionEmpty(i, j, board))  {  if (DifferentColour(i, j, board))  {  if (board[i, j].pieceType == "bishop" || board[i, j].pieceType == "queen" || (board[i, j].pieceType == "pawn" && i == x + 1 && j == x - 1 && board[i, j].colour == "black"))  {  return true;  }  else  {  breakLoop = true;  break;  }  }  else  {  breakLoop = true;  break;  }  }  }  }  if (breakLoop)  {  break;  }  }  for (int i = x - 1; i > -1; i--) //up right  {  bool breakLoop = false;  for (int j = y + 1; j < 8; j++)  {  if (-(i - x) == j - y)  {  if (!Globals.IsPositionEmpty(i, j, board))  {  if (DifferentColour(i, j, board))  {  if (board[i, j].pieceType == "bishop" || board[i, j].pieceType == "queen" || (board[i, j].pieceType == "pawn" && i == x - 1 && j == y + 1 && board[i, j].colour == "white"))  {  return true;  }  else  {  breakLoop = true;  break;  }  }  else  {  breakLoop = true;  break;  }  }  }  }  if (breakLoop)  {  break;  }  }  //horizontals  for (int i = x + 1; i < 8; i++) // right  {  if (!Globals.IsPositionEmpty(i, y, board))  {  if (DifferentColour(i, y, board))  {  if (board[i, y].pieceType == "rook" || board[i, y].pieceType == "queen")  {  return true;  }  else  {  break;  }  }  else  {  break;  }  }  }  for (int i = x - 1; i > -1; i--) // left  {  if (!Globals.IsPositionEmpty(i, y, board))  {  if (DifferentColour(i, y, board))  {  if (board[i, y].pieceType == "rook" || board[i, y].pieceType == "queen")  {  return true;  }  else  {  break;  }  }  else  {  break;  }  }  }  //verticals  for (int i = y + 1; i < 8; i++) // down  {  if (!Globals.IsPositionEmpty(x, i, board))  {  if (DifferentColour(x, i, board))  {  if (board[x, i].pieceType == "rook" || board[x, i].pieceType == "queen")  {  return true;  }  else  {  break;  }  }  else  {  break;  }  }  }  for (int i = y - 1; i > -1; i--) // up  {  if (!Globals.IsPositionEmpty(x, i, board))  {  if (DifferentColour(x, i, board))  {  if (board[x, i].pieceType == "rook" || board[x, i].pieceType == "queen")  {  return true;  }  else  {  break;  }  }  else  {  break;  }  }  }  //knights  if (x < 7 && y < 6)  {  if (!Globals.IsPositionEmpty(x + 1, y + 2, board))  {  if (DifferentColour(x + 1, y + 2, board))  {  if (board[x + 1, y + 2].pieceType == "knight")  {  return true;  }  }  }  }  if (x < 7 && y > 1)  {  if (!Globals.IsPositionEmpty(x + 1, y - 2, board))  {  if (DifferentColour(x + 1, y - 2, board))  {  if (board[x + 1, y - 2].pieceType == "knight")  {  return true;  }  }  }  }  if (x > 0 && y < 6)  {  if (!Globals.IsPositionEmpty(x - 1, y + 2, board))  {  if (DifferentColour(x - 1, y + 2, board))  {  if (board[x - 1, y + 2].pieceType == "knight")  {  return true;  }  }  }  }  if (x > 0 && y > 1)  {  if (!Globals.IsPositionEmpty(x - 1, y - 2, board))  {  if (DifferentColour(x - 1, y - 2, board))  {  if (board[x - 1, y - 2].pieceType == "knight")  {  return true;  }  }  }  }  if (x > 1 && y < 7)  {  if (!Globals.IsPositionEmpty(x - 2, y + 1, board))  {  if (DifferentColour(x - 2, y + 1, board))  {  if (board[x - 2, y + 1].pieceType == "knight")  {  return true;  }  }  }  }  if (x > 1 && y > 0)  {  if (!Globals.IsPositionEmpty(x - 2, y - 1, board))  {  if (DifferentColour(x - 2, y - 1, board))  {  if (board[x - 2, y - 1].pieceType == "knight")  {  return true;  }  }  }  }  if (x < 6 && y > 0)  {  if (!Globals.IsPositionEmpty(x + 2, y - 1, board))  {  if (DifferentColour(x + 2, y - 1, board))  {  if (board[x + 2, y - 1].pieceType == "knight")  {  return true;  }  }  }  }  if (x < 6 && y < 7)  {  if (!Globals.IsPositionEmpty(x + 2, y + 1, board))  {  if (DifferentColour(x + 2, y + 1, board))  {  if (board[x + 2, y + 1].pieceType == "knight")  {  return true;  }  }  }  }  return false;  }  public bool Checkmate(int x, int y, Piece[,] board)  {  if (!Check(x, y, board))  {  return false; // if the king is not in check it is impossible for it to be checkmate  }  for (int i = 0; i < 8; i++)  {  for (int j = 0; j < 8; j++)  {  if (!Globals.IsPositionEmpty(i, j, board))  {  if (!DifferentColour(i, j, board))  {  bool[,] possibleMoves = new bool[8, 8];  possibleMoves = board[i, j].PossibleMoves(possibleMoves, board);  for (int k = 0; k < 8; k++)  {  for (int l = 0; l < 8; l++)  {  if (possibleMoves[k, l])  {  return false;  }  }  }  }  }  }  }    return true; // no pieces can move legally and the king is in check therefore it is checkmate  }  }  } |

|  |
| --- |
| **3.2.2.7 Knight.cs** |
| namespace ChessProject  {  class Knight : Piece  {  public override bool[,] PossibleMoves(bool[,] possibleMoves, Piece[,] board)  {  King king = new King();  Globals.GetKing(colour, board);  king.xGridPos = Globals.kingX;  king.yGridPos = Globals.kingY;  king = (King)board[Globals.kingX, Globals.kingY];  if (xGridPos != 7 && yGridPos < 6) // right one, down two  {  if (Globals.IsPositionEmpty(xGridPos + 1, yGridPos + 2, board) || (!Globals.IsPositionEmpty(xGridPos + 1, yGridPos + 2, board) && DifferentColour(xGridPos + 1, yGridPos + 2, board)))  {  Piece[,] duplicateBoard = new Piece[8, 8];  duplicateBoard = AI.DuplicateBoard(duplicateBoard, board);  AI.AIUpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos + 1, yGridPos + 2, duplicateBoard);  if (!king.Check(king.xGridPos, king.yGridPos, duplicateBoard))  {  possibleMoves[xGridPos + 1, yGridPos + 2] = true;  }  }  }  if (xGridPos < 6 && yGridPos != 7) // right two, down one  {  if (Globals.IsPositionEmpty(xGridPos + 2, yGridPos + 1, board) || (!Globals.IsPositionEmpty(xGridPos + 2, yGridPos + 1, board) && DifferentColour(xGridPos + 2, yGridPos + 1, board)))  {  Piece[,] duplicateBoard = new Piece[8, 8];  duplicateBoard = AI.DuplicateBoard(duplicateBoard, board);  AI.AIUpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos + 2, yGridPos + 1, duplicateBoard);  if (!king.Check(king.xGridPos, king.yGridPos, duplicateBoard))  {  possibleMoves[xGridPos + 2, yGridPos + 1] = true;  }  }  }  if (xGridPos < 6 && yGridPos != 0) // right two, up one  {  if (Globals.IsPositionEmpty(xGridPos + 2, yGridPos - 1, board) || (!Globals.IsPositionEmpty(xGridPos + 2, yGridPos - 1, board) && DifferentColour(xGridPos + 2, yGridPos - 1, board)))  {  Piece[,] duplicateBoard = new Piece[8, 8];  duplicateBoard = AI.DuplicateBoard(duplicateBoard, board);  AI.AIUpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos + 2, yGridPos - 1, duplicateBoard);  if (!king.Check(king.xGridPos, king.yGridPos, duplicateBoard))  {  possibleMoves[xGridPos + 2, yGridPos - 1] = true;  }  }  }  if (xGridPos != 7 && yGridPos > 1) // right one, up two  {  if (Globals.IsPositionEmpty(xGridPos + 1, yGridPos - 2, board) || (!Globals.IsPositionEmpty(xGridPos + 1, yGridPos - 2, board) && DifferentColour(xGridPos + 1, yGridPos - 2, board)))  {  Piece[,] duplicateBoard = new Piece[8, 8];  duplicateBoard = AI.DuplicateBoard(duplicateBoard, board);  AI.AIUpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos + 1, yGridPos - 2, duplicateBoard);  if (!king.Check(king.xGridPos, king.yGridPos, duplicateBoard))  {  possibleMoves[xGridPos + 1, yGridPos - 2] = true;  }  }  }  if (xGridPos != 0 && yGridPos > 1) // left one, up two  {  if (Globals.IsPositionEmpty(xGridPos - 1, yGridPos - 2, board) || (!Globals.IsPositionEmpty(xGridPos - 1, yGridPos - 2, board) && DifferentColour(xGridPos - 1, yGridPos - 2, board)))  {  Piece[,] duplicateBoard = new Piece[8, 8];  duplicateBoard = AI.DuplicateBoard(duplicateBoard, board);  AI.AIUpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos - 1, yGridPos - 2, duplicateBoard);  if (!king.Check(king.xGridPos, king.yGridPos, duplicateBoard))  {  possibleMoves[xGridPos - 1, yGridPos - 2] = true;  }  }  }  if (xGridPos > 1 && yGridPos != 0) // left two, up one  {  if (Globals.IsPositionEmpty(xGridPos - 2, yGridPos - 1, board) || (!Globals.IsPositionEmpty(xGridPos - 2, yGridPos - 1, board) && DifferentColour(xGridPos - 2, yGridPos - 1, board)))  {  Piece[,] duplicateBoard = new Piece[8, 8];  duplicateBoard = AI.DuplicateBoard(duplicateBoard, board);  AI.AIUpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos - 2, yGridPos - 1, duplicateBoard);  if (!king.Check(king.xGridPos, king.yGridPos, duplicateBoard))  {  possibleMoves[xGridPos - 2, yGridPos - 1] = true;  }  }  }  if (xGridPos > 1 && yGridPos != 7) // left two, down one  {  if (Globals.IsPositionEmpty(xGridPos - 2, yGridPos + 1, board) || (!Globals.IsPositionEmpty(xGridPos - 2, yGridPos + 1, board) && DifferentColour(xGridPos - 2, yGridPos + 1, board)))  {  Piece[,] duplicateBoard = new Piece[8, 8];  duplicateBoard = AI.DuplicateBoard(duplicateBoard, board);  AI.AIUpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos - 2, yGridPos + 1, duplicateBoard);  if (!king.Check(king.xGridPos, king.yGridPos, duplicateBoard))  {  possibleMoves[xGridPos - 2, yGridPos + 1] = true;  }  }  }  if (xGridPos != 0 && yGridPos < 6) // left one, down two  {  if (Globals.IsPositionEmpty(xGridPos - 1, yGridPos + 2, board) || (!Globals.IsPositionEmpty(xGridPos - 1, yGridPos + 2, board) && DifferentColour(xGridPos - 1, yGridPos + 2, board)))  {  Piece[,] duplicateBoard = new Piece[8, 8];  duplicateBoard = AI.DuplicateBoard(duplicateBoard, board);  AI.AIUpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos - 1, yGridPos + 2, duplicateBoard);  if (!king.Check(king.xGridPos, king.yGridPos, duplicateBoard))  {  possibleMoves[xGridPos - 1, yGridPos + 2] = true;  }  }  }  return possibleMoves;  }  }  } |

