



**CHARLES
UNIVERSITY**

Checkers (Draughts) Application Document

By Turan Isgandarli

Submitted to- Jirka Sejnoha

Table of Contents

Application Overview and Rules	4
Objective:	4
Setup:	4
Moves for Regular pieces:	4
Moves for Regular pieces:	4
Prerequisites	5
End users:	5
Developer:	5
Developer Guide	6
PlayerColor.cs	6
Creating a board (CheckerBoard.cs)	6
Get winner function (CheckerBoard.cs)	8
SwapTurns (CheckerBoard.cs)	8
Move on the board (CheckerBoard.cs)	9
Minimax algorithm (AIController.cs)	10
Menu functions (MainWindow.xaml.cs)	11
User Guide (Graphical User Interface)	12
The main window contains:	13
Selected Piece:	14
Possible Moves:	15
Diagonally Cut:	15
King:	16
Quit:	16
New Game:	17
Developer Information:	17

List of Figures

Figure 1:PlayerColor.cs File	6
Figure 2: CheckerBoard.cs file.....	7
Figure 3: GetWinner function from CheckerBoard.cs file	8
Figure 4: Swap function from CheckerBoard.cs file	8
Figure 5: MakeMoveOnBoard function from CheckerBoard.cs file	10
Figure 6: Minimax algorithm code from AIController.cs file	11
Figure 7: All menu functions from MainWindow.xaml.cs file	12
Figure 8: Open application using exe file	13
Figure 9: Main application window.....	14
Figure 10: Selected piece having green background.....	14
Figure 11: Possible moves for the selected piece	15
Figure 12: Diagonally cut to take other pieces.....	15
Figure 13: King view, once the piece moves to the other side.....	16
Figure 14: Quit game.....	16
Figure 15: New game, sets to initial game	17
Figure 16: Popup showing the developer's information.	17

Purpose

The purpose of developing the Checkers (Draughts) Windows application in .net is to create a game where users can entertain and increase their mental ability.

It is a strategy and tactic game where the user needs to think ahead, plan their moves, and anticipate their opponent's actions. It helps to improve problem-solving, critical thinking, and concentration skills.

Application Overview and Rules

The traditional two-player board game Checkers is also known as Draughts in some regions. It is played on an 8x8 grid. In order to capture their opponent's pieces and eventually their king, players must use skill and strategy.

Objective:

- To capture all of your opponent's pieces or block them so they can't move legally is the main goal of the game of checkers.

Setup:

- 64 squares make up the 8x8 grid on which the game is played.
- Twelve pieces are placed on the dark squares of the first three rows for the player on the bottom and the last three rows for the player on top for each player at the beginning.
- Typically, players utilize pieces in pairs of two distinct colors, such as red and black or white and black.

Moves for Regular pieces:

- Regular pieces are allowed to diagonally into an empty adjacent square while moving forward one square at a time.
- If there is an empty square immediately after an enemy piece, pieces can grab it by leaping over it diagonally. If there are still jumps remaining, you can keep capturing in the same move.
- An ordinary piece turns into a "king" when it enters the opponent's rear row.

Moves for Regular pieces:

- A king can move diagonally, one square at a time, either forward or backward.
- Kings can also jump over pieces and continue to jump if more captures are possible. They can also grab pieces diagonally forward or backward.

Prerequisites

End users:

1. Windows operating system. (As a user can run directly from the exe file.)
2. Minimum I3 processor with 4 GB RAM for smooth run.

Developer:

3. .NET Framework 4.8.
4. Windows operating system.
5. Integrated development environment (Prefer- Visual studio)
6. Minimum I3 processor with 4 GB RAM for smooth build and run.

Developer Guide

PlayerColor.cs

Setting an enum class to set a group of constants as Red and black, which will be used in other classes.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace CheckersMinimax
8  {
9      27 references
10     public enum PlayerColor
11     {
12         Red,
13         Black
14     }
15 }
```

Figure 1: PlayerColor.cs File.

Creating a board (CheckerBoard.cs)

- Inside CheckerBoard.cs class, I have created a function MakeBoard which sets the main playing area of an 8*8 grid.
- Setting alternative red and black boxes using Brushes.Red and Brushes.Black by putting even odd conditions using the remainder(%) operator.

```
if (Row%2==0)
{
    Brushes.Red
}
Else
{
    Brushes.Black
}
```

- Adding black and red pieces for the first three rows of both sides.