|  |
| --- |
| **3.2.2.8 Queen.cs** |
| namespace ChessProject  {  class Queen : Piece  {  public override bool[,] PossibleMoves(bool[,] possibleMoves, Piece[,] board)  {  King king = new King();  Globals.GetKing(colour, board);  king.xGridPos = Globals.kingX;  king.yGridPos = Globals.kingY;  king = (King)board[Globals.kingX, Globals.kingY];  for (int i = 1; i < 8; i++) // diagonals  {  if (xGridPos + i <= 7 && yGridPos + i <= 7) // diagonal down and right  {  if (Globals.IsPositionEmpty(xGridPos + i, yGridPos + i, board) || (!Globals.IsPositionEmpty(xGridPos + i, yGridPos + i, board) && DifferentColour(xGridPos + i, yGridPos + i, board)))  {  Piece[,] duplicateBoard = new Piece[8, 8];  duplicateBoard = AI.DuplicateBoard(duplicateBoard, board);  AI.AIUpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos + i, yGridPos + i, duplicateBoard);  if (!king.Check(king.xGridPos, king.yGridPos, duplicateBoard))  {  possibleMoves[xGridPos + i, yGridPos + i] = true;  }  if (!Globals.IsPositionEmpty(xGridPos + i, yGridPos + i, board) && possibleMoves[xGridPos + i, yGridPos + i])  {  break;  }  }  else  {  break;  }  }  else  {  break;  }  }  for (int i = 1; i < 8; i++)  {  if (xGridPos + i <= 7 && yGridPos - i >= 0) // diagonal up and right  {  if (Globals.IsPositionEmpty(xGridPos + i, yGridPos - i, board) || (!Globals.IsPositionEmpty(xGridPos + i, yGridPos - i, board) && DifferentColour(xGridPos + i, yGridPos - i, board)))  {  Piece[,] duplicateBoard = new Piece[8, 8];  duplicateBoard = AI.DuplicateBoard(duplicateBoard, board);  AI.AIUpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos + i, yGridPos - i, duplicateBoard);  if (!king.Check(king.xGridPos, king.yGridPos, duplicateBoard))  {  possibleMoves[xGridPos + i, yGridPos - i] = true;  }  if (!Globals.IsPositionEmpty(xGridPos + i, yGridPos - i, board) && possibleMoves[xGridPos + i, yGridPos - i])  {  break;  }  }  else  {  break;  }  }  else  {  break;  }  }  for (int i = 1; i < 8; i++)  {  if (xGridPos - i >= 0 && yGridPos - i >= 0) // diagonal up and left  {  if (Globals.IsPositionEmpty(xGridPos - i, yGridPos - i, board) || (!Globals.IsPositionEmpty(xGridPos - i, yGridPos - i, board) && DifferentColour(xGridPos - i, yGridPos - i, board)))  {  Piece[,] duplicateBoard = new Piece[8, 8];  duplicateBoard = AI.DuplicateBoard(duplicateBoard, board);  AI.AIUpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos - i, yGridPos - i, duplicateBoard);  if (!king.Check(king.xGridPos, king.yGridPos, duplicateBoard))  {  possibleMoves[xGridPos - i, yGridPos - i] = true;  }  if (!Globals.IsPositionEmpty(xGridPos - i, yGridPos - i, board) && possibleMoves[xGridPos - i, yGridPos - i])  {  break;  }  }  else  {  break;  }  }  else  {  break;  }  }  for (int i = 1; i < 8; i++)  {  if (xGridPos - i >= 0 && yGridPos + i <= 7) // diagonal up and right  {  if (Globals.IsPositionEmpty(xGridPos - i, yGridPos + i, board) || (!Globals.IsPositionEmpty(xGridPos - i, yGridPos + i, board) && DifferentColour(xGridPos - i, yGridPos + i, board)))  {  Piece[,] duplicateBoard = new Piece[8, 8];  duplicateBoard = AI.DuplicateBoard(duplicateBoard, board);  AI.AIUpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos - i, yGridPos + i, duplicateBoard);  if (!king.Check(king.xGridPos, king.yGridPos, duplicateBoard))  {  possibleMoves[xGridPos - i, yGridPos + i] = true;  }  if (!Globals.IsPositionEmpty(xGridPos -i, yGridPos + i, board) && possibleMoves[xGridPos - i, yGridPos + i])  {  break;  }  }  else  {  break;  }  }  else  {  break;  }  }  for (int i = 1; i < 8; i++) // horizontals  {  if (xGridPos - i >= 0) // left  {  if (Globals.IsPositionEmpty(xGridPos - i, yGridPos, board) || (!Globals.IsPositionEmpty(xGridPos - i, yGridPos, board) && DifferentColour(xGridPos - i, yGridPos, board)))  {  Piece[,] duplicateBoard = new Piece[8, 8];  duplicateBoard = AI.DuplicateBoard(duplicateBoard, board);  AI.AIUpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos - i, yGridPos, duplicateBoard);  if (!king.Check(king.xGridPos, king.yGridPos, duplicateBoard))  {  possibleMoves[xGridPos - i, yGridPos] = true;  }  if (!Globals.IsPositionEmpty(xGridPos - i, yGridPos, board) && possibleMoves[xGridPos - i, yGridPos])  {  break;  }  }  else  {  break;  }  }  else  {  break;  }  }  for (int i = 1; i < 8; i++)  {  if (xGridPos + i <= 7) // right  {  if (Globals.IsPositionEmpty(xGridPos + i, yGridPos, board) || (!Globals.IsPositionEmpty(xGridPos + i, yGridPos, board) && DifferentColour(xGridPos + i, yGridPos, board)))  {  Piece[,] duplicateBoard = new Piece[8, 8];  duplicateBoard = AI.DuplicateBoard(duplicateBoard, board);  AI.AIUpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos + i, yGridPos, duplicateBoard);  if (!king.Check(king.xGridPos, king.yGridPos, duplicateBoard))  {  possibleMoves[xGridPos + i, yGridPos] = true;  }  if (!Globals.IsPositionEmpty(xGridPos + i, yGridPos, board) && possibleMoves[xGridPos + i, yGridPos])  {  break;  }  }  else  {  break;  }  }  else  {  break;  }  }  for (int i = 1; i < 8; i++) // verticals  {  if (yGridPos - i >= 0) // up  {  if (Globals.IsPositionEmpty(xGridPos, yGridPos - i, board) || (!Globals.IsPositionEmpty(xGridPos, yGridPos - i, board) && DifferentColour(xGridPos, yGridPos - i, board)))  {  Piece[,] duplicateBoard = new Piece[8, 8];  duplicateBoard = AI.DuplicateBoard(duplicateBoard, board);  AI.AIUpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos, yGridPos - i, duplicateBoard);  if (!king.Check(king.xGridPos, king.yGridPos, duplicateBoard))  {  possibleMoves[xGridPos, yGridPos - i] = true;  }  if (!Globals.IsPositionEmpty(xGridPos, yGridPos - i, board) && possibleMoves[xGridPos, yGridPos - i])  {  break;  }  }  else  {  break;  }  }  else  {  break;  }  }  for (int i = 1; i < 8; i++)  {  if (yGridPos + i <= 7) // down  {  if (Globals.IsPositionEmpty(xGridPos, yGridPos + i, board) || (!Globals.IsPositionEmpty(xGridPos, yGridPos + i, board) && DifferentColour(xGridPos, yGridPos + i, board)))  {  Piece[,] duplicateBoard = new Piece[8, 8];  duplicateBoard = AI.DuplicateBoard(duplicateBoard, board);  AI.AIUpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos, yGridPos + i, duplicateBoard);  if (!king.Check(king.xGridPos, king.yGridPos, duplicateBoard))  {  possibleMoves[xGridPos, yGridPos + i] = true;  }  if (!Globals.IsPositionEmpty(xGridPos, yGridPos + i, board) && possibleMoves[xGridPos, yGridPos + i])  {  break;  }  }  else  {  break;  }  }  else  {  break;  }  }  return possibleMoves;  }  }  } |

|  |
| --- |
| **3.2.2.9 Rook.cs** |
| namespace ChessProject  {  class Rook : Piece  {  public override bool[,] PossibleMoves(bool[,] possibleMoves, Piece[,] board)  {  King king = new King();  Globals.GetKing(colour, board);  king.xGridPos = Globals.kingX;  king.yGridPos = Globals.kingY;  king = (King)board[Globals.kingX, Globals.kingY];    for (int i = 1; i < 8; i++) // horizontals  {  if (xGridPos - i >= 0) // left  {  if (Globals.IsPositionEmpty(xGridPos - i, yGridPos, board) || (!Globals.IsPositionEmpty(xGridPos - i, yGridPos, board) && DifferentColour(xGridPos - i, yGridPos, board)))  {  Piece[,] duplicateBoard = new Piece[8, 8];  duplicateBoard = AI.DuplicateBoard(duplicateBoard, board);  AI.AIUpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos - i, yGridPos, duplicateBoard);  if (!king.Check(king.xGridPos, king.yGridPos, duplicateBoard))  {  possibleMoves[xGridPos - i, yGridPos] = true;  }  if (!Globals.IsPositionEmpty(xGridPos - i, yGridPos, board) && possibleMoves[xGridPos - i, yGridPos])  {  break;  }  }  else  {  break;  }  }  else  {  break;  }  }  for (int i = 1; i < 8; i++)  {  if (xGridPos + i <= 7) // right  {  if (Globals.IsPositionEmpty(xGridPos + i, yGridPos, board) || (!Globals.IsPositionEmpty(xGridPos + i, yGridPos, board) && DifferentColour(xGridPos + i, yGridPos, board)))  {  Piece[,] duplicateBoard = new Piece[8, 8];  duplicateBoard = AI.DuplicateBoard(duplicateBoard, board);  AI.AIUpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos + i, yGridPos, duplicateBoard);  if (!king.Check(king.xGridPos, king.yGridPos, duplicateBoard))  {  possibleMoves[xGridPos + i, yGridPos] = true;  }  if (!Globals.IsPositionEmpty(xGridPos + i, yGridPos, board) && possibleMoves[xGridPos + i, yGridPos])  {  break;  }  }  else  {  break;  }  }  else  {  break;  }  }  for (int i = 1; i < 8; i++) // verticals  {  if (yGridPos - i >= 0) // up  {  if (Globals.IsPositionEmpty(xGridPos, yGridPos - i, board) || (!Globals.IsPositionEmpty(xGridPos, yGridPos - i, board) && DifferentColour(xGridPos, yGridPos - i, board)))  {  Piece[,] duplicateBoard = new Piece[8, 8];  duplicateBoard = AI.DuplicateBoard(duplicateBoard, board);  AI.AIUpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos, yGridPos - i, duplicateBoard);  if (!king.Check(king.xGridPos, king.yGridPos, duplicateBoard))  {  possibleMoves[xGridPos, yGridPos - i] = true;  }  if (!Globals.IsPositionEmpty(xGridPos, yGridPos - i, board) && possibleMoves[xGridPos, yGridPos - i])  {  break;  }  }  else  {  break;  }  }  else  {  break;  }  }  for (int i = 1; i < 8; i++)  {  if (yGridPos + i <= 7) // down  {  if (Globals.IsPositionEmpty(xGridPos, yGridPos + i, board) || (!Globals.IsPositionEmpty(xGridPos, yGridPos + i, board) && DifferentColour(xGridPos, yGridPos + i, board)))  {  Piece[,] duplicateBoard = new Piece[8, 8];  duplicateBoard = AI.DuplicateBoard(duplicateBoard, board);  AI.AIUpdatePosition(duplicateBoard[xGridPos, yGridPos], xGridPos, yGridPos, xGridPos, yGridPos + i, duplicateBoard);  if (!king.Check(king.xGridPos, king.yGridPos, duplicateBoard))  {  possibleMoves[xGridPos, yGridPos + i] = true;  }  if (!Globals.IsPositionEmpty(xGridPos, yGridPos + i, board) && possibleMoves[xGridPos, yGridPos + i])  {  break;  }  }  else  {  break;  }  }  else  {  break;  }  }  return possibleMoves;  }  }  } |