```

51 public void MakeBoard(RoutedEventHandler routedEventHandler)
52 {
53     int count = 0;
54     for (int row = 0; row < 8; row++)
55     {
56         BoardArray.Add(new List<CheckersSquareUserControl>());
57
58         for (int column = 0; column < 8; column++)
59         {
60             CheckersSquareUserControl checkerSquareUC;
61             if (row % 2 == 0)
62             {
63                 if (column % 2 == 0)
64                 {
65                     checkerSquareUC = new CheckersSquareUserControl(
66                         Brushes.Red,
67                         new CheckersPoint(row, column, CheckerPieceType.nullPiece),
68                         routedEventHandler);
69                 }
70             }
71             else
72             {
73                 if (row < 3)
74                 {
75                     checkerSquareUC = new CheckersSquareUserControl(
76                         Brushes.Black,
77                         new CheckersPoint(row, column, CheckerPieceType.BlackPawn),
78                         routedEventHandler);
79                 }
80                 else if (row > 4)
81                 {
82                     checkerSquareUC = new CheckersSquareUserControl(
83                         Brushes.Black,
84                         new CheckersPoint(row, column, CheckerPieceType.RedPawn),
85                         routedEventHandler);
86                 }
87                 else
88                 {
89                     checkerSquareUC = new CheckersSquareUserControl(
90                         Brushes.Black,
91                         new CheckersPoint(row, column, CheckerPieceType.nullPiece),
92                         routedEventHandler);
93                 }
94             }
95         }
96     }
97     if (column % 2 == 0)
98     {
99         if (row < 3)
100         {
101             checkerSquareUC = new CheckersSquareUserControl(
102                 Brushes.Black,
103                 new CheckersPoint(row, column, CheckerPieceType.BlackPawn),
104                 routedEventHandler);
105         }
106         else if (row > 4)
107         {
108             checkerSquareUC = new CheckersSquareUserControl(
109                 Brushes.Black,
110                 new CheckersPoint(row, column, CheckerPieceType.RedPawn),
111                 routedEventHandler);
112         }
113         else
114         {
115             //empty middle spot
116             checkerSquareUC = new CheckersSquareUserControl(
117                 Brushes.Black,
118                 new CheckersPoint(row, column, CheckerPieceType.nullPiece),
119                 routedEventHandler);
120         }
121     }
122     else
123     {
124         checkerSquareUC = new CheckersSquareUserControl(
125             Brushes.Red,
126             new CheckersPoint(row, column, CheckerPieceType.nullPiece),
127             routedEventHandler);
128     }
129 }
130 count++;
131 BoardArray[row].Add(checkerSquareUC);
132 }
133 }
134 }
135 }

```

Figure 2: CheckerBoard.cs file.

Get winner function (CheckerBoard.cs)

GetWinner()- This function helps to return the winner at the end.

I have used the condition when blackCheckersPoint.Count is 0, which means all the pieces of black are out, so in that case, Red will be the winner. Same with the redCheckersPoint.Count == 0 then it returns Black as a winner.

```
141 public object GetWinner()
142 {
143     List<CheckersPoint> redCheckerPoints = GetPointsForColor<IRedPiece>();
144     List<CheckersPoint> blackCheckerPoints = GetPointsForColor<IBlackPiece>();
145
146     // MessageBox.Show(blackCheckerPoints.Count.ToString());
147     if (blackCheckerPoints.Count == 0)
148     {
149         return PlayerColor.Red;
150     }
151     else if (redCheckerPoints.Count == 0)
152     {
153         return PlayerColor.Black;
154     }
155     else
156     {
157         return null;
158     }
159 }
```

Figure 3: GetWinner function from CheckerBoard.cs file.

SwapTurns (CheckerBoard.cs)

SwapTurns()- This function helps to swap the chance for the players.

I have used the if else condition, when the Current player is red, so in the next move the current player will be black and vice versa.

```
377
378 // Swaps the turn.
379
380 1 reference
381 public void SwapTurns()
382 {
383     if (CurrentPlayerTurn == PlayerColor.Red)
384     {
385         CurrentPlayerTurn = PlayerColor.Black;
386     }
387     else
388     {
389         CurrentPlayerTurn = PlayerColor.Red;
390     }
391 }
```

Figure 4: Swap function from CheckerBoard.cs file.

Move on the board (CheckerBoard.cs)

MakeMoveOnBoard()- This function helps to set and return the possible moves available. Multiple if and else conditions are added to make the moves.

Example- `if (!(realDestination.Checker is KingCheckerPiece))`

The above condition is used to set the king when the piece will move to the 1st row of the opponent.

```
if (realDestination.Checker is IRedPiece)
{
    realDestination.Checker = new RedKingCheckerPiece();
}
else
{
    realDestination.Checker = new BlackKingCheckerPiece();
}
```

The above conditions are used to check if it is a red piece moved to another end then it will convert to red king and vice versa for black as well.

```
if (moveToMake.NextMove == null && swapTurn)
{
    //Swap the current players turn
    SwapTurns();
    return true;
}
```

If the player's current move is completed, it will call the SwapTurn function. Which will change the move to the next colour.