|  |
| --- |
| **3.2.2.10 Form1.cs** |
| using System;  using System.Diagnostics;  using System.Drawing;  using System.IO;  using System.Threading;  using System.Windows.Forms;  namespace ChessProject  {  public partial class Form1 : Form  {  public string fileSave = "savedGame.txt"; // The name of the file which will store saved games  public bool whiteWin = false; // Stores whether white has won  public bool timeEnabled = false; // Stores whether the users are playing against the clock  public bool gamePaused = false; // Stores whether the game is paused  public bool gameOver = false; // Stores whether the game has finished  public bool playerVComputer = false; // Stores whether the game is one or two player  public bool stalemate = false; // Stores whether the game has resulted in stalemate  public bool checkmate = false; // Stores whether the game has resulted in checkmate  public int whiteTimer; // Stores the time the white player has left  public int blackTimer; // Stores the time the black player has left  public int timeLimit; // Stores the time limit for each player  public Stopwatch whiteStopwatch = new Stopwatch(); // Stopwatch to measure white's time  public Stopwatch blackStopwatch = new Stopwatch(); // Stopwatch to measure black's time  public Piece movedPiece; // Stores the piece which moved most recently  public Form1()  {  InitializeComponent();  }  public void CreatePawn(int i, int j)  {  Pawn newPawn = new Pawn();  newPawn.pieceType = "pawn";  PictureBox newPictureBox = new PictureBox();  if (i == 0)  {  newPawn.colour = "white";  newPawn.xGridPos = j;  newPawn.yGridPos = 6;  newPawn.strength = 10;  }  else  {  newPawn.colour = "black";  newPawn.xGridPos = j;  newPawn.yGridPos = 1;  newPawn.strength = -10;  }  newPawn.AddPiece(newPawn, Globals.grid);  CreatePieceImages(newPawn, newPictureBox, newPawn.colour);  }  public void CreateKnight(int i, int j)  {  Knight newKnight = new Knight();  newKnight.pieceType = "knight";  PictureBox newPictureBox = new PictureBox();  if (i == 0)  {  if (j == 0)  {  newKnight.colour = "white";  newKnight.xGridPos = 1;  newKnight.yGridPos = 7;  }  else  {  newKnight.colour = "white";  newKnight.xGridPos = 6;  newKnight.yGridPos = 7;  }  newKnight.strength = 30;  }  else  {  if (j == 0)  {  newKnight.colour = "black";  newKnight.xGridPos = 1;  newKnight.yGridPos = 0;  }  else  {  newKnight.colour = "black";  newKnight.xGridPos = 6;  newKnight.yGridPos = 0;  }  newKnight.strength = -30;  }  newKnight.AddPiece(newKnight, Globals.grid);  CreatePieceImages(newKnight, newPictureBox, newKnight.colour);  }  public void CreateRook(int i, int j)  {  Rook newRook = new Rook();  newRook.pieceType = "rook";  PictureBox newPictureBox = new PictureBox();  if (i == 0)  {  if (j == 0)  {  newRook.colour = "white";  newRook.xGridPos = 0;  newRook.yGridPos = 7;  }  else  {  newRook.colour = "white";  newRook.xGridPos = 7;  newRook.yGridPos = 7;  }  newRook.strength = 50;  }  else  {  if (j == 0)  {  newRook.colour = "black";  newRook.xGridPos = 0;  newRook.yGridPos = 0;  }  else  {  newRook.colour = "black";  newRook.xGridPos = 7;  newRook.yGridPos = 0;  }  newRook.strength = -50;  }  newRook.AddPiece(newRook, Globals.grid);  CreatePieceImages(newRook, newPictureBox, newRook.colour);  }    public void CreateBishop(int i, int j)  {  Bishop newBishop = new Bishop();  newBishop.pieceType = "bishop";  PictureBox newPictureBox = new PictureBox();  if (i == 0)  {  if (j == 0)  {  newBishop.colour = "white";  newBishop.xGridPos = 2;  newBishop.yGridPos = 7;  }  else  {  newBishop.colour = "white";  newBishop.xGridPos = 5;  newBishop.yGridPos = 7;  }  newBishop.strength = 30;  }  else  {  if (j == 0)  {  newBishop.colour = "black";  newBishop.xGridPos = 2;  newBishop.yGridPos = 0;  }  else  {  newBishop.colour = "black";  newBishop.xGridPos = 5;  newBishop.yGridPos = 0;  }  newBishop.strength = -30;  }  newBishop.AddPiece(newBishop, Globals.grid);  CreatePieceImages(newBishop, newPictureBox, newBishop.colour);  }  public void CreateKing(int i)  {  King newKing = new King();  newKing.pieceType = "king";  PictureBox newPictureBox = new PictureBox();  if (i == 0)  {  newKing.colour = "white";  newKing.xGridPos = 4;  newKing.yGridPos = 7;  newKing.strength = 900;  }  else  {  newKing.colour = "black";  newKing.xGridPos = 4;  newKing.yGridPos = 0;  newKing.strength = -900;  }  newKing.AddPiece(newKing, Globals.grid);  CreatePieceImages(newKing, newPictureBox, newKing.colour);  }  public void CreateQueen(int i)  {  Queen newQueen = new Queen();  newQueen.pieceType = "queen";  PictureBox newPictureBox = new PictureBox();  if (i == 0)  {  newQueen.colour = "white";  newQueen.xGridPos = 3;  newQueen.yGridPos = 7;  newQueen.strength = 90;  }  else  {  newQueen.colour = "black";  newQueen.xGridPos = 3;  newQueen.yGridPos = 0;  newQueen.strength = -90;  }  newQueen.AddPiece(newQueen, Globals.grid);  CreatePieceImages(newQueen, newPictureBox, newQueen.colour);  }  public void CreatePieces()  {  for (int i = 0; i < 2; i++)  {  CreateQueen(i);  CreateKing(i);    for (int j = 0; j < 2; j++)  {  CreateRook(i, j);  CreateKnight(i, j);  CreateBishop(i, j);  }  for (int j = 0; j < 8; j++)  {  CreatePawn(i, j);  }  }  }  public void CreatePieceImages(Piece piece, PictureBox newPictureBox, string colour)  {  if (piece.pieceType == "king")  {  if (colour == "white")  {  newPictureBox.Image = PieceImages.king\_white;  }  else if (colour == "black")  {  newPictureBox.Image = PieceImages.king\_black;  }  }  else if (piece.pieceType == "queen")  {  if (colour == "white")  {  newPictureBox.Image = PieceImages.queen\_white;  }  else if (colour == "black")  {  newPictureBox.Image = PieceImages.queen\_black;  }  }  else if (piece.pieceType == "bishop")  {  if (colour == "white")  {  newPictureBox.Image = PieceImages.bishop\_white;  }  else if (colour == "black")  {  newPictureBox.Image = PieceImages.bishop\_black;  }  }  else if (piece.pieceType == "knight")  {  if (colour == "white")  {  newPictureBox.Image = PieceImages.knight\_white;  }  else if (colour == "black")  {  newPictureBox.Image = PieceImages.knight\_black;  }  }  else if (piece.pieceType == "rook")  {  if (colour == "white")  {  newPictureBox.Image = PieceImages.rook\_white;  }  else if (colour == "black")  {  newPictureBox.Image = PieceImages.rook\_black;  }  }  else if (piece.pieceType == "pawn")  {  if (colour == "white")  {  newPictureBox.Image = PieceImages.pawn\_white;  }  else if (colour == "black")  {  newPictureBox.Image = PieceImages.pawn\_black;  }  }  newPictureBox.Location = new Point(100 + (150 \* piece.xGridPos), 100 + (150 \* piece.yGridPos));  Controls.Add(newPictureBox);  newPictureBox.BringToFront();  newPictureBox.Size = new Size(150, 150);  Globals.pieceImages[piece.xGridPos, piece.yGridPos] = newPictureBox;  if ((piece.xGridPos == 0 || piece.xGridPos == 2 || piece.xGridPos == 4 || piece.xGridPos == 6) && (piece.yGridPos == 0 || piece.yGridPos == 2 || piece.yGridPos == 4 || piece.yGridPos == 6))  {  newPictureBox.BackColor = Color.LightYellow;  }  else if ((piece.xGridPos == 1 || piece.xGridPos == 3 || piece.xGridPos == 5 || piece.xGridPos == 7) && (piece.yGridPos == 1 || piece.yGridPos == 3 || piece.yGridPos == 5 || piece.yGridPos == 7))  {  newPictureBox.BackColor = Color.LightYellow;  }  else  {  newPictureBox.BackColor = Color.Brown;  }  }  private void AddLabel(int i)  {  Label newLabelX = new Label();  newLabelX.Text = Convert.ToString(i + 1);  newLabelX.Location = new Point(150 + (150 \* i), (150 \* 8) + 125);  newLabelX.Size = new Size(50, 50);  Label newLabelY = new Label();  newLabelY.Text = Convert.ToString(i + 1);  newLabelY.Location = new Point(50, 150 + (150 \* i));  newLabelY.Size = new Size(50, 50);  Controls.Add(newLabelX);  Controls.Add(newLabelY);  }  private void AddPanel(int i, int j)  {  Panel newPanel = new Panel();  newPanel.Size = new Size(150, 150);  newPanel.Location = new Point(100 + (150 \* i), 100 + (150 \* j));  Controls.Add(newPanel);  if (i % 2 == 0)  {  if (j % 2 != 0)  {  newPanel.BackColor = Color.Brown;  }  else  {  newPanel.BackColor = Color.LightYellow;  }  }  else  {  if (j % 2 != 0)  {  newPanel.BackColor = Color.LightYellow;  }  else  {  newPanel.BackColor = Color.Brown;  }  }  }    private void Form1\_Load(object sender, EventArgs e)  {  for (int i = 0; i < 8; i++)  {  AddLabel(i);  for (int j = 0; j < 8; j++)  {  AddPanel(i, j);  }  }  WindowState = FormWindowState.Maximized;  }  private void NewGameButton\_Click(object sender, EventArgs e)  {  CreatePieces();  Globals.whiteTurn = true;  SetModeButton.Enabled = true;  ModeGroupBox.Enabled = true;  OnePlayerRadioButton.Enabled = true;  TwoPlayerRadioButton.Enabled = true;  WhiteTakenListBox.Items.Clear();  BlackTakenListBox.Items.Clear();  }  private void SaveGameButton\_Click(object sender, EventArgs e)  {  // This saves the state of the game to a file called saveGame.txt  // The file stores each piece and its colour and location  // The data is stored in this order as this is the order it will be read in when loading a game  // If the game is being played against the clock, the timers for each player will also be saved  // The mode will also be saved (one player or two player)  if (!gameOver)  {  StreamWriter sw = new StreamWriter(fileSave);  for (int i = 0; i < 8; i++)  {  for (int j = 0; j < 8; j++)  {  if (!Globals.IsPositionEmpty(i, j, Globals.grid))  {  Piece piece = Globals.grid[i, j];  sw.WriteLine(piece.pieceType);  sw.WriteLine(piece.colour);  sw.WriteLine(Convert.ToString(piece.xGridPos));  sw.WriteLine(Convert.ToString(piece.yGridPos));  }  }  }  if (timeEnabled)  {  sw.WriteLine(Convert.ToString(whiteTimer));  sw.WriteLine(Convert.ToString(blackTimer));  }  if (playerVComputer)  {  sw.WriteLine("One player");  }  else  {  sw.WriteLine("Two players");  }  if (Globals.whiteTurn)  {  sw.WriteLine("White");  }  else  {  sw.WriteLine("Black");  }  sw.Close();  }  }  private void LoadGameButton\_Click(object sender, EventArgs e)  {  // This reads from the file savedGame.txt until it reaches an empty line  // It reads in the order: piece type, colour, x-ordinate, y-ordinate  // It then creates a new piece and an image and adds them to the game  // After loading the pieces it loads whether the game is being played against the clock  // If it is, the timers for the game will be loaded, and the timeEnabled variable will be set to true  // Then it loads the game mode (one player or two player) and sets the game accordingly  if (File.Exists(fileSave))  {  StreamReader sr = new StreamReader(fileSave);  string line;  int i = 0;  do  {  line = sr.ReadLine();  if (line == "pawn")  {  Pawn newPawn = new Pawn();  newPawn.pieceType = line;  line = sr.ReadLine();  newPawn.colour = line;  line = sr.ReadLine();  newPawn.xGridPos = Convert.ToInt32(line);  line = sr.ReadLine();  newPawn.yGridPos = Convert.ToInt32(line);  PictureBox newPictureBox = new PictureBox();  newPawn.AddPiece(newPawn, Globals.grid);  CreatePieceImages(newPawn, newPictureBox, newPawn.colour);  if (newPawn.colour == "white")  {  newPawn.strength = 10;  }  else  {  newPawn.strength = -10;  }  }  else if (line == "king")  {  King newKing = new King();  newKing.pieceType = line;  line = sr.ReadLine();  newKing.colour = line;  line = sr.ReadLine();  newKing.xGridPos = Convert.ToInt32(line);  line = sr.ReadLine();  newKing.yGridPos = Convert.ToInt32(line);  PictureBox newPictureBox = new PictureBox();  newKing.AddPiece(newKing, Globals.grid);  CreatePieceImages(newKing, newPictureBox, newKing.colour);  if (newKing.colour == "white")  {  newKing.strength = 900;  }  else  {  newKing.strength = -900;  }  }  else if (line == "queen")  {  Queen newQueen = new Queen();  newQueen.pieceType = line;  line = sr.ReadLine();  newQueen.colour = line;  line = sr.ReadLine();  newQueen.xGridPos = Convert.ToInt32(line);  line = sr.ReadLine();  newQueen.yGridPos = Convert.ToInt32(line);  PictureBox newPictureBox = new PictureBox();  newQueen.AddPiece(newQueen, Globals.grid);  CreatePieceImages(newQueen, newPictureBox, newQueen.colour);  if (newQueen.colour == "white")  {  newQueen.strength = 90;  }  else  {  newQueen.strength = -90;  }  }  else if (line == "bishop")  {  Bishop newBishop = new Bishop();  newBishop.pieceType = line;  line = sr.ReadLine();  newBishop.colour = line;  line = sr.ReadLine();  newBishop.xGridPos = Convert.ToInt32(line);  line = sr.ReadLine();  newBishop.yGridPos = Convert.ToInt32(line);  PictureBox newPictureBox = new PictureBox();  newBishop.AddPiece(newBishop, Globals.grid);  CreatePieceImages(newBishop, newPictureBox, newBishop.colour);  if (newBishop.colour == "white")  {  newBishop.strength = 30;  }  else  {  newBishop.strength = -30;  }  }  else if (line == "rook")  {  Rook newRook = new Rook();  newRook.pieceType = line;  line = sr.ReadLine();  newRook.colour = line;  line = sr.ReadLine();  newRook.xGridPos = Convert.ToInt32(line);  line = sr.ReadLine();  newRook.yGridPos = Convert.ToInt32(line);  PictureBox newPictureBox = new PictureBox();  newRook.AddPiece(newRook, Globals.grid);  CreatePieceImages(newRook, newPictureBox, newRook.colour);  if (newRook.colour == "white")  {  newRook.strength = 50;  }  else  {  newRook.strength = -50;  }  }  else if (line == "knight")  {  Knight newKnight = new Knight();  newKnight.pieceType = line;  line = sr.ReadLine();  newKnight.colour = line;  line = sr.ReadLine();  newKnight.xGridPos = Convert.ToInt32(line);  line = sr.ReadLine();  newKnight.yGridPos = Convert.ToInt32(line);  PictureBox newPictureBox = new PictureBox();  newKnight.AddPiece(newKnight, Globals.grid);  CreatePieceImages(newKnight, newPictureBox, newKnight.colour);  if (newKnight.colour == "white")  {  newKnight.strength = 30;  }  else  {  newKnight.strength = -30;  }  }  else if (Int32.TryParse(line, out i))  {  whiteTimer = Convert.ToInt32(line);  line = sr.ReadLine();  blackTimer = Convert.ToInt32(line);  }  else if (line == "One player")  {  playerVComputer = true;  }  else if (line == "Two players")  {  playerVComputer = false;  }  else if (line == "White")  {  Globals.whiteTurn = true;  }  else if (line == "Black")  {  Globals.whiteTurn = false;  }  } while (line != null);  sr.Close();  StartGameButton.Enabled = true;  }  else  {  MessageBox.Show("There is no saved game.");  }  }  private void SetTimerButton\_Click(object sender, EventArgs e)  {  timeEnabled = true;  if (OneMinRadioButton.Checked)  {  timeLimit = 60;  WhiteTimer.Text = "1:00";  BlackTimer.Text = "1:00";  }  else if (FiveMinRadioButton.Checked)  {  timeLimit = 300;  WhiteTimer.Text = "5:00";  BlackTimer.Text = "5:00";  }  else if (TenMinRadioButton.Checked)  {  timeLimit = 600;  WhiteTimer.Text = "10:00";  BlackTimer.Text = "10:00";  }  WhiteTimer.Visible = true;  BlackTimer.Visible = true;  WhiteTimer.ForeColor = Color.Green;  BlackTimer.ForeColor = Color.Green;  whiteTimer = timeLimit;  blackTimer = timeLimit;  StartGameButton.Enabled = true;  }  private void SetModeButton\_Click(object sender, EventArgs e)  {  if (OnePlayerRadioButton.Checked)  {  playerVComputer = true;  StartGameButton.Enabled = true;  }  else if (TwoPlayerRadioButton.Checked)  {  playerVComputer = false;  StartGameButton.Enabled = true;  SetTimerButton.Enabled = true;  TimeGroupBox.Enabled = true;  OneMinRadioButton.Enabled = true;  FiveMinRadioButton.Enabled = true;  TenMinRadioButton.Enabled = true;  }  }  private void StartGameButton\_Click(object sender, EventArgs e)  {  StartGameButton.Enabled = false;  PauseGameButton.Enabled = true;  gameOver = false;  gamePaused = false;  NewGameButton.Enabled = false;  LoadGameButton.Enabled = false;  SaveGameButton.Enabled = true;  SetTimerButton.Enabled = false;  TimeGroupBox.Enabled = false;  SetModeButton.Enabled = false;  ModeGroupBox.Enabled = false;  MoveLabel.Visible = true;  TurnLabel.Visible = true;  CurrentXTextBox.Enabled = true;  CurrentYTextBox.Enabled = true;  NewXTextBox.Enabled = true;  NewYTextBox.Enabled = true;  MoveButton.Enabled = true;  if (timeEnabled)  {  whiteStopwatch.Start();  WhiteTimer.ForeColor = Color.Red;  }  }  private void PauseGameButton\_Click(object sender, EventArgs e)  {  gamePaused = true;  if (timeEnabled)  {  if (Globals.whiteTurn)  {  whiteStopwatch.Stop();  WhiteTimer.Text = Convert.ToString(whiteTimer / 60) + ":" + Convert.ToString(whiteTimer % 60);  }  else  {  blackStopwatch.Stop();  BlackTimer.Text = Convert.ToString(blackTimer / 60) + ":" + Convert.ToString(blackTimer % 60);  }  }  StartGameButton.Enabled = true;  PauseGameButton.Enabled = false;  }  private void MoveButton\_Click(object sender, EventArgs e)  {  string currentXStr = CurrentXTextBox.Text;  string currentYStr = CurrentYTextBox.Text;  string newXStr = NewXTextBox.Text;  string newYStr = NewYTextBox.Text;  string colour;  CurrentXTextBox.Clear();  CurrentYTextBox.Clear();  NewXTextBox.Clear();  NewYTextBox.Clear();  if (Globals.whiteTurn)  {  colour = "white";  }  else  {  colour = "black";  }  if (currentXStr == "" || currentYStr == "" || newXStr == "" || newYStr == "")  {  MoveLabel.Text = "Please enter a number in all boxes.";  }  else if (currentXStr.Length > 1 || currentYStr.Length > 1 || newXStr.Length > 1 || newYStr.Length > 1)  {  MoveLabel.Text = "Please enter a single digit from 1 to 8 in all boxes.";  }  else  {  if (!