```

456 2 references
457 public bool MakeMoveOnBoard(CheckersMove moveToMake, bool swapTurn)
458 {
459     CheckersPoint moveSource = moveToMake.SourcePoint;
460     CheckersPoint moveDestination = moveToMake.DestinationPoint;
461
462     //was this a cancel?
463     if (moveSource != moveDestination)
464     {
465         CheckersPoint realDestination = this.BoardArray[moveDestination.Row][moveDestination.Column].CheckersPoint;
466         CheckersPoint realSource = this.BoardArray[moveSource.Row][moveSource.Column].CheckersPoint;
467
468         realDestination.Checker = (CheckerPiece)realSource.Checker.GetMinimaxClone();
469         realSource.Checker = CheckerPieceFactory.GetCheckerPiece(CheckerPieceType.nullPiece);
470
471         //was this a jump move?
472         CheckersPoint jumpedPoint = moveToMake.JumpedPoint;
473         if (jumpedPoint != null)
474         {
475             //delete the checker piece that was jumped
476             CheckersSquareUserControl jumpedSquareUserControl = this.BoardArray[jumpedPoint.Row][jumpedPoint.Column];
477             jumpedSquareUserControl.CheckersPoint.Checker = CheckerPieceFactory.GetCheckerPiece(CheckerPieceType.nullPiece);
478             jumpedSquareUserControl.UpdateSquare();
479         }
480
481         //Is this piece a king now?
482         if (!(realDestination.Checker is KingCheckerPiece)
483             && (realDestination.Row == 7 || realDestination.Row == 0))
484         {
485             //Should be a king now
486             if (realDestination.Checker is IRedPiece)
487             {
488                 realDestination.Checker = new RedKingCheckerPiece();
489             }
490             else
491             {
492                 realDestination.Checker = new BlackKingCheckerPiece();
493             }
494         }
495
496         //Is this players turn over?
497         if (moveToMake.NextMove == null && swapTurn)
498         {
499             //Swap the current players turn
500             SwapTurns();
501             return true;
502         }
503         else
504         {
505             return false;
506         }
507     }
508     return false;
509 }
510

```

Figure 5: MakeMoveOnBoard function from CheckerBoard.cs file.

Minimax algorithm (AIController.cs)

Minimax algorithm works on the recursive or backtracking algorithm which is used in decision-making. In the event that the opponent is likewise playing optimally, it offers the player an optimal move. Recursion is used by the Mini-Max algorithm to search across the game tree.

MinMaxStart()-

- This is the main minimax algorithm code. This method helps to trigger the algorithm and find the best move for the player.
- If two or more moves have having the same value, so in that case random() function will call and will pick any of the moves.
- I have used Thread.Sleep(200); method, in order to take a 2-second break for computer movement.
- `foreach` (CheckersMove move in possibleMoves) Using foreach loop, I am iterating all the possible moves, and inside this loop, there are more conditions to find the best move out of all possible moves.
- It will return the best move after doing all the computations like depth search.

```

35 public static CheckersMove MinimaxStart(CheckerBoard board)
36 {
37     int alpha = int.MinValue;
38     int beta = int.MaxValue;
39     thinking = true;
40
41     List<CheckersMove> possibleMoves = board.GetMovesForPlayer();
42     List<int> values = new List<int>();
43
44     Logger.Info(string.Format("Max is {0}", board.CurrentPlayerTurn));
45
46     if (possibleMoves.IsNullOrEmpty())
47     {
48         return null;
49     }
50
51     foreach (CheckersMove move in possibleMoves)
52     {
53         Thread.Sleep(200);
54
55         CheckersMove moveToMake = move;
56         CheckerBoard boardToMakeMoveOn = board;
57         do
58         {
59             Logger.Debug("Board Before");
60             Logger.Debug(boardToMakeMoveOn.ToString());
61
62             boardToMakeMoveOn = (CheckerBoard)boardToMakeMoveOn.GetMinimaxClone();
63             boardToMakeMoveOn.MakeMoveOnBoard((CheckersMove)moveToMake.GetMinimaxClone());
64             moveToMake = moveToMake.NextMove;
65
66             Logger.Debug("Board After");
67             Logger.Debug(boardToMakeMoveOn.ToString());
68         }
69         while (moveToMake != null);
70
71         values.Add(Minimax(boardToMakeMoveOn, Settings.AIDepth - 1, alpha, beta, false, board.CurrentPlayerTurn));
72     }
73
74     int maxHeuristics = int.MinValue;
75     foreach (int value in values)
76     {
77         if (value >= maxHeuristics)
78         {
79             maxHeuristics = value;
80         }
81     }
82
83     //filter the list of moves based on max value
84     List<CheckersMove> bestMoves = new List<CheckersMove>();
85     for (int i = 0; i < values.Count; i++)
86     {
87         if (values[i] == maxHeuristics)
88         {
89             bestMoves.Add(possibleMoves[i]);
90         }
91     }
92
93     counter = 0;
94     thinking = false;
95     Logger.Info("Node Values: " + string.Join(", ", values.Select(x => x.ToString()).ToArray()));
96     return bestMoves[Rng.Next(bestMoves.Count)];
97 }
98
99

```

Figure 6: Minimax algorithm code from AIController.cs file.

Menu functions (MainWindow.xaml.cs)

RestartGame()- This function helps to abort the current game using *Abort()* method. Later it will reset the game to the initial condition using *InitializeCheckers()* function.

Quit()- This function helps to quit(resign) the game and it will show the message as the opponent wins using *MessageBox.Show("Black wins ")*; to display the popup window. Apart from that, it will abort the current game and initialize the game as a new game.

DV()- This function helps to display the developer information using *messageBox.Show()* window form.