(Convert.ToChar(currentXStr) >= 49 && Convert.ToChar(currentXStr) <= 56) && (Convert.ToChar(currentYStr) >= 49 && Convert.ToChar(currentYStr) <= 56) && (Convert.ToChar(newXStr) >= 49 && Convert.ToChar(newXStr) <= 56) && (Convert.ToChar(newYStr) >= 49 && Convert.ToChar(newYStr) <= 56))  {  MoveLabel.Text = "Please enter a single digit from 1 to 8 in all boxes.";  }  else  {  int currentX = Convert.ToInt32(currentXStr) - 1;  int currentY = Convert.ToInt32(currentYStr) - 1;  if (!Globals.IsPositionEmpty(currentX, currentY, Globals.grid))  {  if (Globals.grid[currentX, currentY].colour == colour)  {  int newX = Convert.ToInt32(newXStr) - 1;  int newY = Convert.ToInt32(newYStr) - 1;  Piece piece = Globals.grid[currentX, currentY];  bool[,] possibleMoves = new bool[8, 8];  possibleMoves = piece.PossibleMoves(possibleMoves, Globals.grid);  if (possibleMoves[newX, newY])  {  piece.UpdatePosition(piece, currentX, currentY, newX, newY);  if (colour == "white")  {  MoveLabel.Text = string.Format("White has moved a white {0} to ({1}, {2}).", piece.pieceType, newX, newY);  }  else  {  MoveLabel.Text = string.Format("Black has moved a black {0} to ({1}, {2}).", piece.pieceType, newX, newY);  }  movedPiece = piece;  EndOfTurn();  if (playerVComputer)  {  Thread.Sleep(1000);  AI.Turn();  Thread.Sleep(1000);  EndOfTurn();  }  }  else  {  MoveLabel.Text = "Please make a legal move.";  }  }  else  {  MoveLabel.Text = string.Format("Please select a {0} piece.", colour);  }  }  else  {  MoveLabel.Text = "Please select a piece.";  }  }  }  }  public bool EndOfTurn()  {  if (movedPiece.pieceType == "pawn")  {  if (movedPiece.colour == "white")  {  if (movedPiece.yGridPos == 0)  {  QueenConvert(movedPiece);  }  }  else  {  if (movedPiece.yGridPos == 7)  {  QueenConvert(movedPiece);  }  }  }  Globals.GetKing("white", Globals.grid);  King whiteKing = new King();  whiteKing.xGridPos = Globals.kingX;  whiteKing.yGridPos = Globals.kingY;  whiteKing.colour = "white";  Globals.GetKing("black", Globals.grid);  King blackKing = new King();  blackKing.xGridPos = Globals.kingX;  blackKing.yGridPos = Globals.kingY;  blackKing.colour = "black";  if (whiteKing.Checkmate(whiteKing.xGridPos, whiteKing.yGridPos, Globals.grid) || blackKing.Checkmate(blackKing.xGridPos, blackKing.yGridPos, Globals.grid) || Stalemate(Globals.grid))  {  if (whiteKing.Checkmate(whiteKing.xGridPos, whiteKing.yGridPos, Globals.grid))  {  whiteWin = false;  checkmate = true;  stalemate = false;  gameOver = true;  StateOfGameLabel.Text = "Checkmate!";  }  else if (blackKing.Checkmate(blackKing.xGridPos, blackKing.yGridPos, Globals.grid))  {  whiteWin = true;  checkmate = true;  stalemate = false;  gameOver = true;  StateOfGameLabel.Text = "Checkmate!";  }  else if (Stalemate(Globals.grid))  {  whiteWin = false;  checkmate = false;  stalemate = true;  gameOver = true;  StateOfGameLabel.Text = "Stalemate!";  }  StateOfGameLabel.Visible = true;  GameOver();  return gameOver;  }  if (whiteKing.Check(whiteKing.xGridPos, whiteKing.yGridPos, Globals.grid) || blackKing.Check(blackKing.xGridPos, blackKing.yGridPos, Globals.grid))  {  StateOfGameLabel.Text = "Check!";  StateOfGameLabel.Visible = true;  }  else  {  StateOfGameLabel.Visible = false;  }  if (Globals.whiteTurn)  {  if (timeEnabled)  {  whiteStopwatch.Stop();  WhiteTimer.ForeColor = Color.Green;  UpdateTimer();  if (whiteTimer <= 0)  {  gameOver = true;  whiteWin = false;  stalemate = false;  checkmate = false;  GameOver();  return gameOver;  }  }  Globals.whiteTurn = false;  TurnLabel.Text = "Black's Turn";  }  else  {  if (timeEnabled)  {  blackStopwatch.Stop();  BlackTimer.ForeColor = Color.Green;  UpdateTimer();  if (blackTimer <= 0)  {  gameOver = true;  whiteWin = true;  stalemate = false;  checkmate = false;  GameOver();  return gameOver;  }  }  Globals.whiteTurn = true;  TurnLabel.Text = "White's Turn";  }  if (Globals.takenPiece != null)  {  if (Globals.takenPiece.colour == "white")  {  WhiteTakenListBox.Items.Add(Globals.takenPiece.pieceType);  }  else  {  BlackTakenListBox.Items.Add(Globals.takenPiece.pieceType);  }  Globals.takenPiece = null;  }  return gameOver;  }  public void QueenConvert(Piece pawn)  {  pawn.RemovePiece(pawn);    Queen newQueen = new Queen();  newQueen.colour = pawn.colour;  newQueen.pieceType = "queen";  newQueen.xGridPos = pawn.xGridPos;  newQueen.yGridPos = pawn.yGridPos;  newQueen.AddPiece(newQueen, Globals.grid);  PictureBox newPictureBox = new PictureBox();  CreatePieceImages(newQueen, newPictureBox, newQueen.colour);  }  public void UpdateTimer()  {  if (Globals.whiteTurn)  {  TimeSpan ts = whiteStopwatch.Elapsed;  whiteTimer -= ((ts.Minutes \* 60) + ts.Seconds);  if (whiteTimer % 60 >= 10)  {  WhiteTimer.Text = Convert.ToString(whiteTimer / 60) + ":" + Convert.ToString(whiteTimer % 60);  }  else  {  WhiteTimer.Text = Convert.ToString(whiteTimer / 60) + ":0" + Convert.ToString(whiteTimer % 60);  }  whiteStopwatch.Reset();  }  else  {  TimeSpan ts = blackStopwatch.Elapsed;  blackTimer -= ((ts.Minutes \* 60) + ts.Seconds);  if (blackTimer % 60 >= 10)  {  BlackTimer.Text = Convert.ToString(blackTimer / 60) + ":" + Convert.ToString(blackTimer % 60);  }  else  {  BlackTimer.Text = Convert.ToString(blackTimer / 60) + ":0" + Convert.ToString(blackTimer % 60);  }  blackStopwatch.Reset();  }  }    private void GameOver()  {  if (checkmate)  {  if (whiteWin)  {  MessageBox.Show("Checkmate! White wins!");  }  else  {  MessageBox.Show("Checkmate! Black wins!");  }  }  else if (stalemate)  {  MessageBox.Show("Stalemate");  }  else if (whiteTimer <= 0)  {  MessageBox.Show("White ran out of time. Black wins!");  }  else if (blackTimer <= 0)  {  MessageBox.Show("Black ran out of time. White wins!");  }  for (int i = 0; i < 8; i++)  {  for (int j = 0; j < 8; j++)  {  if (!Globals.IsPositionEmpty(i, j, Globals.grid))  {  Globals.grid[i, j].RemovePiece(Globals.grid[i, j]);  }  }  }  NewGameButton.Enabled = true;  LoadGameButton.Enabled = true;  SaveGameButton.Enabled = false;  if (timeEnabled)  {  WhiteTimer.Text = "0:00";  BlackTimer.Text = "0:00";  WhiteTimer.Visible = false;  BlackTimer.Visible = false;  }  }  public bool Stalemate(Piece[,] board)  {  // Check for certain piece combinations which result in stalemate  bool whiteStalemateCombination = StalemateCombinations("white", board);  bool blackStalemateCombination = StalemateCombinations("black", board);  if (whiteStalemateCombination && blackStalemateCombination)  {  stalemate = true;  return stalemate;  }  // From here, if both of the other conditions are false, it is stalemate  Globals.GetKing("white", board);  King whiteKing = new King();  whiteKing.xGridPos = Globals.kingX;  whiteKing.yGridPos = Globals.kingY;  Globals.GetKing("black", board);  King blackKing = new King();  blackKing.xGridPos = Globals.kingX;  blackKing.yGridPos = Globals.kingY;  // If there are any possible moves it is not stalemate  for (int i = 0; i < 8; i++)  {  for (int j = 0; j < 8; j++)  {  if (!Globals.IsPositionEmpty(i, j, board))  {  for (int k = 0; k < 8; k++)  {  for (int l = 0; l < 8; l++)  {  bool[,] possibleMoves = new bool[8, 8];  possibleMoves = board[i, j].PossibleMoves(possibleMoves, board);  if (possibleMoves[k, l])  {  stalemate = false;  return stalemate;  }  }  }  }  }  }  // If the king is in check it is not stalemate  if (whiteKing.Check(whiteKing.xGridPos, whiteKing.yGridPos, board) || blackKing.Check(blackKing.xGridPos, blackKing.yGridPos, board))  {  stalemate = false;  return stalemate;  }  // If both of the previous conditions are false it is stalemate  stalemate = true;  return stalemate;  }  public bool StalemateCombinations(string colour, Piece[,] board)  {  // There are three different combinations of pieces that cannot checkmate a king:  // A king and a knight  // A king and two knights  // A king and a bishop  // Therefore, if both sides have any of these three combinations, the game is stalemate  int pawnCount = 0;  int bishopCount = 0;  int knightCount = 0;  int rookCount = 0;  int kingCount = 0;  int queenCount = 0;  for (int i = 0; i < 8; i++)  {  for (int j = 0; j < 8; j++)  {  if (!Globals.IsPositionEmpty(i, j, board))  {  if (board[i, j].colour == colour)  {  if (board[i, j].pieceType == "pawn")  {  pawnCount++;  }  else if (board[i, j].pieceType == "bishop")  {  bishopCount++;  }  else if (board[i, j].pieceType == "knight")  {  knightCount++;  }  else if (board[i, j].pieceType == "rook")  {  rookCount++;  }  else if (board[i, j].pieceType == "king")  {  kingCount++;  }  else if (board[i, j].pieceType == "queen")  {  queenCount++;  }  }  }  }  }  if (kingCount == 1 && knightCount == 1 && pawnCount == 0 && bishopCount == 0 && rookCount == 0 && queenCount == 0)  {  return true;  }  else if (kingCount == 1 && knightCount == 2 && pawnCount == 0 && bishopCount == 0 && rookCount == 0 && queenCount == 0)  {  return true;  }  else if (kingCount == 1 && knightCount == 0 && pawnCount == 0 && bishopCount == 1 && rookCount == 0 && queenCount == 0)  {  return true;  }  return false;  }  }  } |

### Testing

Evidence of testing is given in Section 6 in the form of either photos or links to videos.

Testing was conducted in two player mode unless otherwise stated (testing the AI).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test** | **Expected output** | **Actual output** | **Possible cause of errors** | **Changes made** | **Evidence** |
| 1. Test whether the program loads successfully and in full screen. | The form loads with the board created and loads in full screen. | The form loads with the board created and loads in full screen. | N/A | N/A | 6.1 |
| 1. Test whether the pieces are created successfully and represented on the interface with the correct images in the correct positions. | The images of the pieces will be correctly placed on the board. | The images of the pieces are correctly placed on the board. | N/A | N/A | 6.2 |
| 1. Test whether setting a timer in two player mode works properly. | Timers will appear on both sides of the board. | Timers appear on both sides of the board. | N/A | N/A | 6.3 |
| 1. Test whether the timers tick down to zero and whether the game ends when this happens. | The timers will tick down until one reaches zero when the game will end. | Stack overflow error. | When checking if a king is in check, the program calls PossibleMoves for every piece on the board. | Added an if statement to the Check algorithm to ensure the PossibleMoves algorithm is only called for pieces of the same colour as the king. | 6.4.1 |
| 1. Repeat of 4 | The timers will tick down until one reaches zero when the game will end. | The white timer ticked down to zero, and the game ended. | N/A | N/A | 6.4.2 |
| 1. Test whether the computer makes a legal move. | The computer will make a legal move (likely moving a pawn). | Null exception | The algorithm was checking the colour of an empty position as it updated the piece’s position before duplicating the board rather than after. | Moved a call of DuplicateBoard from after a call of UpdatePosition to before, in the GenerateMoves algorithm. | 6.5.1 |
| 1. Repeat of 6. | The computer will make a legal move (likely moving a pawn). | Stack overflow error. | There is an infinite loop because the PossibleMoves algorithm needs to know if the king is in check. To know if the king is in check, the Check algorithm needs to know the possible moves for all pieces of the same colour, however this can’t be done without knowing if the king is in check. | Wrote a different Check algorithm which doesn’t use the PossibleMoves algorithm. | 6.5.2 |
| 1. Repeat of 6. | The computer will make a legal move (likely moving a pawn). | No move made, and the game produced the expected output for when white checkmates black. | Could possibly be due to forgetting to put if statements in some of the checks for the new Check algorithm. | Added if statements to the Check algorithm. | 6.5.3 |
| 1. Repeat of 6. | The computer will make a legal move (likely moving a pawn). | Same as last test. However, after acknowledging the end of game message, an argument out of range exception was thrown. | The Checkmate algorithm is incorrectly returning true, and this is occurring before the AI move, therefore crashing the program. The exception was thrown in the AI Turn algorithm, suggesting that the AI is trying to move after the game has finished. | Fixed the AI trying to play after the game has finished by adding gameOver = true to the if statements for checkmate and stalemate in EndOfTurn. Added an if statement to the Check algorithm for the diagonal moves to ensure only diagonals are checked. | 6.5.4 |
| 1. Repeat of 6. | The computer will make a legal move (likely moving a pawn). | The Checkmate algorithm correctly returns false, but the Minimax algorithm incorrectly returns 1290, where there are no moves which would result in such a score. | The issue could be with the alpha-beta pruning. | Commented out alpha-beta pruning to check if that is the issue. | 6.5.4 |
| 1. Repeat of 6. | The computer will make a legal move (likely moving a pawn). | No change, so the issue is not the alpha-beta pruning. | The issue is within the minimax algorithm, as each call returns either 1290 or -1290. | Uncommented alpha-beta pruning. Added various optimisations for the PossibleMoves algorithms for all classes and the GetKing algorithm. Moved the assignment of resultPos into the Minimax algorithm to check if that is the issue. | 6.5.4 |
| 1. Repeat of 6. | The computer will make a legal move (likely moving a pawn). | The Minimax algorithm still incorrectly returns 1290 or -1290, however the computer does make a move. The move is an illegal move, as the pawn moves the wrong way and takes a rook of the same colour. | newX and newY are never assigned and so are always 0. | Assigned newX and newY in the GenerateMoves algorithm. | 6.5.5 |
| 1. Repeat of 6. | The computer will make a legal move (likely moving a pawn). | The computer makes some legal moves, before pieces start moving randomly and disappearing. | The background colours of the images don’t update to match the board, and an image of a pawn disappears before the same pawn then moves illegally to take a knight. | Fixed the error in the PossibleMoves algorithm for the Pawn class which allowed the pawn to illegally take the white knight. | 6.5.6 |
| 1. The computer will only make legal moves. | The computer will move legally every time. | The pawn now takes a white pawn rather than a knight, but after that the computer continues making illegal moves, such as taking the white king. | Unknown. | Created separate algorithms for the AI for RemovePiece, UpdatePosition, and QueenConvert. | 6.5.7 |
| 1. Repeat of 14. | The computer will move legally every time. | The pawn moves illegally, before the rook illegally takes both white rooks. The same sequence of moves from the previous test then occurs. | The illegal rook move was caused by break statements being in a separate if statement, rather than an else statement in PossibleMoves. | Changed if statements to else statements in the PossibleMoves algorithms for bishop, rook, and queen. | 6.5.8 |
| 1. The player can play a turn. | The player can move a piece legally. | Starting the game freezes the UI so the player cannot move, and therefore cannot get out of a do while loop. | Starting the game freezes the UI so the player cannot move, and therefore cannot get out of a do while loop. | Made PickOrDropPiece an async Task method, and NewPictureBox\_MouseDown and PlayerTurn async void methods and added await statements. | 6.6.1 |
| 1. Repeat of 16. | The player can move a piece legally. | No change. | Starting the game freezes the UI so the player cannot move, and therefore cannot get out of a do while loop. | Made Turn an async void method and added await statements. | 6.6.1 |
| 1. Repeat of 16. | The player can move a piece legally. | No change. | Starting the game freezes the UI so the player cannot move, and therefore cannot get out of a do while loop. | Removed the calling of Turn algorithm from the StartGameButton\_Click algorithm. | 6.6.1 |
| 1. Test whether removing Turn prevents the UI freezing. | The user can press buttons and move a piece. | The UI no longer freezes but the user cannot make a move. | There is no event handler for the picture boxes being clicked. | Added an event handler for the picture boxes being clicked. | 6.6.2 |
| 1. Repeat of 16. | The player can move a piece legally. | No change. | Unknown. | Changed the method called by the event handler to PickOrDropPiece, rather than NewPictureBox\_MouseDown to test if that changes anything. | 6.6.2 |
| 1. Repeat of 16. | The player can move a piece legally. | No change. | Unknown. | Removed code for clicking a piece and moving it in favour of asking the user to enter co-ordinates and pressing a button to make the move. Added a label to denote whose turn it is. | 6.6.2 |
| 1. Repeat of 16. | The player can move a piece legally. | The player and the AI move legally. | N/A | N/A | 6.6.3 |
| 1. Test whether the game saves successfully. | The state of the game will be correctly saved to the file. | The state of the game is correctly saved to the file. | N/A | N/A | 6.7 |
| 1. Test whether the game loads successfully. | The state of the game will be correctly loaded to the program and the pieces will be represented on the board. | Nothing happens. | Pieces are not assigned a pieceType when loaded. | Assigned pieceType when loading pieces. | 6.8.1 |
| 1. Repeat of 23. | The state of the game will be correctly loaded to the program and the pieces will be represented on the board. | The game is successfully loaded. | N/A | N/A | 6.8.2 |
| 1. Test whether the program recognises when the game is in check. | A label saying Check will appear at the top of the screen when the king is in check. | The program says that the king is in check through a pawn. | Missing break statements for pieces of the same colour as the king. | Added break statements for when a piece is the same colour as the king. | 6.9.1 |
| 1. Repeat of 26. | A label saying Check will appear at the top of the screen when the king is in check. | The program correctly displays the check message when the king is in check. | N/A | N/A | 6.9.2 |
| 1. Test whether the program recognises when the game is in checkmate. | The game should end with a display message saying the game is in checkmate and who the winner is. | The game recognises it is check, but not checkmate. There are also a few issues where pawns are putting kings in check where they shouldn’t do. | The Check algorithm doesn’t check the colour of the pawn when determining if a king is in check. | Added a colour check for pawns in the Check algorithm. | 6.10.1 |
| 1. Repeat of 28. | The game should end with a display message saying the game is in checkmate and who the winner is. | Checkmate still not recognised. | Missing ! in a DifferentColour check so it was checking if pieces of the opposite colour to the king could move. | Added ! in the DifferentColour check. Simplified the Checkmate algorithm by removing the check specifically for the king’s possible moves. | 6.10.2 |
| 1. Repeat of 28. | The game should end with a display message saying the game is in checkmate and who the winner is. | Checkmate is recognised and the game ends. | N/A | N/A | 6.10.3 |
| 1. Test whether the program recognises when the game is in stalemate. | The game should end with a display message saying the game is in stalemate. | Stalemate is recognised and the game ends. | N/A | N/A | 6.11 |
| 1. Test whether pawns convert to queens if they reach the other side of the board. | The pawn will become a queen when it reaches the other side of the board. | Null error. | The QueenConvert algorithm uses xGridPos and yGridPos when updating the image, where it should be using newX and newY. | Changed xGridPos and yGridPos to newX and newY. | 6.12.1 |
| 1. Repeat of 30. | The pawn will become a queen when it reaches the other side of the board. | The pawn converts to a queen but moves to a different position. | The previous code was actually correct, but the picture didn’t exist. | Reverted back to the previous code and created a new picture box. Had to move QueenConvert to Form1.cs to do this. | 6.12.2 |
| 1. Repeat of 30. | The pawn will become a queen when it reaches the other side of the board. | The pawn converts to a queen in the correct position. | N/A | N/A | 6.12.3 |

### Evaluation

**Table of Contents**

[**5.1** **Evaluation of objectives** 101](#_Toc129439124)

[**5.2** **How the program could be improved** 103](#_Toc129439125)

[**5.3** **Conclusion** 104](#_Toc129439126)

* 1. **Evaluation of objectives**

This section includes an evaluation of the objectives and how they have or have not been met.