```

482 1 reference
483 private void RestartGame(object sender, RoutedEventArgs e)
484 {
485     if (aiThread != null)
486     {
487         aiThread.Abort();
488     }
489     InitializeCheckers();
490 }
491
492 /// Quits the game.
493
494 1 reference
495 private void Quit(object sender, RoutedEventArgs e)
496 {
497     MessageBox.Show("Black wins ");
498     if (aiThread != null)
499     {
500         aiThread.Abort();
501     }
502     InitializeCheckers();
503 }
504
505 /// Developer Information
506
507 1 reference
508 private void DV(object sender, RoutedEventArgs e)
509 {
510     MessageBox.Show("Developed by Turan Isgandarli");
511 }

```

Figure 7: All menu functions from MainWindow.xaml.cs file.

User Guide (Graphical User Interface)

Users can run the application by just clicking on the .exe file available in the debug/release folder of the application.









 > Draughts > CheckersMinimax > bin > Debug				
	Name	Date modified	Type	Size
—	 CheckersMinimax	06-09-2023 02:45	Application	2,886 KB
✦	 CheckersMinimax.exe.config	03-09-2023 01:39	CONFIG File	3 KB
✦	 CheckersMinimax.pdb	06-09-2023 02:45	Program Debug D...	160 KB
✦	 CheckersMinimax_03-09-2023	03-09-2023 15:03	Text Document	17 KB
✦	 CheckersMinimax_04-09-2023	04-09-2023 22:01	Text Document	3 KB
✦	 CheckersMinimax_05-09-2023	05-09-2023 04:57	Text Document	84 KB
✦	 CheckersMinimax_06-09-2023	06-09-2023 13:09	Text Document	57 KB

Figure 8: Open application using exe file.

The main window contains:

- **Menu-** Menu has 3 buttons placed horizontally:
 1. **New Game-** New Game will start.
 2. **Developer information-** This tab will show the developer information.
 3. **Quit-** This button helps to quit (Resign) the game. By clicking it, another player will automatically win.
- **8*8 checkers board-** It is the main screen to play the checkers game, where Black is the computer side and red is the player 1 side.
- **Title bar-** The title bar will show the dynamic content like which player needs to move the piece that is red or black.