|  |  |  |
| --- | --- | --- |
| **Objective** | **Met?** | **Explanation** |
| 1.5.1 – To create a chess game that allows a user to play a game of chess against an AI. | Yes | The player can a game of chess against an AI or against another player. |
| 1.5.1.1 – Move Generation – all possible moves for each AI-controlled piece will be generated. | Yes | The move generation algorithm was implemented as GenerateMoves in the AI class. The algorithm generates every possible move for each piece of the specified colour and creates a 2D array of pieces for each move to represent the board after that move has taken place. This is then added to a list of the possible boards. |
| 1.5.1.2 – Positional Evaluation – the relative strength of each piece will be added to calculate the state of the board after each move generated. | Yes | Positional Evaluation was implemented through the Evaluation algorithm in the AI class. This adds the relative strength of all of the pieces on the board and is used to determine which player is in the stronger position after a move has been made. |
| 1.5.1.3 – Minimax Search Tree – this will search through all the possible moves to find the best one. | Yes | The Minimax Search Tree was implemented in the Minimax algorithm in the AI class. Using all of the outcomes generated in the GenerateMoves algorithm, the Minimax algorithm applies the Evaluation algorithm to each outcome, with a search depth of three, to determine the best possible move. |
| 1.5.1.4 – Alpha-Beta Pruning – an optimisation to the Minimax search. | Yes | Alpha-Beta Pruning was added to the Minimax algorithm to greatly reduce the number of nodes that needed to be searched in the Minimax tree. |
| 1.5.2 – To implement algorithms which determine the state of the game. | Yes | All three of the game states for chess have implemented algorithms which determine whether the game is in that state. The program checks for checkmate and stalemate at the end of every turn. |
| 1.5.2.1 – Check. | Yes | The Check algorithm was implemented in the King class to determine whether that king is in check. This is done by searching every position that could possibly have a piece which could move to the king’s position and checking if that position is occupied by an enemy piece which can move to the king’s position. The algorithm is used when generating moves for each piece as moves will only be added to the list if the king is not in check. The Check algorithm is also used in the Checkmate algorithm below. |
| 1.5.2.2 – Checkmate. | Yes | The Checkmate algorithm was implemented in the King class to determine whether that king is in checkmate. This is done by first determining whether the king is in check. The algorithm then checks if the king has any possible moves. If the king has no possible moves, the rest of the pieces of the same colour are checked. If there are no possible moves the king is in checkmate. |
| 1.5.2.3 – Stalemate. | Yes | There are two separate algorithms for stalemate: the Stalemate algorithm, and the StalemateCombinations algorithm, both of which were implemented in the Form1 class. The StalemateCombinations algorithm checks for certain combinations of pieces on both sides. This is then used in the Stalemate algorithm. If both sides have one of these combinations, the game is stalemate. The game then checks the other conditions: if there are any possible moves, and if a king is in check. If both of these conditions are false the game is stalemate. |
| 1.5.3 – To allow the user to save their progress and come back to the game at a later date. | Yes | The user can save games to savedGame.txt and later load them again successfully and continue playing by hitting the start button. |
| 1.5.3.1 – Saving – when saving the program will write to a .txt file called ‘savedGame’. | Yes | The state of the game is successfully saved to savedGame.txt. |
| 1.5.3.2 – Loading – when loading a game, the program will read from savedGame.txt in the same order in which it wrote to the file. | Yes | The saved game is successfully loaded, and the pieces are represented on the board. |
| 1.5.4 – To allow users to create a profile. | No | This was not implemented as the AI and other features, such as playing against a timer, or saving and loading a game were prioritised. |
| 1.5.4.1 – The user will be able to save their stats and profile to a unique .txt file. | No | This was not implemented as the AI and other features, such as playing against a timer, or saving and loading a game were prioritised. |
| 1.5.4.2 – This file will store their username, number of games played, number of wins, draws, and losses. | No | This was not implemented as the AI and other features, such as playing against a timer, or saving and loading a game were prioritised. |
| 1.5.4.3 – Using the stats of all losers, a leader board will be created, ranking users by their wins and draws. | No | This was not implemented as the AI and other features, such as playing against a timer, or saving and loading a game were prioritised. |
| 1.5.5 – To allow users to play against the clock when playing in two player. | Yes | Timers for both players were implemented for two player games only. The relevant buttons are only enabled when the user sets the mode to two player. |
| 1.5.5.1 – The user will have the option to set a timer for one, five, or ten minutes. | Yes | Setting the mode to two player enables the buttons to set the timer. There is the option to set the timer for one, five, or ten minutes. The timer then appears on both sides of the screen. |
| 1.5.5.2 – The time will start ticking at the beginning of a player’s turn and stop when their turn ends. | Yes | The timer starts ticking when the player’s turn begins and stops when it ends. The colour of the timer changes from green to red when it is ticking down. |
| 1.5.5.3 – During each player’s turn, their clock will tick down using the stopwatch function. If a player’s time reaches zero, that player will be able to finish their turn. If their turn results in a stalemate, the game will end as a draw. If their turn results in a checkmate, they will win. If neither of these happen, they will lose. | Yes | Once a player’s timer runs out they are able to finish their turn and the program checks for checkmate and stalemate. If either of these algorithms returns true, the appropriate ending is executed. If it is neither checkmate nor stalemate, the player whose timer reached zero loses and the appropriate ending is executed. |

* 1. **How the program could be improved**

The program could be improved by implementing the final feature in the objectives, the profile system (1.5.4). This was not implemented as its importance to the project was far less than the importance of getting the AI working and implementing player movement, so the profile system was not prioritised. The large number of problems encountered during testing the AI and in particular the player movement meant that there was not enough time to implement the profile system, which was something which was identified as a possible feature during the communication with Anders, however it was not deemed essential to the program.

Another way in which the program could be improved is by implementing movement by clicking the pieces and where you would like to move them to. I struggled to get this to work and instead opted for asking the user to type the co-ordinates of the piece they would like to move, and the co-ordinates they would like to move to. The current way of typing the co-ordinates works but is tedious.

* 1. **Conclusion**

In conclusion, the program successfully enables the user to play a game of chess against either an AI or another player. Anders’ requested features of playing against the clock and saving and loading games were implemented, however the profile and leader board system was not.

### Screenshots and Videos

|  |
| --- |
| * 1. **Testing whether the program loads successfully and in full screen** |
| * 1. **Testing whether pieces are created correctly** |
| * 1. **Testing whether timers appear when selected** |
| * 1. **Testing whether the timers work**      1. **Stack overflow error**      * + 1. **Working timers** |
| * 1. **Testing whether the AI moves legally**      1. **Null exception**      * + 1. **Stack overflow error**      * + 1. **No move made, white checkmate**      * + 1. **Out of range exception**      * + 1. **Illegal move** |
| * + 1. **AI moving**   <https://www.youtube.com/watch?v=c6oO4Kd6kk0>   * + 1. **AI moving 2**   <https://www.youtube.com/watch?v=r4IS3w9w-2g>   * + 1. **AI moving broken**   <https://www.youtube.com/watch?v=I0oNJMqgy64> |
| * 1. **Testing player movement**      1. **UI freezing**   <https://www.youtube.com/watch?v=0mMaRCHzwVE>   * + 1. **UI not freezing but movement not working**   <https://www.youtube.com/watch?v=fX0v2aM982U>   * + 1. **Player and AI moving**   <https://www.youtube.com/watch?v=bOKyM7mXcdw> |
| * 1. **File saving**   <https://www.youtube.com/watch?v=FIWR32NCQ5M> |
| * 1. **File loading**      1. **Unsuccessful load**   <https://www.youtube.com/watch?v=CZ1Ah7yAgls>   * + 1. **Successful load**   <https://www.youtube.com/watch?v=hrxUg8ISKrc> |
| * 1. **Check**      1. **Incorrect check**   <https://www.youtube.com/watch?v=8V0kEtQgIrw>   * + 1. **Correct check**   <https://www.youtube.com/watch?v=2pLMuNEAAoA> |
| * 1. **Checkmate**      1. **Checkmate not working (Full game)**   <https://www.youtube.com/watch?v=e-9xPu6WwEI>   * + 1. **Checkmate not working**   <https://www.youtube.com/watch?v=5zksPjWsi7o>   * + 1. **Checkmate**   <https://www.youtube.com/watch?v=mqmfGuT4lds> |
| * 1. **Stalemate**   <https://www.youtube.com/watch?v=csKd3uSL2lY> |
| * 1. **QueenConvert**      1. **Null error**   <https://www.youtube.com/watch?v=-FGry4wDrA0>   * + 1. **Successful conversion in wrong position**   <https://www.youtube.com/watch?v=TVVXNY_rnyY>   * + 1. **Successful conversion**   <https://www.youtube.com/watch?v=dtRrefEzmOA> |

### References

1. [https://chessquestions.com/stalemate-in-chess/#:~:text=The%203%20Combinations%20Pieces%20that%20will%20cause%20a,Two%20Knights%20...%203%203.%20King%20and%20Bishop](https://chessquestions.com/stalemate-in-chess/%23:~:text=The%203%20Combinations%20Pieces%20that%20will%20cause%20a,Two%20Knights%20...%203%203.%20King%20and%20Bishop)

[https://www.freecodecamp.org/news/simple-chess-ai-step-by-step-1d55a9266977/#:~:text=A%20step-by-step%20guide%20to%20building%20a%20simple%20chess,...%205%20Step%205%3A%20Improved%20evaluation%20function%20](https://www.freecodecamp.org/news/simple-chess-ai-step-by-step-1d55a9266977/%23:~:text=A%20step-by-step%20guide%20to%20building%20a%20simple%20chess,...%205%20Step%205%3A%20Improved%20evaluation%20function%20)

1. <https://philippmuens.com/minimax-and-mcts>
2. <https://stackoverflow.com/questions/22882110/how-to-implement-movement-in-a-chess-game>
3. <https://stackoverflow.com/questions/66082633/minimax-algorithm-with-chess-in-c-sharp-not-working-properly>

1. [Stalemate In Chess: Rules, Tips & Pieces To Avoid - Chess Questions](https://chessquestions.com/stalemate-in-chess/#:~:text=The%203%20Combinations%20Pieces%20that%20will%20cause%20a,Two%20Knights%20...%203%203.%20King%20and%20Bishop) [↑](#footnote-ref-1)
2. [A step-by-step guide to building a simple chess AI (freecodecamp.org)](https://www.freecodecamp.org/news/simple-chess-ai-step-by-step-1d55a9266977/#:~:text=A%20step-by-step%20guide%20to%20building%20a%20simple%20chess,...%205%20Step%205%3A%20Improved%20evaluation%20function%20) [↑](#footnote-ref-2)
3. [Minimax and Monte Carlo Tree Search - Philipp Muens](https://philippmuens.com/minimax-and-mcts) [↑](#footnote-ref-3)
4. <https://stackoverflow.com/questions/22882110/how-to-implement-movement-in-a-chess-game> [↑](#footnote-ref-4)
5. <https://stackoverflow.com/questions/66082633/minimax-algorithm-with-chess-in-c-sharp-not-working-properly> [↑](#footnote-ref-5)