- **Textblock:** Textblock is the area on the right side of the main screen where I have added the text “Welcome to Draughts”

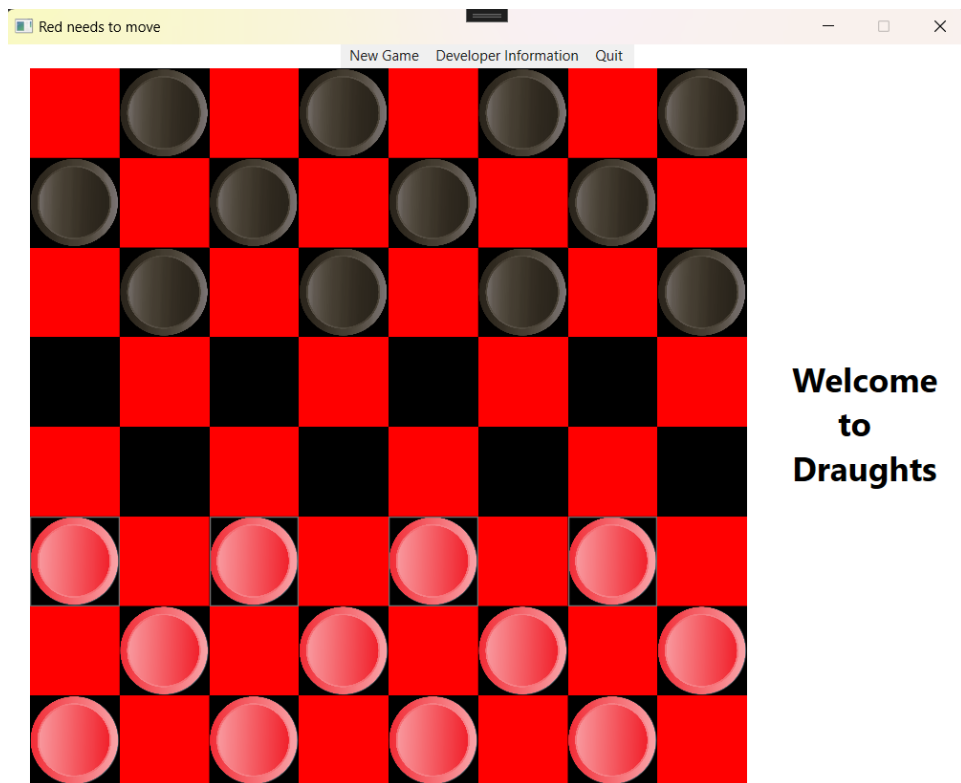


Figure 9: Main application window.

Selected Piece:

If users select and hold any piece then, that box will be highlighted in the green border, in order to distinguish between the selected piece and the non-selected piece.

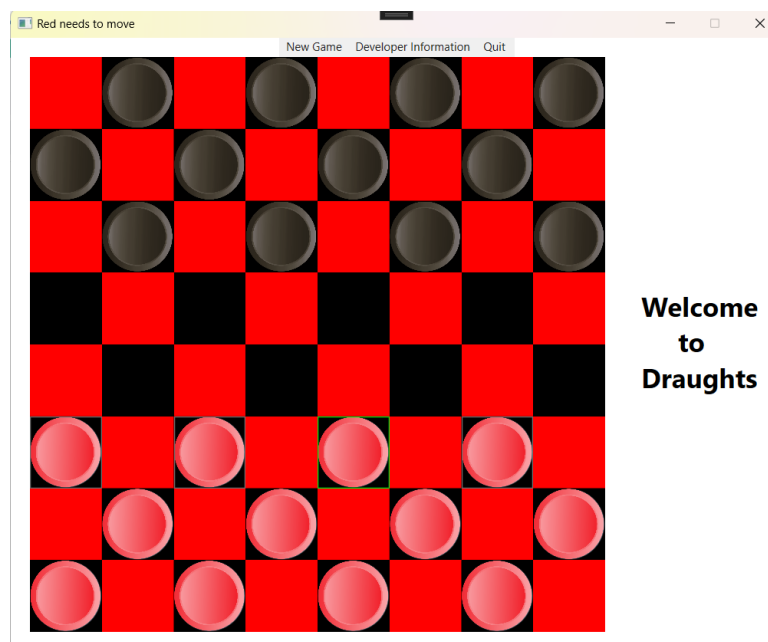


Figure 10: Selected piece having green background.

Possible Moves:

When users release the hold on the selected piece then possible moves will be highlighted in the aqua colour.

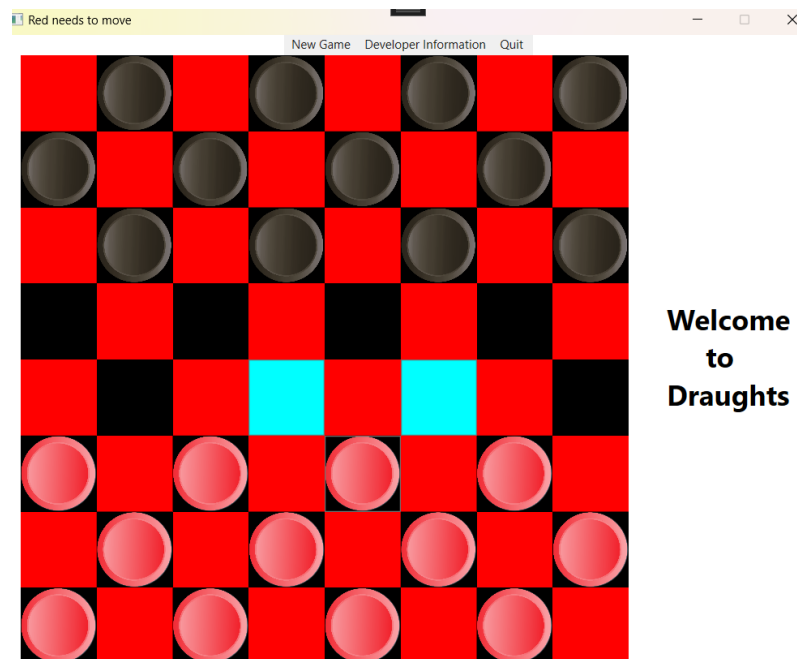


Figure 11: Possible moves for the selected piece.

Diagonally Cut:

As per the rules of checkers, the Player can remove the opponent's piece from the board by jumping them if empty space is available next to the diagonal. Please check the below image.

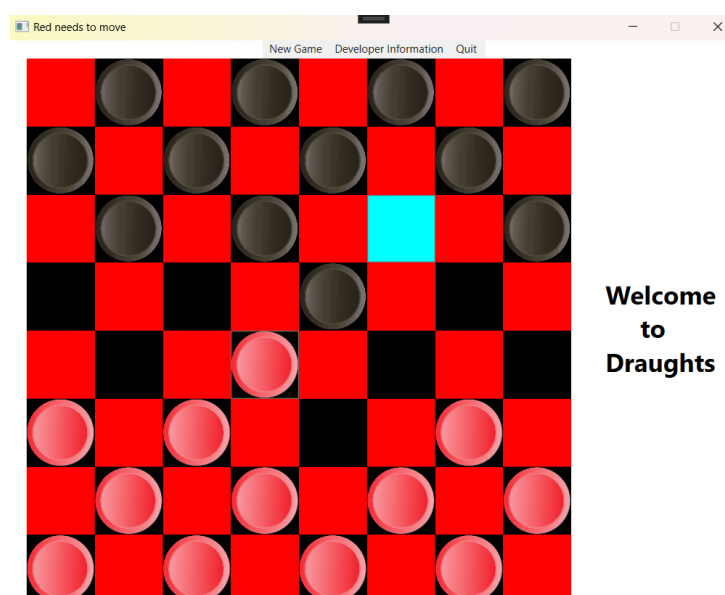


Figure 12: Diagonally cut to take others piece.

King:

As per the rules of checkers, if the opponent moves to the other side, normal piece will convert into king.

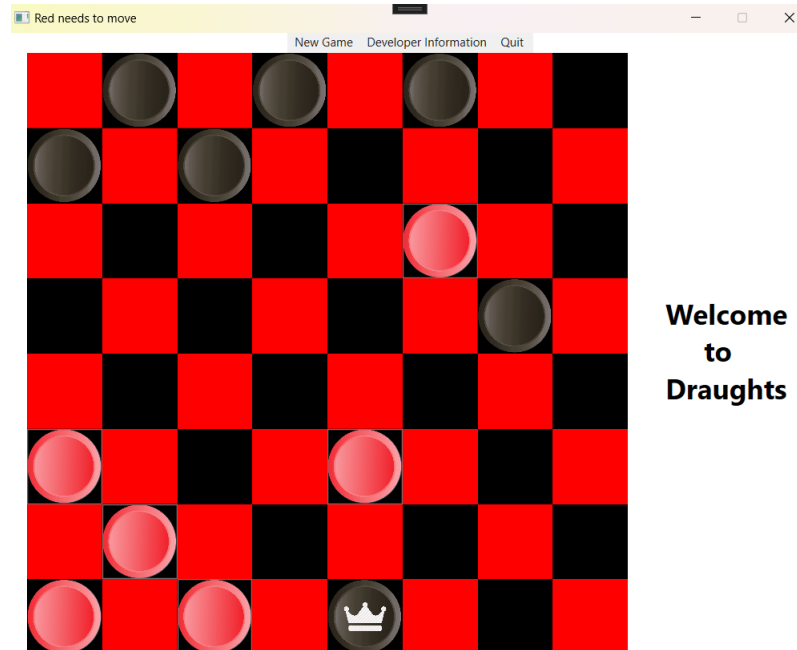


Figure 13: King view, once piece move to other side.

Quit:

If the Player clicks on the quit, then the opponent will win the game automatically.

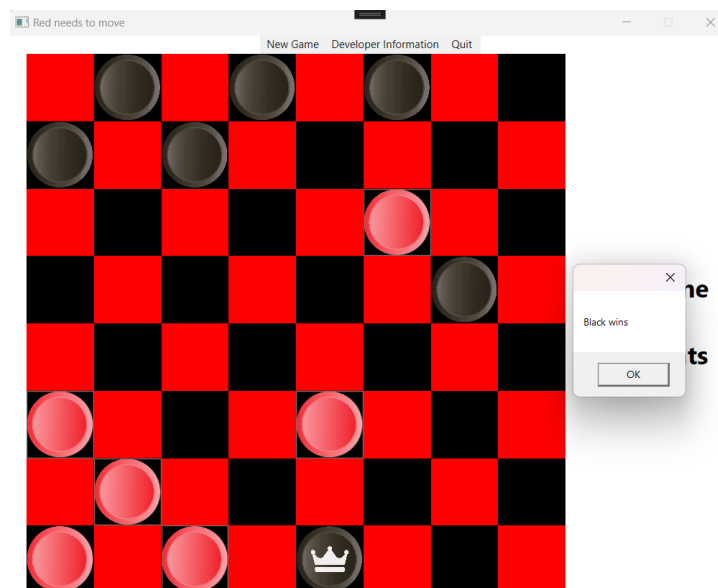


Figure 14: Quit game.

New Game:

If the Player clicks on the New Game button, then the board will reset all the pieces and new game will start.

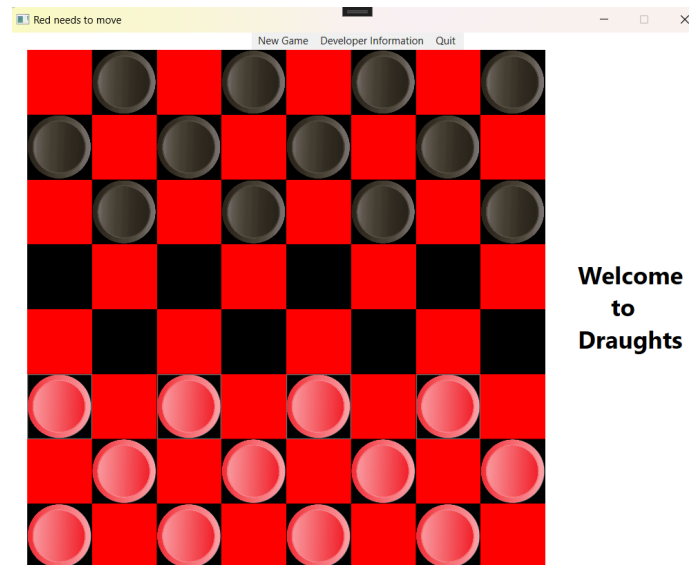


Figure 15: New game, sets to initial game.

Developer Information:

If the Player clicks on the Developer information button, then one dialog box will open and it will show the developer information.

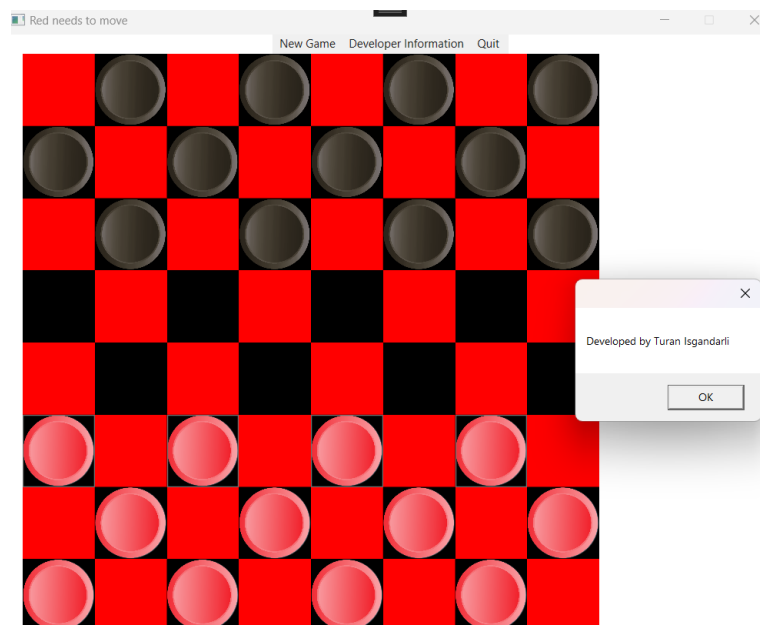
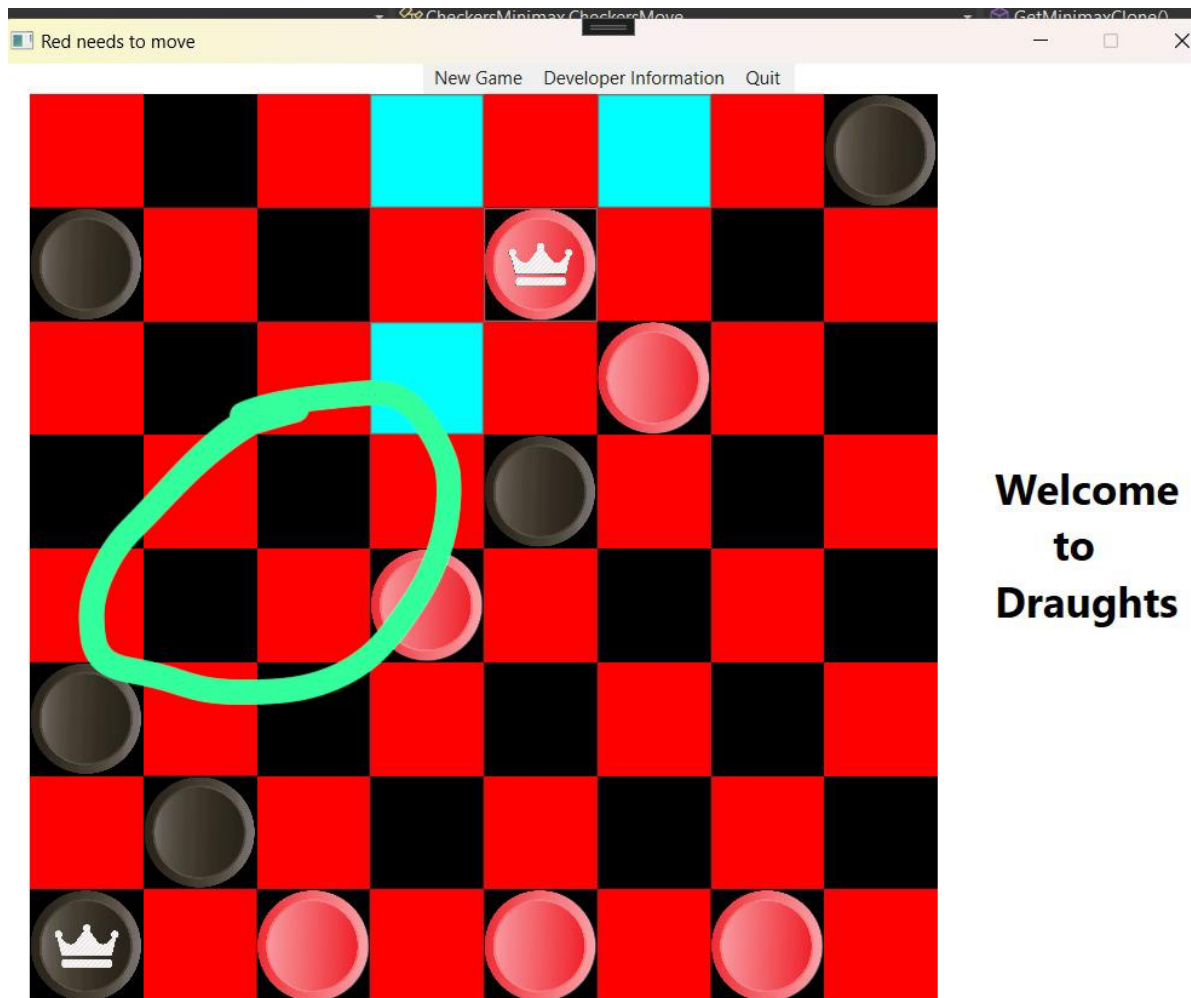


Figure 16: Popup showing developer's information.

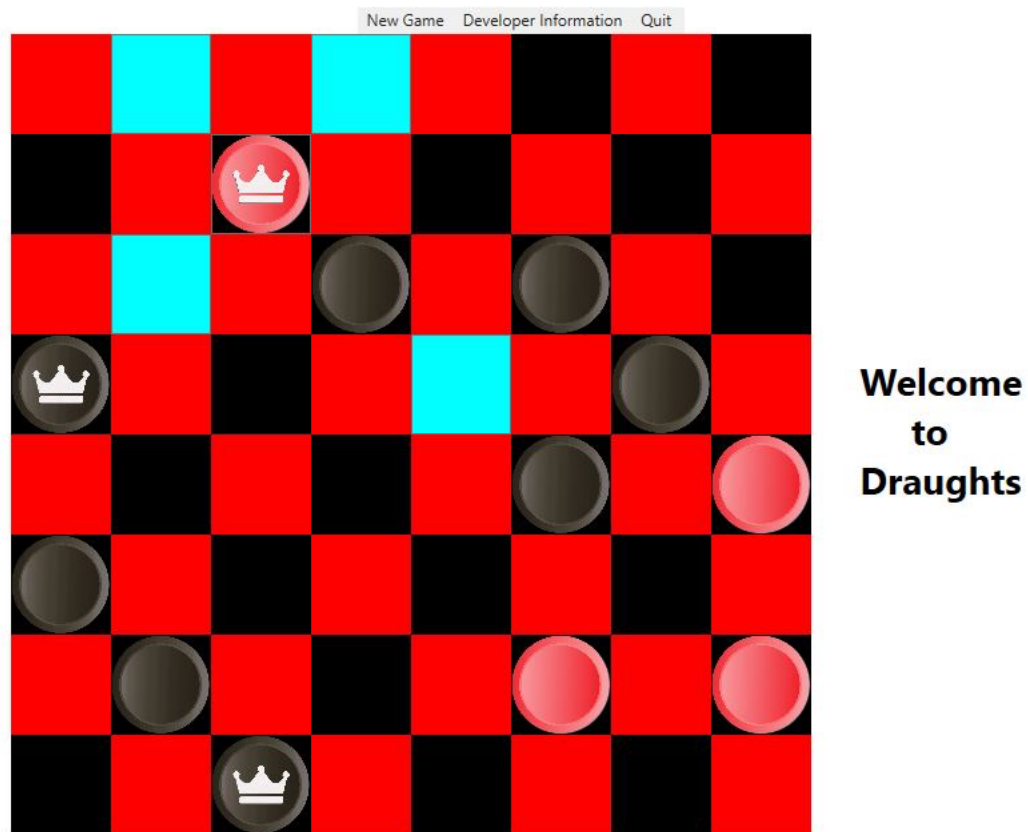
Updated version details:

King(Dama) moves **before**:

It should generate all possible moves.



After: I newly added 2 function in Kingchekers piece class and I return the result to get possible moves.



```
private List<CheckersMove> GetBishopLikeMoves(CheckersPoint currentLocation, CheckerBoard checkerBoard)
{
    List<CheckersMove> list = new List<CheckersMove>();

    // Define the directions for bishop-like movement (diagonals)
    int[] verticalModifiers = { -1, -1, 1, 1 };
    int[] horizontalModifiers = { -1, 1, -1, 1 };

    for (int i = 0; i < 4; i++)
    {
        int row = currentLocation.Row + verticalModifiers[i];
        int col = currentLocation.Column + horizontalModifiers[i];

        while (row >= 0 && row < 8 && col >= 0 && col < 8)
        {
            CheckerPiece pieceOnPoint = checkerBoard.BoardArray[row][col].CheckersPoint.Checker;

            if (pieceOnPoint == null || pieceOnPoint is NullCheckerPiece)
            {
                // The destination point is empty, we can go here
                list.Add(new CheckersMove(currentLocation, new CheckersPoint(row, col)));
            }
        }
    }
}
```